Just-in-Time Data Structures

A Language Design Approach to Promote the Shift from Choosing a Single Data Structure to Choosing a Set of Data Representations

Just-in-Time Data Structures

A Language Design Approach to Promote the Shift from Choosing a Single Data Structure to Choosing a Set of Data Representations

Mattias De Wael

A dissertation submitted in fulfilment of the requirements for the award of the degree of Doctor of Science

May 2016

Promotors
Prof. Dr. Wolfgang De Meuter
Prof. Dr. Jennifer B. Sartor

Vrije Universiteit Brussel
Faculty of Science and Bio-Engineering Sciences
Department of Computer Science
Software Languages Lab

Jury:
Prof. Dr. Viviane Jonckers (chair), SOFT, Vrije Universiteit Brussel
Prof. Dr. Peter Vrancx (secretary), AI, Vrije Universiteit Brussel
Prof. Dr. Wolfgang De Meuter (promotor), SOFT, Vrije Universiteit Brussel
Prof. Dr. Jennifer B. Sartor (promotor), SOFT, Vrije Universiteit Brussel
Prof. Dr. Jan Hidders, WISE, Vrije Universiteit Brussel
Prof. Dr. Philippe Cara, DWIS, Vrije Universiteit Brussel
Prof. Dr. Richard Jones, School of Computing, University of Kent (UK)
Prof. Dr. Yvonne Coady, Computer Science Department, University of Victoria (Canada)

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / Fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 9789492312105
NUR 980 989

Acknowledgements:
Het werk in deze dissertatie werd gefinancierd door het Agentschap voor Innovatie door Wetenschap en Technologie.
The work in this dissertation has been funded by the Agency for Innovation by Science and Technology in Flanders (IWT).

Copyright:
Alle rechten voorbehouden. Niets van deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, electronisch of op welke andere wijze ook, zonder voorafgaande schriftelijke toestemming van de auteur.

All rights reserved. No parts of this book may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without prior written permission from the author.

©2016 Mattias De Wael

Misschien wel leuk om te vertellen:
"Ik heb het geschreven met in m'n
achterhoofd het thema dat bewustzijn
uiteindelijk niet bevrijdt maar veel
meer de muren van de gevangenis
zichtbaar maakt".

THEO MAASSEN

Acknowledgements

First of all, I have to thank my jury (Viviane Jonckers, Peter Vrancx, Wolfgang De Meuter, Jennifer B. Sartor, Jan Hidders, Philippe Cara, Richard Jones, and Yvonne Coady) and my promotors (Wolfgang and Jennifer) in particular. They have read my text and commented on my work in order to lift it to a higher level. I also have to thank the Agency for Innovation by Science and Technology in Flanders (IWT) for providing me with two times two years of funding. Further, I want to thank the people of the administration offices of DINF and SOFT for helping me fight the bureaucracy at the VUB and beyond. The other people that I want to thank for contributing to the scientific part of the Ph.D endeavour are, in anti-chronological order, Theo and Eline (for proof reading), Lode and Joeri (also for proof reading), Janwillem (for being the soundboard and participant in many discussions), Dries and Thierry (and later on Christophe) (for `find . -exec grep -l -e '*problem' \; | solve`), Stefan (for being a genuine co-author in most of my publications), David Ungar (for teaching me to be thorough and exact), and Tom (for guiding and advising me during my final master's year and my first Ph.D. years). Finally, I would like to thank Thibalt for creating a cover for this dissertation that is maybe even more beautiful and elegant than the idea of just-in-time data structures itself.

On a more personal note I want to thank my parents and my sister, just for being the most important people in the world for me; Eline (Ponnetje), just for "de zoentjes" and because I love her; Ellen and Sanne, just for being their supportive-selves; "de mannen van 86", just for being there day and night, through thick and thin since for what seems like forever; "de mannen van den voetbal", just for sharing beers in the Ghelamco Arena with me (for instance that time when KAA Gent won their first championship); SOFT′, just for providing a relaxed atmosphere; JW, just for never suggesting a "muziekje van de dag" and letting me do all the hard work; Laure, just for so many small things (for instance sticking the "defence in progress"-sign on the door); Simon, just for the support (although for the wrong team); the IWT-triangle, just for being the best batch of Ph.D. students to have ever wandered the 10th floor since the existence of PROG, SSEL, and SOFT.

Mattias De Wael[1]
May 2016

[1] A version of the dissertation without typos is available on request.

Abstract

In 1978, Niklaus Wirth surveyed the set of most commonly used algorithmic tasks in function of the accompanying data structures and vice versa. He published this survey in a book titled "Algorithms + Data Structures = Programs". After more than three decades, it can still be considered to be an accurate title: Even today, software engineering practices focus on finding the "best" data structure for a program (i.e., the interplay of a set of algorithms). A single program often consists of multiple algorithms that operate on the same data. In such a program, the "best" data structure is the data structure that yields the best *overall performance*, where performance can be interpreted as shorter execution times, lower memory consumption, higher throughput, or any other desirable non-functional feature.

However, we observe that for some programs, relying on a *single data representation* can be less performant than *changing the representation* of the data at runtime. Hence, we argue in favour of using the "best" data representation for each phase of a program, and to enable changes between the different data representations at runtime. We call this *just-in-time representation changes*. Implementing representation changes is a tedious and error-prone task, for which little to no support exists in contemporary programming languages.

In this dissertation we explore the idea of just-in-time representation changes and taxonomize the design space of programming languages in which such representation changes can be implemented. From this taxonomy, we identify a vacuum in the landscape of programming languages, to allow developers to express data representations changes in order to realise *non-functional features*, such as for instance improving performance. Hence, the core contribution of this work is the design of a new programming language called JITds, which fills this vacuum. Besides representation changes themselves, another core design feature of JITds is that it is possible to disentangle general application logic from logic responsible for the representation changes. In short, with JITds we want to promote the shift from choosing a single data structure to choosing a set of data representations, in order to improve performance.

Samenvatting

Niklaus Wirth schreef in 1978 een boek waarin hij de meest voorkomende algoritmen en datastructuren beschrijft in functie van elkaar. De titel van dit boek is "Algorithms + Data Structures = Programs". Dit is een titel die na meer dan 30 jaar nog steeds goed gekozen blijkt te zijn, want ook vandaag gaan softwareontwikkelaars nog steeds op zoek naar de "beste" datastructuur voor een programma. Vaak bestaat een programma uit een samenspel van verschillende algoritmen die inwerken op dezelfde data. In zo een programma is de "beste" datastructuur, die datastructuur die *over het algemeen* de beste performantie oplevert. Hierbij kan performantie gezien worden als kortere executietijd, lager geheugenverbruik, snellere reactietijd of om het even welke combinatie van gewenste, niet-functionele eigenschappen.
We stellen echter vast dat voor sommige programma's de performantie kan verbeterd worden als tijdens de uitvoering van het programma gewisseld wordt van gekozen datastructuur, i.p.v. zich te beperken tot het gebruik van één datastructuur. We stellen dus voor om voor elke fase in een programma een "beste" datastructuur te kiezen en deze te laten wisselen. We noemen dit *just-in-time representation changes*. Het implementeren van *just-in-time representation changes* is een moeilijke opgave die bovendien zeer foutgevoelig is. We merken op dat hedendaagse programmeertalen weinig tot geen abstracties voorzien om *just-in-time representation changes* correct te implementeren.
In dit proefschrift spitten we het idee van *just-in-time representation changes* verder uit in de context van programmeertaalontwerp. We brengen het landschap van de programmeertalen, waarin zulke just-in-time representation changes kunnen geïmplementeerd worden, in kaart. We stellen echter vast dat er een vacuüm bestaat in dit landschap: het uitdrukken van *niet-functionele representation changes* wordt door geen enkele programmeertaal ondersteund. Daarom introduceren we in dit proefschift JITds, een nieuwe programmeertaal die dit vacuüm moet opvullen. Omdat JITds als doel heeft *niet-functionele representation changes* toe te laten is een belangrijke eigenschap van JITds dat de logica die de representation changes (niet-functioneel) aanstuurt, volledig losgekoppeld kan worden van de de logica van het programma (functioneel) zelf. Het doel van JITds is de focus te verleggen van het kiezen van één datastructuur naar het kiezen van een verzameling van datarepresentaties, met het oog op het verbeteren van de performantie.

Contents

Chapter 1

Introduction

In 1978, Niklaus Wirth surveyed the set of most commonly used algorithmic tasks in function of their accompanying data structures and vice versa. He published this survey in a book titled "Algorithms + Data Structures = Programs" [88]. After more than three decades, it can still be considered to be an accurate title. Even today, software engineering practices focus on finding the "best" data structure for a program. A single program often consists of a set of algorithms that operate on the same data. In such a program, the "best" data structure is then considered to be the data structure that yields the best *overall performance*, where performance can be interpreted as shorter execution times, lower memory consumption, higher throughput, or any other desirable non-functional feature.

There exist programs, however, where relying on a *single data representation* is less efficient than *changing the representation* of the data at runtime. In this dissertation, we argue in favour of using the "best" data representation for the different phases of a program when the benefits of changing the data representation at runtime exceeds the cost of the actual representation change. We call these *just-in-time representation changes*. Even though the idea of just-in-time representation changes seems like a straightforward approach towards improving the performance of programs, it is not often implemented in actual software or libraries. We argue this is the case because implementing representation changes is a tedious and error-prone task, for which little to no support exists in contemporary programming languages.

In this dissertation we explore this idea of just-in-time representation changes and present a taxonomy of programming language features to express representation changes. This taxonomy is based on a survey of programming languages, and from this survey we identify a lacuna in the landscape of programming languages. Today, no programming language exists that enables developers to express data representation changes in order to realise *non-functional features*, such as improving the performance. Hence, the core contribution of this work is the design of a new programming language, **JITds**, which fills up this lacuna. Besides enabling representation changes, another core design feature of JITds is that it is possible to disentangle general application logic from the crosscutting concern of representation changes.

1.1 Problem Statement

In the seventies and eighties, a significant amount of effort was needed to make software efficient. CPUs were slow, memory banks were small, and these scarce resources had to be put to best use. When Wirth [88] surveyed the set of most commonly used algorithmic tasks in combination with their accompanying data structures, these combinations were often optimised for performance, generally for faster execution or lower memory consumption, or a combination thereof. As a consequence of Moore's law, writing efficient programs became less important for the developers, because the performance of their programs was virtually being doubled every three years by the hardware engineers. Technically, Moore's law applies to transistor densities only, but also in related areas, such as clock speeds, the same growth has been observed until approximately the year 2000. Since the turn of the century, this so-called free lunch is over and developers will need to take performance back into account [79].

Understanding and predicting the performance characteristics of an application that is run on today's hardware is not longer trivial. Today's hardware architectures have become evermore complex, e. g., we have non-uniform memory access (NUMA) and branch prediction, but also multi-core and even many-core architectures. A great deal of programming research effort has gone into investigating the potential of this parallel hardware, e. g., [55, 24, 22, 19] to mention just our own work. However, also on a single core, performance gains can still be realised.

When writing software today, the focus lays with finding the "best" representation for the data, i. e., the representation that yields the best *overall performance*. There exist programs, however, where using this single data representation is less efficient than changing the representation of the data during the execution of the program [21, 20]. In these programs, the cost of these representation changes (e. g., execution time) is earned back many times over by the reduction in cost of using an alternate representation for the remainder of the execution.

Implementing representation changes is a difficult and error-prone task and today no programming languages exist that help developers with this. One could argue that implementing representation changes in dynamic (object-oriented) languages is relatively easy, e. g., by using `become` in SmallTalk [36] or dynamic reparenting in Self [12]. However, dynamic languages cannot provide static guarantees about which messages an object can understand, let alone in a setting where the representation of an object can change at runtime. A small quorum of statically-typed languages exist that 1) support representation changes and 2) guarantee type safety, e. g., Gilgul [15], Fickle [27], and Plaid [78]. The limitation of these languages is that the representation changes are an explicit part of the type signature of the data. Consider, for instance, the class `File` for which two representations exist: `OpenFile` and `ClosedFile`. To open a closed file, the class `ClosedFile` provides the `open()` method which changes the representation from `ClosedFile` to `OpenFile`. In Plaid[1], for instance, such an `open()` method would have the type `[ClosedFile»OpenFile]`. Here, the explicit information is in-

[1] This introductory example uses syntax from the programming language Plaid [78], which is discussed in section 9.1.5.

teresting because the representation change is introduced to model *functional features* of the software, here opening a closed file. However, when the representation change is introduced to implement *non-functional* features, such as reducing execution time, adding *transition signatures* clutters the application logic.

Thus, what is missing in the landscape of programming languages is a programming language that, first, enables representation changes, and second, provides sufficient guarantees w.r.t. the expected behaviour (i. e., type safety) without cluttering the general application logic.

1.2 Separating Interface from Implementation: History and Terminology

"Separating interface from implementation" is a well-known principle in software engineering (e. g., from [64]). Before the advent of object-technology, it was a mechanism for allowing changes in the implementation in later stages of the development cycle without needing to update the code that uses the interface, e. g., header files in C. Later, with the advent of object-technology, late binding was introduced which enabled different implementations to coexist at runtime. Today, most (object-oriented) programming languages allow the representation of a data object to be chosen at allocation time. This evolution is a clear example of *ever-later binding times* in the history of software development (paraphrased from Ralston et al. [67]).

In this context, the four terms we use most in this dissertation are *data object*, *abstract data type*, *data interface*, and *data representation*. Without proper definitions, confusion or ambiguity may arise due to multiple pre-existing meanings of these terms. Hence, to avoid confusion, we find it important to define these terms explicitly here.

> A **Data Object** is a referable value, e. g., by using an identifier. Concretely, a data object is either a primitive data object (e. g., a number or a boolean) or a compound data object (e. g., cons-cell, struct, array, ...) [1].

> An **Abstract Data Type** (ADT) defines a class (i. e., category) of data objects. This class is completely characterised by the available operations. Hence, an abstract data type is defined by its characterising operations (definition taken from [53]).

> A **Data Interface** is the set of characterising operations of an abstract data type (i. e., a class of data object) or a concrete data object. In other words: 1) a data interface is the set of characterising operations applicable to a data object to retrieve or update information; and 2) a data interface defines an abstract data type.

> A **Data Representation** is the concrete realisation or implementation of a data interface (i. e., often realised in a concrete programming language).

Note that these four terms denote well-known and well-understood concepts. For the latter two, we explicitly choose to use our own terminology, in order to avoid ambiguity with other, sometimes more popular, terminology. We now argue why we opt to use *Data Object*, *Data Interface*, and *Data Representation*; rather than the popular alternatives.

Data Object instead of Data or Object In the context of computer science, (plain) "data" can have the connotation of being unstructured and being only indirectly accessible. In the context of object-oriented programming, "object" has a well-defined meaning, i. e., encapsulated state and behaviour. Hence, if we want to denote an *referable piece of data* we cannot use the words "data" or "object", but use the compound term "data object" instead.

Data Interface instead of Interface The term "interface" is overloaded. In Java, for instance, an `interface` is a language construct introduced to support a safe but restricted form of multiple inheritance. Because JITds has the look-and-feel of Java, we explicitly want to avoid the confusion between "data interface", as introduced above, and the language construct "interface" as it is known in Java.

Data Representation instead of Implementation In the context of programming language research, "implementation" can denote two completely different things: either the implementation of a data interface or algorithm, or the implementation of a programming language. We therefore refrain from using "implementation" to mean the realisation of a data interface, but use the term "data representation" instead.

1.3 Research Design

This research is motivated by two observations:

O1 The expertise of the average software engineer is no longer sufficient for him to predict and understand the performance of the software he develops.

O2 Applications exist where the use of any single fixed data representation is less efficient than changing the data representation at runtime.

In chapter 2 we extensively discuss **O1** and **O2**: We explain how difficult performance engineering has become and we show how various programs in various domains benefit from online representation changes. We conclude that the ideal programming language adequately supports developers in safely implementing non-functional representation changes (cf. **O2**). We focus on non-functional representation changes that improve the performance. Furthermore, the ideal programming language allows performance experts to introduce representation changes without interfering with the developer of the application logic, in order to relieve the average developer from the cognitive burden of implementing representation changes (cf. **O1**). In other words, such the ideal programming language must meet the following three requirements:

R1 The language must enable data objects to change their data representation at runtime.

R2 The language must provide type safety guarantees in the face of representation changes.

R3 The language must allow programmers to disentangle application logic from representation changing logic.

We have performed an extensive survey of programming languages and conclude that, to the best of our knowledge, such a programming language does not exist today [20]. Hence, the main contribution of this work is the design (and implementation) of the programming language JITds that meets **R1** (see chapter 4), **R2** (see chapter 5), and **R3** (see chapter 6).

1.4 Contributions

The goal of this research is to provide programming language support for non-functional representation changes, primarily intended to improve overall performance. The path to this goal is described in section 1.3. This dissertation makes two major contributions, the latter of which can be subdivided into four sub-contributions:

- A taxonomy of programming language features to express representation changes;
- The design of the programming language JITds, which supports non-functional representation changes (cf. **R1, R2, R3**).
  - We propose a set of language features to support (non-functional) representation changes (**R1**).
  - We define an operational semantics for JITds (**R1**).
  - We prove the soundness of the type system of JITds (**R2**).
  - We propose a set of language features to decouple representation change logic from general application logic (**R3**).

The technical contributions of this work are the *two prototype implementations of the language* JITds, which we discuss in chapter 7. The first implementation, JITds-Java, transpiles JITds code to Java. The second implementation, JITds-C, compiles JITds code to a set of C-function calls that use our custom-built object runtime engine. We further use the JITds-Java compiler in chapter 8, where we compare the execution times and the code structure of five synthetic benchmark programs, written in JITds, against the same programs written in plain Java.

1.5 Supporting Publications

The following publications support the key ideas of this dissertation:

Just-in-Time Data Structures [20] (Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B. Sartor, and Wolfgang De Meuter). In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Onward! 2015)
This paper presents the language JITds and the constructs needed to support just-in-time representation changes. Two example programs are discussed where changing the representation of a data object results in improved performance.

Data Interface + Algorithms = Efficient Programs: Separating logic from representation to improve performance [21] (Mattias De Wael, Stefan Marr, and Wolfgang De Meuter). In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (ICOOOLPS '14)
This paper identifies the need for a structured approach to handle the transitions between different data representations and the descriptive programming constructs to express which representation fits a program fragment best.

Just-in-Time Data Structures: Towards Declarative Swap Rules [18] (Mattias De Wael). In *Proceedings of the 13th International Workshop on Dynamic Analysis* (WODA '15)
This paper presents just-in-time data structures to the community of researchers in dynamic analysis. It explains how internal swap rules are currently naively implemented, and argues in favour of a more advanced implementation. Concretely, the paper advocates to take advantage of the declarative nature of swap rules, and to rely on decades of work in this area to obtain a more efficient implementation.

The following publications provide factual evidence for certain claims made in this dissertation:

Partitioned Global Address Space Languages [19] (Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, Wolfgang De Meuter). In *Computing Surveys, Volume 47 , Issue 4* (CSUR)
This paper presents a taxonomy of *partitioned global address space (PGAS) languages*. Many of these languages pay special attention to the distribution and structure of regular data, e. g., as summarised in section 3.3 of this paper.

When Spatial and Temporal Locality Collide: The Case of the Missing Cache Hits [23] (Mattias De Wael, David Ungar, and Tom Van Cutsem). In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools* (ICPE '13)
This paper reports on a multi-person, multi-month endeavour to pin-point the origin of unexpected performance characteristics of a simple benchmark program. This paper

provides factual evidence for our claim that performance engineering is difficult and that understanding the performance characteristics of a larger piece of software is no longer feasible for the average programmer.

Fork/join parallelism in the Wild: Documenting Patterns and Anti-patterns in Java Programs using the Fork/Join Framework [22] (Mattias De Wael, Stefan Marr, and Tom Van Cutsem). In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools* (PPPJ '14)
This paper studies a corpus of 120 Java fork/join applications and identifies best-practice patterns and anti-patterns in those applications. This paper provides factual evidence for our claim that the average programmer uses the path of the least resistance. To get the job done, many developers write the easiest possible code, without necessarily being aware of the implications, e. g., performance penalty of using a certain feature.

1.6 Limitations

There are two main limitations of this work. First, we have the technical limitation that JITds was given two *prototype* implementations. The objective of these implementations was to verify the feasibility of implementing the new programming features proposed in this work, and not on the efficiency and completeness of the implementation. In the future work section (see section 10.3), we already hint at possible improvements over the current implementations. Second, JITds has not been evaluated in the context of real-world applications, but rather was evaluated in the context of a set of synthetic benchmark programs. In chapter 8, we explain why we make this choice.

1.7 Roadmap

The remainder of this dissertation is organised as follows.

Chapter 2: Motivation Chapter 2 motivates the search for a programming language with support for just-in-time representation changes and in which it is possible to disentangle general application logic from the crosscutting concern of representation changes. This motivation is twofold. First, section 2.1 provides factual evidence for **O1**, which states that performance engineering has become too complex for the average programmer. From this claim we conclude that the average programmer prefers to hide the complexity of performance engineering aspects in software engineering, or prefers someone else (e. g., a performance expert) to worry about performance engineering. This can be realised, for instance, by disentangling general application logic from logic responsible for non-functional requirements such as performance. Second, in section 2.2 we illustrate that there exist programs where changing the representation of the data during the execution of the program results in better overall performance than any version of the program with a fixed data representation (cf. **O2**). Thus, combining the arguments from both section 2.1 and section 2.2, we motivate the need for a programming language with

support for just-in-time representation changes and in which it is possible to disentangle general application logic from logic responsible for the representation changes (cf. **R1**, **R2**, and **R3**).

Chapter 3: Matching Data and Computation In chapter 3 we define more precisely what we mean by *just-in-time representation changes*. Based on a survey of contemporary programming languages with support for representation changes, we identify *seven dimensions of representation changes*. Furthermore, from this survey we identify that no programming language exists today that allows developers to express data representation changes to realise *non-functional requirements*, such as improving performance.

Chapter 4: Designing Just-in-Time Data Structures: How to Change the Representation Chapter 4 is the first of the two chapters that discuss the *language design* spectrum of this work. Chapter 4 focuses on the language constructs in JITds that are introduced to enable just-in-time representation changes, i. e., *"How to Change the Representation"*, **R1**. Furthermore, chapter 4 places these language features in relation to well-known and well-studied concepts such as *dynamic object reclassification* and *multiple inheritance*.

Chapter 5: JIT$\Delta\sigma$: A Formal Specification of JITds Chapter 5 presents a formalisation of the core features of JITds presented in chapter 4. This chapter describes an operational semantics of JITds in the form of reduction rules, describes the type system in the form of well-formedness rules, and proves the soundness of this type system. The sound type system guarantees that a well-formed JITds program will never encounter a "method not found" or a "field not found" exception at runtime (**R2**).

Chapter 6: Designing Just-in-Time Data Structures: When to Change the Representation Chapter 6 is the second of the two chapters that discuss the *language design* spectrum of this work. Chapter 6 introduces the language constructs in JITds that enable developers to separate the logic responsible for representation changes from the general application logic, i. e., *"When to Change the Representation"*, **R2**. Again, these new constructs are placed in relation to established areas of research, e. g., *aspect-oriented programming* and *context-oriented programming*.

Chapter 7: JITds: The Language Implementation Chapter 7 briefly presents the *language implementation* aspects of this work. While language implementation is not the focus of this dissertation, we want to show that it is possible to implement JITds. To this end, we present two proof-of-concept implementations of JITds. A first implementation compiles JITds code into plain Java augmented with aspects, and a second implementation compiles JITds code into C code augmented with a custom-built runtime engine.

Chapter 8: Evaluation Chapter 8 presents five small programs written in JITds. The programs were designed to illustrate the different scenarios of applicability (see chapter 3) and to use the main features of JITds (see chapters 4 and 6). Furthermore, these

JITds programs are compared to plain Java programs in terms of efficiency and code structure, in order to evaluate the applicability of JITds.

Chapter 9: Related Work In chapter 9 we present a survey of the research with the most influence on JITds. We discuss this related work both in function of the *seven dimensions of representation changes* (from chapter 3) as well as comparing it to JITds (from chapters 4 and 6).

Chapter 10: Conclusions and Future Work Finally, chapter 10 recapitulates what just-in-time data structures are about, presents our conclusions and formulates some future topics of investigation, especially in the direction of an improved language implementation for JITds.

Chapter 2

Motivation

2.1 On the Complexity of Performance

Today, writing software entails adding yet another layer of abstraction on top of an ever growing stack of technology. Let us consider a Clojure program as an example: such a program is compiled to Java byte-code, the Java byte-code is interpreted by a virtual machine (JVM), the JVM has an implementation for a specific operating system (OS), each OS is run on actual hardware, ... Moreover, each layer of abstraction adds an extra level of cognitive burden for the developer using these contemporary setups. For instance, an optimising compiler reorders instructions and restructures loops. The JVM takes tasks such as memory management out of the hand of the developer and replaces "hot code" by a highly optimised machine code version (just-in-time compilation). The OS is responsible for managing physical and virtual memory. And finally the hardware has become more complex than ever: multi-level caches, various coherence protocols, out-of-order execution, pipelining, vector-instructions, multicore, ... This stack of technology is sometimes referred to as the "ladder of abstraction" [86]. It is clear that this metaphorical ladder is continuously growing. The expertise of the average programmer, however, has not necessarily evolved along with it.

In 1975, for instance, it was still feasible to understand the expected performance of a MOS 6502 chip (e. g., in the Commodore 64) from the smallest transistor to the most high level instruction. Today, however, the average programmer can experience difficulties in mastering even a single framework within a single language [22]. This lack of knowledge could be attributed to the lack of proper training. According to Stackoverflow's 2015 developer survey [63], 48% of software developers never received a degree in computer science.

Holding a degree in computer science, however, is not a guarantee for understanding performance. Suppose, for the sake of the argument, that performance is the result of the combination of the *complexity of algorithms* and the concrete *hardware architecture* they are executed on. Then, today's computer science curricula put forth two "bodies of knowledge" that address performance [2]: Algorithms and Complexity (AL) and Architecture and Organisation (AR).

The principles taught in courses on "Algorithms and Complexity" have been developed decades ago [14], and are encouraged to be taught *independent* of computer hardware [2]. In this section we show by example that the models studied in these courses are losing predictive power. More accurate models, on the other hand, are more complex, highly specialised, and not often discussed in a computer science curriculum. The "Idealised cache model", for instance, is specialised towards analysing algorithms in the context of cache behaviour [33]. In such an analysis, cache line size, data access patterns, and memory latency need to be taken into account.

Courses on "Architecture and Organisation", on the other hand, focus on digital logic, machine representation, machine languages (assembly), and memory organisation [2]. Specialised courses on performance exist, but are suggested as optional (elective), and only intended to make students aware of the *existence* of hardware features such as branch prediction (familiarity [1]).

We claim that understanding, predicting, and improving the performance of computer programs is no longer within the expertise of the average software developer. In this section we support this claim by presenting two examples. These examples are simple programs whose unexpected behaviour can only be explained when the knowledge of certain low-level hardware details exceeds the level of "familiarity". We opt to use "academic" examples in this section for two reasons. First, even small academic programs can exhibit unexpected performance characteristics. This strengthens our claim that performance engineering is no longer within the expertise of the average software developer, because it is even hard in the smallest example programs. Second, using academic examples, instead of a more complex example, allows us to streamline that argumentation. In appendix A we report in detail on a non-academic example: a multi-month, multi-person endeavour of understanding and explaining the performance of a contemporary piece of hardware. The combination of these three examples allows us to conclude that on contemporary hardware, the ladder of abstraction has become so high that it is no longer feasible for an average programmer to fully understand the behaviour, and thus the performance characteristics, of computer programs.

2.1.1 Calculating or Estimating Time and Space Complexity

In sections 2.1.2 and 2.1.3 we discuss two academic examples and analyse their performance. First, we present the algorithm analysis techniques that we used to analyse these examples. These techniques are well-know and are commonly part of any computer science curriculum. According to ACM/IEEE-CS Joint Task Force on Computing Curricula [2] "Computing and estimating a program's performance characteristics" is fundamental to computer science and software engineering.

While the program written for the theoretic Turing machine has access to unlimited time and memory, a practical program has only access to limited resources. In many cases writing software is making the standard tradeoff between using *space* and *time*. Both can be considered equally important, depending on the context. In the following

[1] "Outcomes listed at the Familiarity level will typically require less coverage than topics at the Usage level, which in turn require less coverage time than Assessment outcomes." – quoted from [2]

chapters "performance" denotes both space and time complexity. In this section, however, we focus on time complexity only, as it suffices to make our case.

Typically, the analysis of a program is based upon the concept of "basic operations" which represent (conceptually) *indivisible units* of work. The following steps describe how such an analysis is to be performed:[2]

1. Implement the program.
2. Determine the various basic operations.
3. Determine the cost of each basic operation.
4. Determine the frequency of each basic operation.
5. Calculate the total cost by first multiplying the frequency of each operation by its determined cost, then adding all the products.

The predictive power of the analysis depends on how precise each of the steps is performed. Since algorithms and hardware have become more complex, simplifying the analyses of some of the steps is necessary to ensure feasibility. Below we give an example of such simplifications for each of the steps.

Determine the various basic operations. Scientific computations make heavy use of floating-point calculations. For such algorithms, often only the number of floating-point operations (FLOPs) are counted. Since the number of FLOPs dominate the performance of scientific computations, counting only the FLOPs simplifies the analysis while retaining the accuracy of the analysis. A second example is the class of sorting algorithms, where commonly only the "number of comparisons" and the "'number of moves"' are considered.

Determine the cost of each basic operation. The lowest level of basic operation is "the instruction" and its cost is expressed in cycles per instruction (CPI). The CPI varies from instruction to instruction, e. g., a division is often more expensive than a multiplication, while the multiplication is often more expensive than an addition. Moreover, these numbers vary from processor to processor. In order to reduce analysis complexity, instructions can be grouped into classes with approximately the same CPI cost, where each class is assigned a single "unit cost" which allows for easy computation. For example the *fast* instructions (e. g., add) can be assigned a cost of 1, the *medium* instructions can be assigned cost of 5 (e. g., multiply) and the *slow* instructions (e. g., division) can be assigned a cost of 10. In this example we thus differentiate between three *classes* of instructions (i. e., fast, medium, and slow) which are assigned costs that allow for simple calculation (i. e., 1, 5, an 10 respectively).

[2] Based on *An Introduction to the Analysis of Algorithms* (`http://aofa.cs.princeton.edu`)

Determine the frequency of each basic operation. The frequency of each basic operation can be dependent on various (known) variables. To determine the frequency, one often has to solve recurrence equations, e. g., for a merge sort the number of comparisons is given by eq. (2.1).

$$C(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 2C(\frac{n}{2}) + n & \text{if } n > 1 \end{cases} \tag{2.1}$$

In practice, the frequency of a basic operation is often dependent on the input size. However, other variables influence the frequency too. The recurrence equation eq. (2.1) for instance gives the largest possible number of comparisons. For an already sorted sequence, however, the number of comparisons would be $2C(\frac{n}{2}) + \frac{n}{2}$. In this case, "being sorted" is a special characteristic of the input. To simplify the analysis, "special characteristics" of the input are often ignored. Instead, the analysis of the cost of an algorithm is split up into worst-case cost, average-case cost, and best-case cost.

Calculate the total cost. In larger and more complex algorithms the exact cost function becomes difficult to reason about. Therefore, if only the limiting behaviour of the cost function is of interest, one can resort to a variety of asymptotic bounds on the growth rate of the cost function. The most commonly used bound is probably the upper bound of the growth, also known as Big-Oh (O). Besides O, used bounds are o, Ω, ω, and Θ [14].

Experimental Verification. Above we briefly surveyed the five conventional steps in algorithm analysis. Further, we showed for each step how analysis is often simplified for convenience. An important feature of any analysis is of course that the expected result is also observed in practice. This feature can be verified experimentally by comparing the performance as expected by the analysis with the measured performance.

Experimental Counter Example. We observe that the suggested analysis is only accurate in the context of a *uniform cost model*. In this model it is implicitly assumed that the same basic operation always has the same cost. First, this means that memory accesses are assumed to consume constant time. And second, this means that instructions (i. e., the most basic computational steps) are assumed to consume constant time. Note that the uniform cost model does not imply the same cost for all instructions, but rather that the same instruction always has the same cost. By providing a counter example we can show that the uniform cost loses its predictive power.

Explanation. In the following sections we analyse two simple programs — using the five steps surveyed above — and compare the expected performance with actual performance as it can be measured on contemporary hardware. We measure the performance of both programs, once with an input that results in the expected performance, and once with an input where expected and measured performance are different. Finally, we *explain* the underlaying reason for the discrepancy between expected and measured performance.

2.1.2 Non-Uniform Memory Access Cost

We present a first example of a non-uniform cost model by studying memory access costs. Using the five steps presented earlier, we analyse the performance of a function that computes the length of a list. We then compare the expected results with the measured results.

Implement the program A (singly) linked list is a data structure that consists of *nodes* containing a value (here `int`) and a *pointer* to the next node. To compute the length of an acyclic list, we traverse the list — by transitively following this $next$-pointer — and count the number of steps required to reach the end of the list. An implementation in C is given in listing 2.1.

Determine the various basic operations. The basic operations that are used in this fragment are *assignment*, *NULL-test*, *pointer dereference* (cf., `current->next`), and *addition*.

Listing 2.1: Traversing a linked list by following the $next$-pointer.

```
1  typedef struct Node {
2    int value;
3    struct Node* next;
4  } Node_t;
5
6  int length(Node_t* first){
7    int counter = 0;
8
9    Node_t* current =  first;
10   while ( current != NULL ){
11     current = current->next;
12     counter++;
13   }
14   return counter;
15 }
```

Determine the cost and frequency of each basic operation. Based on the assumption of a uniform cost model, we assign a *unit cost* of 1 to each operation. Outside the `while`-loop, the only real operation is the *assignment* to `current`. Inside the `while`-loop, we always have one *assignment*, one *pointer dereference*, and one *addition*. The condition to check the termination of the `while`-loop is a simple *NULL-test*. The body and the condition of the `while`-loop are executed exactly `n` times, where `n` is the length of the list.

Calculate the total cost. Assuming a list of `n` nodes, we can estimate the time complexity of the `length`-algorithm, as shown in listing 2.1, using the following computation:

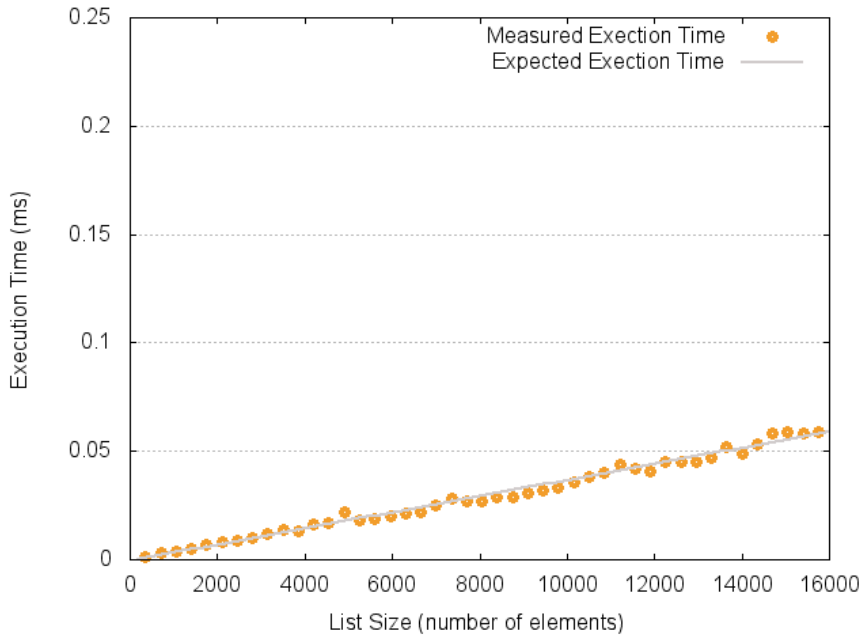


Figure 2.1: The execution time of `length` is proportional to the size of the list.

$$
\begin{aligned}
T(n) &= 1 \times \text{assignment} && \texttt{(current = first)} \\
&+ n \times \text{NULL-test} && \texttt{(current == NULL)} \\
&+ n \times \text{pointer dereference} && \texttt{(current->next)} \\
&+ n \times \text{assignment} && \texttt{(current = current->next)} \\
&+ n \times \text{increment} && \texttt{(counter++)} \\
&= 4n + 1 && \text{(unit costs of 1)} \\
&= O(n)
\end{aligned}
$$

According to this analysis, the execution time of the `length`-function only depends on the size of the list. Moreover, the expected execution time is proportional to the size of the list.

Experimental Verification. As a first experiment we measure the actual execution time of computing the length of a list. We let the size of the list vary from 0 to 16.000 elements. To reduce measurement errors in this micro-benchmark, we repeat this short running experiment 100 times and we report the average of the measured execution times.

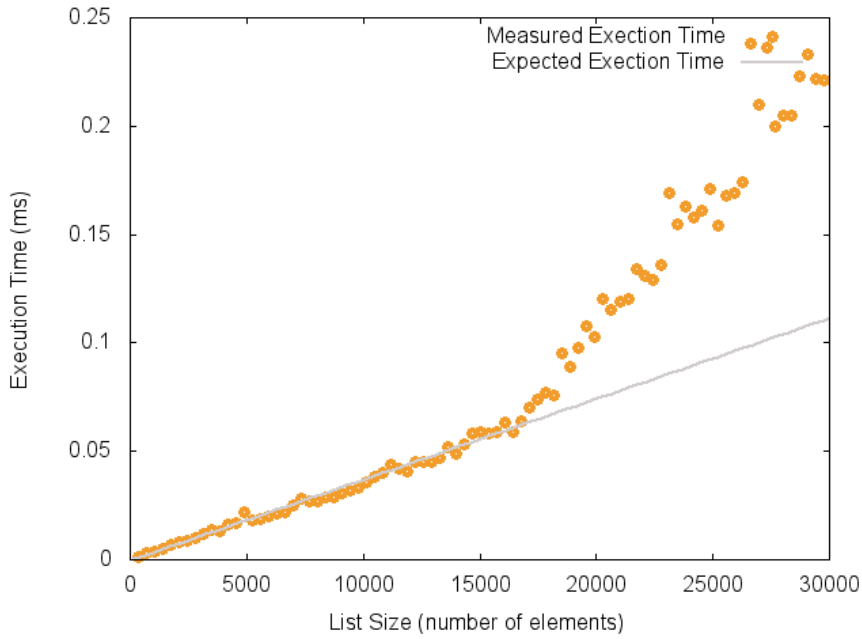


Figure 2.2: The execution time of `length` is not necessarily proportional to the size of the list.

The results of this experiment are shown in figure 2.1, where the dots represent actual measurements, whereas the continuous line is the computed fitting curve through all measured points.[3] Based on these data points it is fair to conclude that the size of the list really is the only factor that has an impact on the performance, and that the uniform cost model suffices to explain and predict the performance of the algorithm implemented by `length`.

Experimental Counter Example. However, if we repeat the experiment described above, but this time we let the the size vary from 0 up to 30.000 elements, we observe a completely different behaviour for those lists that contain more than 16k elements. Figure 2.2 shows these results: the actual measurements are represented as dots, the continuous line is the same as in figure 2.1, but continued beyond 16k.

Explanation. A linked list data structure consists of *nodes* containing a value and a *pointer* to the next node. Conceptually, node `A` and the node `B=A->next` are adjacent nodes. Physically, however, nodes `A` and `B` can reside anywhere in memory. The physical location depends on the allocator. Alternatively, a mismatch between conceptual and

[3] We used the nonlinear least-squares (NLLS) Marquardt-Levenberg algorithm from GnuPlot to compute this fit.

physical locations of nodes can occur by insertion and deletion of nodes, for instance by sorting.

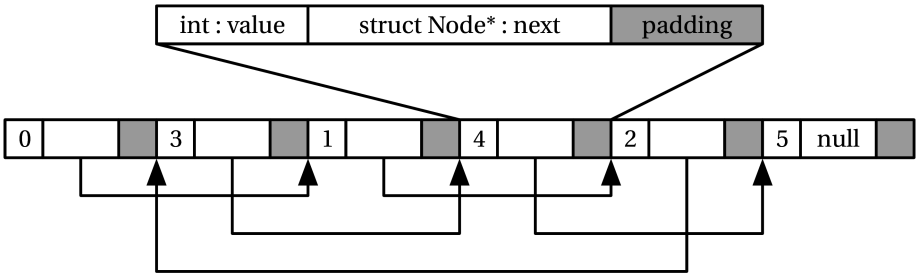


Figure 2.3: Adjacent nodes of a linked list stored in a round robin fashion with a stride of 2.

In this experiment we allocated a chunk of memory large enough to contain N nodes (i. e., array `Node_t[N]`). Instead of placing consecutive nodes in consecutive memory, we used a round robin stride of `s`. By construction we ensure that `s` divides `N`. Then, the function f, as defined in eq. (2.2) gives the relation from physical location (index in array) to conceptual location (index in list). Informally, this implies that the each node is placed s locations further than its predecessor (except those that would fall outside the reserved memory). The nodes in figure 2.3 are stored in a round robin fashion with a stride of 2.

$$f(x) = \frac{N \times (x \bmod s)}{s} + \left\lfloor \frac{x}{s} \right\rfloor \tag{2.2}$$

Each node consumes exactly 16 bytes in memory, i. e., four bytes for the `int` value, eight bytes for the pointer to the next node, and four bytes of padding (see figure 2.3). If we then take into account that the device this experiment was run on has a L2 data cache of 256KiB, then we can compute the number of nodes that simultaneously fit in the L2 cache: $\frac{256KiB}{16B} = 16,384$. The striding causes the nodes to be scattered through the allocated memory, which in turn results in low spatial locality when traversing the linked list. As long as all nodes fit in the (L2) cache, execution times increase linearly with the input size. Conversely, when the data set does not fit in the cache, the low spatial locality incurs a high performance penalty which in turn destroys the linear performance. This is clearly visible in figure 2.2.

While the aforementioned analysis is theoretically correct, experimentations reveals that the execution time is not necessarily proportional to the input size. This experiment allows us to conclude that the uniform memory access model is not a suitable model to analyse memory bound computations. The alternative, a *non-uniform memory access model* (NUMA), however, is much more complex to reason about [19].

Listing 2.2: Counting the number of elements below some threshold `t`.

```
1  int count(int t) {
2    int counter = 0;
3    for (int i = 0; i < N; i++) {
4      if ( numbers[i]<t ) {
5        counter++;
6      }
7    }
8    return counter;
9  }
```

2.1.3 Non-Uniform Instruction Execution Cost

We now present a second example of a non-uniform cost model by zooming in on the cycles consumed by each instruction (CPI). Using the five steps presented earlier, we analyse the performance of a function that computes the cumulative frequency of an element in a given distribution. We then compare the expected results with the measured results.

Implement the program In statistics, the *cumulative frequency* [87] is the sum of all absolute frequencies of class intervals below a given upper bound.

The program `count` computes the cumulative frequency of integers below a certain threshold `t`. Listing 2.2 shows an implementation of the `count` function, where `numbers` is a globally allocated array of constant size `N` containing the numbers. The `count` function simply iterates over the array and increments a counter every time the *smaller than*-test (`numbers[i]<t`) succeeds.

Determine the various basic operations. In contrast to the previous example, `count` traverses the memory in a regular fashion. Therefore we can expect the execution time of *reading* an element to be constant on average. Additionally, the operations introduced in the header of the `for`-statement induce a constant overhead as well. Hence, we can ignore both, reading and the for-loop header, in our estimate. Consequently, the only basic operations of interest for analysing the performance of this program are those in the body of the `for`-loop: a *smaller than*-test and an *increment*.

Determine the cost and frequency of each basic operation. Based on the assumption of a uniform cost model, we assign a *unit cost* of 1 to both the *less-than test* and the *increment* operation. The body of the `for`-loop is executed exactly `N` times, i. e., the constant number of elements in the array `numbers`. Thus, the condition of the `if`-statement is executed `N` times as well. The body of the `if`-statement, however, is only executed when the *less than* condition holds. Because we defined the distribution of the elements of `numbers` we can, for each value of `t`, determine exactly how many times the condition holds.

To simplify the analysis of the performance characteristics we ensure that the integers in the array `numbers` are uniformly distributed. Assume `N` to be the fixed length of the array `numbers` and `U` the upper bound on the numbers appearing in the collection. By construction we ensure that `U` divides `N`. Concretely, we construct the array `numbers` such that the frequency of each number appearing in the array `numbers` is $\frac{N}{U}$. The cumulative frequency $F(x)$ is then given by eq. (2.3).

$$\forall x \in [0, U] : F(x) = x \times \frac{N}{U} \tag{2.3}$$

If `count` is called with parameter `t` and `t` $\in [0, U]$, then $F(t)$ gives the number of times the condition `numbers[i]<t` holds, and thus the number of times the increment operation on line 5 is performed.

Calculate the total cost. If we want to estimate the time complexity of the $count$-algorithm as shown in listing 2.2 we get the following computation:

$$\begin{aligned} T(count(t)) &= N \times \text{ less-than test} && \texttt{(numbers[i]<t)} \\ &\quad + F(t) \times \text{ increment} && \texttt{(counter++)} \\ &= O(N + t) && N \text{ is kept constant} \\ &= O(t) \end{aligned}$$

According to this analysis, the execution time of the `count`-function is dominated by the value of the parameter t. Stated more precisely, the execution time of `count(t)` is proportional to t.

Experimental Verification As a first experiment we measure the execution time of counting numbers between 0 and 999 inclusive, uniformly distributed in an array of size 10^6 elements. Thus, each number is present exactly 10^3 times ($\frac{N}{U}$). The physical location of the numbers, however, is randomised. Figure 2.4 shows the measured execution times of computing `count` as dots. In this experiment we vary `t` from 0 to 400.

The solid line in figure 2.4 is the best fit computed from the actual measurements (see footnote 3). This shows that the measured execution times indeed match the expected execution times.

Experimental Counter Example. We now repeat the same experiment but with `t` varying from 0 to 1000, i. e., the whole spectrum of numbers that occur in the array. Again, we observe unexpected behaviour. The measured execution times of this second experiment are shown in figure 2.5. The function representing the expected execution time, as shown in figure 2.4, is plotted as a solid line. The actual measurements are plotted as dots. It is clear to see that for values of `t` larger than ± 500 the measured and expected execution times start to diverge significantly.

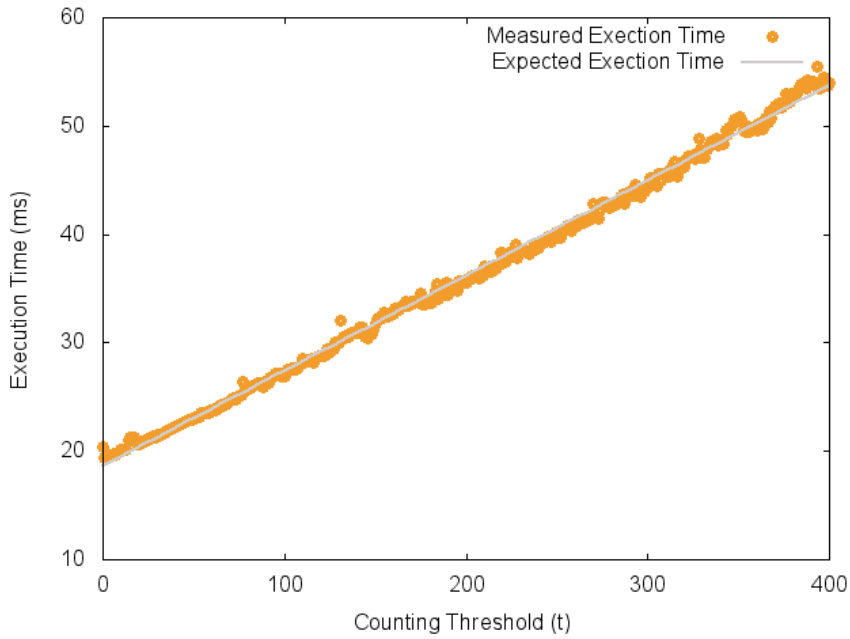


Figure 2.4: The execution time of `count` is proportional to the value of the counting threshold `t`, for all `t` up to 400.

Explanation. The origin of the discrepancy in expected and measured execution time is not apparent from the conceptual understanding of this program: increasing the amount of work should increase the execution time. What is not taken into account in the analysis presented above is the important role optimisations *within* contemporary CPUs play in the execution time of a program.

Pipelining is a technique that increases the instruction level parallelism (ILP) and as a result improves performance. Pipelining increases the instruction level parallelism by starting to execute subsequent instructions before the previous instructions are finished. Correctness is maintained by ensuring that only instructions that do not (directly) *depend* on unfinished instructions can be started. A `mov` instruction, for example, should not be executed before the source value is present in the source register. A second example of instruction dependency that hampers instruction pipelining are conditional branches: it is not possible to know which branch needs to be taken before the instructions computing the condition have produced their results. Consequently, it is not possible to pipeline "the next instruction" in the pipeline, because the next instruction is simply unknown.

Branch prediction, however, is a CPU optimisation that tries to *guess* the next instruction when it encounters a *conditional jump*, and which already feeds the guessed instruction to the pipeline. If the guess was wrong, the instruction is canceled and a misspredict penalty cost is paid (pipeline stall). If the guess was correct, however,

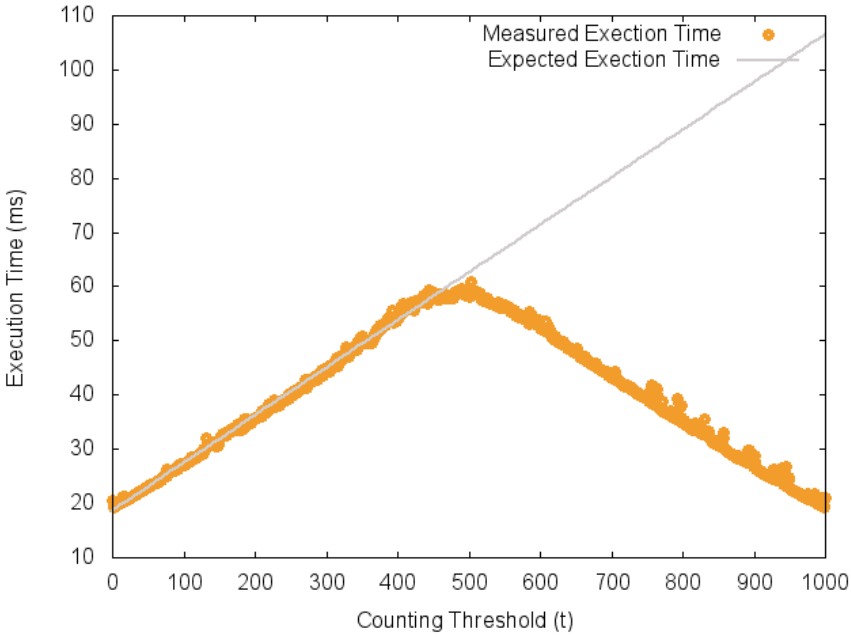


Figure 2.5: The execution time of `count` is not necessarily proportional to the value of the counting threshold `t`.

the pipeline bubble that would otherwise be introduced by waiting for the conditional jump to finish, is avoided. In general, avoiding pipeline bubbles results in better performance. Branch prediction makes its guesses based on which branches where previously taken. The two-bit Smith algorithm , for instance, encodes the history in a state machine with four states (2 bits): strongly taken, weakly taken, weakly not taken, and strongly not taken [38].

To understand what is going on in figure 2.5, consider lines 11-15 in listing 2.3, which shows the machines instructions generated from the `for`-loop of `count`. These lines compare the value `numbers[i]` with `t` (line 11), and jump to the label `AFTER_IF` if the value `numbers[i]` greater than or equal to `t` (`jge`, line 12). Lines 13–15 simply increment the variable `counter` if the *branch* was taken.

In our example, the number of times the same branch is taken as a function of `t` is given by $f(t) = \frac{N}{U} max(U-t,t)$. Thus for $t = 0$ and $t = U$ the same branch is always taken, while either branch becomes equally likely of being taken when `t` approaches $\frac{U}{2}$. Since branch prediction makes its guesses based on branches taken in the past, the branch miss prediction rate is highest around $t = 500$. When $t \approx 500$ the probability of `numbers[i]<t` is around 50%. This means that the either branch is as likely to be taken as the alternative branch. Moreover, there is no correlation with the history of branches taken previously . Consequently, any guess is as good as the other: i. e., 50% success rate. Once `t` exceeds 500, the number of right branch predictions exceeds 50%. As a result,

Listing 2.3: Assembler code of the `for`-loop from listing 2.2.

```
1  ...
2  mov     %edi,-0x4(%rbp)       # init variable t (parameter)
3  movl    $0x0,-0x8(%rbp)       # init variable counter
4  movl    $0x0,-0xc(%rbp)       # inti variable i
5  #FOR_LOOP_HEADER:
6  cmpl    $0x989680,-0xc(%rbp)  # compare 10000000 and i
7  jge     FOR_LOOP_END          # conditinal escape loop
8  lea     0x227(%rip),%rax      # read numbers
9  movslq  -0xc(%rbp),%rcx       # read i
10 mov     (%rax,%rcx,4),%edx    # read numbers[i]
11 cmp     -0x4(%rbp),%edx       # compare t and numbers[i]
12 jge     AFTER_IF
13 mov     -0x8(%rbp),%eax       # read counter
14 add     $0x1,%eax             # increment counter
15 mov     %eax,-0x8(%rbp)       # store counter
16 #AFTER_IF:
17 mov     -0xc(%rbp),%eax       # read i
18 add     $0x1,%eax             # increment i
19 mov     %eax,-0xc(%rbp)       # store i
20 jmpq    FOR_LOOP_HEADER       # jump to loop header
21 #FOR_LOOP_END:
22 ...
```

the actual cost of executing the `if` statements's consequent is hidden by the processor and the measured performance is better then theoretically expected.

2.1.4 Non-Uniform Cost Model

In section 2.1.1 we revisited the well known steps for for analysing the performance of algorithms. These same steps were used to analyse two example programs in sections 2.1.2 and 2.1.3, respectively. We present the observed measurements for these two example programs as "surprising results". Of course, both examples are carefully crafted to expose these "anomalous behaviours". Note that crafting these examples to evoke such behaviour as clearly as possible is not at all trivial. Moreover, the inverse exercise, explaining such unanticipated behaviour, is arguably even more difficult. Appendix A reports in detail on such an exercise. Here, we present a summary of the search for explaining anomalous performance in a program that computes *triangular numbers*.

The $\mathtt{N}^{th}$ triangular number is the sum of the first `N` integers (i. e., $\sum_{i=0}^{N} i$). The $\mathtt{N}^{th}$ tetrahedral number is the sum of the first `N` triangular numbers (i. e., $\sum_{i=0}^{N}\sum_{j=0}^{i} j$). Listing 2.4 shows a straightforward implementation in C to compute the $\mathtt{N}^{th}$ tetrahedral number. Listing 2.5 computes the double of the $\mathtt{N}^{th}$ tetrahedral number by running the inner-loop on lines 3–7 of listing 2.4 twice.

The second program (listing 2.5) performs twice the work of the first program (listing 2.4). Thus running the second program is expected to take twice as long as running the first program. When we measured the execution time of both programs, we observed

Listing (2.4) Nested loops computing the N^{th} tetrahedral number.

```
1  long total = 0;
2  for (long i=0; i<N; ++i ) {
3    long sum0 = 0;
4    for (long j0=0; j0<i; ++j0 ) {
5      sum0 += j0;
6    }
7    total += sum0;
8
9
10
11
12
13
14 }
```

Listing (2.5) Nested loops computing the N^{th} tetrahedral number twice.

```
1  long total = 0;
2  for (long i=0; i<N; ++i ) {
3    long sum0 = 0;
4    for (long j0=0; j0<i; ++j0 ) {
5      sum0 += j0;
6    }
7    total += sum0;
8
9    long sum1 = 0;
10   for (long j1=0; j1<i; ++j1 ) {
11     sum1 += j1;
12   }
13   total += sum1;
14 }
```

that the second program was 1.8 faster than expected. Ruling out "programmer error", "inaccurate measurements", "unanticipated compiler optimisation", "compiler error", and "known hardware related performance pitfalls" as the origin of the mismatch between expected and measured performance proved to be a multi-month, multi-person endeavour. Finally, we were able to attribute the 12% mismatch in execution time to an undocumented performance anomaly present in a specific type of hardware, which we named the read-after-write (RAW) hiccup [23]. This *RAW hiccup* occurs when a read instruction is issued "too soon", within 2 cycles, after a read instruction that targets the same cache line. When this happens, the read instruction behaves as if a cache miss occurs, resulting in a performance penalty. Because, `sum0` and `j0` accidentally share a cache line, and because the update of `sum0` (line 5 in both programs) happens just before `j0` is read (`++j0`, line 4 in both programs) this first inner loop suffers from a RAW hiccup in *every* iteration. The second loop in the second program performs significantly better because `sum1` and `j1` do not share a cache line, and thus no performance degradation occurs. In conclusion, because the program in listing 2.5 performs better than initially expected because one of the loops performs better. A more detailed discussion on this example can be found in appendix A.

The presented examples (sections 2.1.2 and 2.1.3 and appendix A) all support the claim that "performance engineering" in general is hard. All three examples take the low level approach, i. e., they study the effects of *hardware* on the execution time of programs. Alternatively, we could have presented a set of examples that exhibit anomalous behaviour as a result of the *software stack* used. Such unexpected performance measurements can, for instance, be caused by *garbage collection pauses* or *VM warm-up time* [35], to name only two.

The Ladder of Abstraction causes Vertigo.

In this section we present anecdotical evidence for the fact that performance engineering is hard. As shown in the examples, not all performance issues are resolved by the compilers, and thus performance engineering remains generally the job of the software developer. Combining the two tasks of performance engineering and program design at the same time, however, has become difficult and complex.
We believe that to manage the aforementioned complexity, **application logic should be enhanced with additional logic that offers explicit technical support in order to deal with performance concerns**. While these two *flavours* of logic should be causally connected, it should be possible to develop them lexically independent from each other. Such a lexical separation is beneficial as it allows for each domain expert to operate on their own terrain: 1. the application developer can focus on the application logic, whereas 2. the performance engineer can focus on the performance critical parts of the application's code base.

2.2 On the Effect of Data Representation on Performance

Contemporary software engineering and computer science curricula train students to be able to determine which data structure is *the best fit* for a given problem [2]. In a classical curriculum, students fairly early learn that interface and implementation are not necessarily tightly coupled, and that for each abstract data type (ADT), a set of possible implementations might exist [1]. Further, such a curriculum introduces complexity analysis (cf. section 2.1) to prepare students to make informed decisions on how to find the "best representation" to use in combination with a given algorithm [2]. However, that ideal representation does not necessarily exist.

First, in sections 2.2.1 to 2.2.3, we launch the idea that, for the sake of performance, using a set of data representations can be better than relying on a single representation only. First, we present an example program that manipulates matrices. When studied carefully, we observe that the way the matrices are being used changes during the execution of the program. Moreover, we show that, for various example programs, using multiple data representations for the matrix data interface leads to better performance than can be achieved by relying on a single representation only. We thus argue in favour of "online data representation changes" to improve performance.

Then, in section 2.2.5, we show by example that in most programming languages implementing sound and scalable representation changes is currently not practicable. We subsequently argue that languages that are expressive enough to support dynamic representation changes do not facilitate the separation of application logic from performance logic as suggested in section 2.1.

Combining the insights from this section (section 2.2) and section 2.1, we argue in favour of a new strand of programming language features to express data representation changes (section 2.3). In chapters 4 and 6 we introduce JITds, a new programming lan-

guage with these features. Programming languages are supposed to change the way developers think about programming [66]. Therefore, JITds facilitates the shift from choosing a single data structure to choosing a set of data structures.

2.2.1 The Matrix and its Representations

A *matrix* is a well known and commonly used abstraction in science. Matrices are particularly useful in linear algebra, e. g., every matrix corresponds to a unique linear transformation. In natural sciences, such as biology and physics, and in engineering sciences, matrices — and their operations — are often used to model real life phenomena. Consequently, matrices are also commonly used in computer programs, e. g., scientific applications.

In this section we present the data interface `Matrix` and two possible data representations. In the remainder of this text the matrix is used as recurring example, therefore we discuss both the data interface and the data representations in considerable detail.

The Matrix Data Interface. We define the data interface of a `Matrix` as a constructor that creates a `rows` × `cols` matrix; 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. For completeness and for use later in this text, we also added accessors to retrieve the number of rows and columns of a matrix, these are `getRows` and `getCols`, respectively. Moreover, we added a copy constructor which creates a deep copy of the argument `Matrix`. Listing 2.6 implements this data interface as a Java abstract class definition, specialised for `double` values.

We explicitly choose to present the `Matrix` *data interface* as an *Java abstract class* instead of using the *interface* construct. First, this allows us to define what the constructor(s) need to look like in terms of types, which is not possible when using Java interfaces. Second, it exemplifies and supports the claim — made earlier in this text, on page 4 — that Java interfaces and data interfaces are not the same concept.

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, *row-major order*, stores elements of the same row in consecutive memory locations. The second data representation, *column-major order*, stores elements of the same column in consecutive memory locations. To visualise the difference, consider the matrix M. The elements of M are letters.[4] When laid out in memory, the letters of matrix M read as “son pie alt” when stored in row-major order. When stored in column-major order we would read the “spa oil net” in the memory occupied by M.

$$M = \begin{pmatrix} s & o & n \\ p & i & e \\ a & l & t \end{pmatrix}$$

[4] We opt to use letters here because they do not have a natural ordering. The natural ordering of numbers, for instance, would make one ordering unintendedly look more appealing, i. e., 1, 2, 3, ..., 9.

Listing 2.6: An abstract class Matrix providing a data interface.

```
1  public abstract class Matrix {
2    int rows, cols;
3
4    public Matrix(int rows, int cols) {
5      this.rows = rows;
6      this.cols = cols;
7    }
8
9    public Matrix(Matrix src) {
10     this(src.getRows(), src.getCols());
11     for (int r=0 ; r<src.getRows() ; r++)
12       for (int c=0 ; c<src.getCols() ; c++)
13         this.set(r,c,src.get(r,c));
14   }
15
16   int getRows() { return rows; }
17   int getCols() { return cols; }
18
19   double abstract get(int row, int col);
20   void   abstract set(int row, int col, double val);
21 }
```

Listing 2.9: The struct `Matrix` combines data values and an access function.

```
1  struct Matrix {
2    int (*coord2idx)(int row, int col);
3    double           data[N*N];
4  };
5
6  double get(Matrix* m, int row, int col){
7    return m->data[ m->coord2idx(row,col) ];
8  }
9
10 void set(Matrix* m, int row, int col, double val){
11   m->data[ m->coord2idx(row,col) ] = val;
12 }
```

Listings 2.7 and 2.8 show concrete implementations in Java of these two suggested representations for `Matrix`, using the row-major and the col-major layout respectively. Both representations store the matrix values in a one dimensional `double` array of exactly the size of the matrix (i. e., $rows \times cols$, line 7). The two representations only differ in how the two-dimensional coordinate `row,col` is translated into a one-dimensional coordinate in the `double` array (see lines 15 and 19).

This is even more clear in the C implementation which we use for experiments later in this chapter. This alternative implementation is shown in listings 2.9 and 2.10. Here, a matrix is implemented as a struct combining an array `data` and a function pointer `f`. `data` contains the actual values in consecutive memory and `f` is responsible for con-

Listing (2.7) Row-major data representation.

```
1  public class RowMajorMatrix extends Matrix {
2
3    double[] data;
4
5    public RowMajorMatrix(int rows, int cols) {
6      super(rows,cols);
7      this.data = new double[rows*cols];
8    }
9
10   public RowMajorMatrix(Matrix src) {
11     super(src);
12   }
13
14   double get(int row, int col) {
15     return data[ row*getCols()+col ];
16   }
17
18   void set(int row, int col, double val) {
19     data[ row*getCols()+col ] = val;
20   }
21 }
```

Listing (2.8) Col-major data representation.

```
public class ColMajorMatrix extends Matrix {

  double[] data;

  public ColMajorMatrix(int rows, int cols) {
    super(rows,cols);
    this.data = new double[rows*cols];
  }

  public ColMajorMatrix(Matrix src) {
    super(src);
  }

  double get(int row, int col) {
    return data[ row+col*getRows() ];
  }

  void set(int row, int col, double val) {
    data[ row+col*getRows() ] = val;
  }
}
```

Listing 2.10: Functions for Row and Column Major Access.

```
1  int rowMajorOrder(int row, int col) {
2    return (row*N)+col;
3  }
4
5  int colMajorOrder(int row, int col) {
6    return row+(col*N);
7  }
```

verting a `(row,col)` coordinate into an index in `data` (listing 2.9). Listing 2.10 shows two such functions, one for row-major access and one for column major access, respectively. The representation of a `Matrix` is thus defined by the value of the field `f`.

A third Matrix Representation. Both representations above require at least one memory location per element. In scientific or engineering applications matrices are often sparse, i. e., most of its elements are zero. For such matrices, it can be beneficial to adhere to a storage scheme where only the non-zero elements are stored. Besides the reduction in space complexity (denoted as $S(x)$), also a reduction in time complexity (denoted as $T(x)$) can be achieved, e. g., by using dedicated algorithms tailored towards sparse matrices. Changing between specialised algorithms, e. g., as studied by Ansel et al. [6], falls outside the scope of this dissertation.

Row compressed format, column compressed format, hash-map (coordinate to value) based, and tree based implementations are only a few of the possible *sparse matrix* representation implementation techniques. The advantage of a sparse matrix representation over a *dense matrix* representation is the reduction in memory consumption. A sparse representation occupies memory proportional to the number of non-zero values, independent of the size of a matrix. Conversely, a dense matrix representation occupies memory proportional to the size of a matrix. For the identity matrix I_n this is a difference between $S(I_n) = O(n)$ and $S(I_n) = O(n^2)$ respectively.

In practice `SparseMatrix` implementations introduce some overhead, both in terms of space and time, to encode the location of the elements which is given "for free" in an array. It is thus important to note that in practice using `SparseMatrix` only pays off when the matrix is actually sparse. Thus, the data representation does not necessarily reveal the characteristics of the data. Note that the "definition" of `SparseMatrix` above ignores special kinds of sparse matrices. An implementation of a diagonal matrix, for instance, does store zeroes sometimes (i. e., when the first diagonal contains a zero), and consumes memory proportional to the size of the matrix's diagonal instead of to the number of non-zero elements (e. g., an array of N values, where N is the number of elements on the matrix's diagonal).

For the remainder of this text, we use `SparseMatrix` to mean a data representation of `Matrix` that consumes memory proportional to the number of non-zero values in the matrix. Concrete implementations of `SparseMatrix` are considerably more complex than listings 2.7 and 2.8 and are therefore omitted here. Appendix C, however,

Listing 2.11: Classic Matrix-Matrix Multiplication Algorithm of two $N \times N$ matrices

```
1  Matrix mul(Matrix mA, Matrix mB) {
2    Matrix mC = 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          mC.set( i, j, mC.get(i, j) +
8                        mA.get(i, k) * mB.get(k, j));
9    return mc;
10 }
```

does provide a concrete implementation for `SparseMatrix`.

2.2.2 Effect of Representation on Performance

Let us now investigate the effect of the chosen representation on performance. To ease the explanation, we ignore the `SparseMatrix` representation for a while, and we focus on both dense representations, i. e., `RowMajorMatrix` and `ColMajorMatrix`.

The Classic Matrix-Matrix Multiplication Algorithm. One of the most commonly used operations on matrices is the matrix-matrix multiplication, or in this context, multiplication for short. The classic multiplication algorithm is a straightforward implementation of the formula $C_{ij} = \sum_k^N A_{ik} \times B_{kj}$. Each of the elements of matrix `mC`, is the dot-product of a *row* of `mA` with a *column* of `mB`.

To simplify the discussion, we only consider *square* matrices of size $N \times N$. Listing 2.11 shows a Java implementation of this algorithm: a method which takes two matrices `mA` and `mB` as input parameters and returns a third matrix `mC`. Lines 6–8, the innermost loop, computes the dot-product.

Effect of Representation on Data Access Pattern. If we consider the conceptual access pattern, i. e., the access pattern at the abstraction level of the data interface, we observe that the matrix `mA` is continuously accessed row by row, while the matrix `mB` is continuously accessed column by column. How this relates to the physical memory access pattern depends on the representations of both `mA` and `mB`.

We say that the conceptual access pattern matches the representation when a `RowMajorMatrix` is accessed per row or when a `ColMajorMatrix` is accessed per column. If the conceptual access pattern matches the representation, then the physical access pattern becomes more linear, which translates to better spatial locality. However, when the conceptual access pattern conflicts with the used representation, the physical memory access pattern is non-linear, which translates to low spatial locality (cf. section 2.1.2). Note that when relying on a single data representation only, it is impossible to have matching access patterns in both `mA` and `mB`.

Effect of Representation on Performance. A simple experiment shows that **the choice of data representation of the matrices `mA` and `mB` has a significant effect on the execution time of multiplying two matrices.** In an experiment, we executed the aforementioned `mul` function with all combinations of data representations for both input matrices `mA` and `mB`, while we kept the data representation of the matrix `mC` fixed in the row-major order representation. The measured execution times in seconds for all combinations are shown in table 2.1. We gathered these numbers from the C implementation (cf. listings 2.9 and 2.10), compiled with gcc's maximal optimisation i. e., `-O3`. The resulting binary was executed on a 2.6 GHz Intel Core i7 processor with 256 KiB of L2 cache per core and 6 MiB of L3 cache shared by all cores. The size of the matrices, 1024 × 1024 elements, is chosen such that the matrices do not fit in the last level cache (LLC). Consequently, the effect of the representation is clearly observable from the experiment.

Data Representation `mA`	Data Representation `mB`	Execution Time
Row-Major Order	Column-Major Order	3.58s
Column-Major Order	Column-Major Order	7.92s
Row-Major Order	Row-Major Order	10.52s
Column-Major Order	Row-Major Order	16.31s

Table 2.1: The execution time of multiplying two 1024 × 1024 matrices depends on the chosen data representation.

When relying on the row-major representation only we measure an execution time of about 11s. The column major representation performs slightly better with an execution time of about 8s. With an execution time of only 4s, matching data access pattern and data representation, i. e., `RowMajorMatrix` × `ColMajorMatrix`, results in a significantly better performance. Conversely, when the data access pattern and data representation conflict for *both* matrices, the execution time is significantly higher as it takes 16s. This is because caches can only hide memory latency when the program exhibits high spatial or temporal locality. This is the case when the the data access pattern and data representation match. Otherwise, the program's performance is dominated by a high cache-miss rate, which results in low performance. Thus, to ensure a "fast" execution one should use a `RowMajorMatrix` as the first argument to `mul`, and a `ColMajorMatrix` as the second argument.[5]

2.2.3 Effect of Changing Representation on Performance.

We now present a second experiment which shows that it is not always possible to use a `RowMajorMatrix` as the first argument and a `ColMajorMatrix` as the second argument. For the predicate `doCommute` (see listing 2.12), for instance, it is impossible to find a combination of representations for `mA` and `mB`, such that both calls to `mul` are "fast". Because `mA` as well as `mB` are used once as the first and once as the second argu-

[5] The results of this experiment are representative for matrices that do not fit into the cache. If the matrices fit into the cache no cache misses will occur, hence no performance penalties apply.

Listing 2.12: A predicate to check if two matrices `mA` and `mB` *commute*.

```
1  boolean doCommute(Matrix mA, Matrix mB) {
2    Matrix mAB = mul(mA, mB);
3    Matrix mBA = mul(mB, mA);
4    return mAB.equals(mBA);
5  }
```

Listing 2.13: Changing the representation of a matrix from row-major to col-major order in C.

```
1  struct Matrix* m = alloc(sizeof(struct Matrix));
2  m.coord2idx = &rowMajorOrder;
3  ...
4  transpose(m);
5  m.coord2idx = &colMajorOrder;
```

ment of `mul`, thus the combination that yields the "fast" performance for the first multiplication, excludes the combination that yields the "fast" performance for the second multiplication. By extrapolating the measurements presented in table 2.1 we expect that calling `doCommute(A,B)` with two instances of `ColMajorMatrix` yields the fastest execution (see table 2.2).

Data Representation of `mA`	Data Representation of `mB`	Execution Time	
		Expected	Measured
Column-Major Order	Column-Major Order	$2 \times 7.92s = 15.84s$	$15.45s$
Row-Major Order	Column-Major Order	$3.58s + 16.31s = 19.89s$	$18.24s$
Column-Major Order	Row-Major Order	$16.31s + 3.58s = 19.89s$	$19.59s$
Row-Major Order	Row-Major Order	$2 \times 10.52s = 21.04s$	$20.86s$

Table 2.2: The execution time of `doCommute` by extrapolation of the measurements from table 2.1 (expected) and by measuring.

Now, consider the function `transpose` which reflects the square matrix `M` over its main diagonal (see listing 2.14). First note that a row-major order matrix M stores its elements in exactly the same order as a the transposed col major order matrix M^T, and vice versa. To illustrate, consider

$$M^T = \begin{pmatrix} s & o & n \\ p & i & e \\ a & l & t \end{pmatrix}^T = \begin{pmatrix} s & p & a \\ o & i & l \\ n & e & t \end{pmatrix}.$$

Note also that, in the context of the C implementation, first calling the `transpose` function (assume a C version of listing 2.14), and then replacing the value of the function pointer `coord2idx`, which points to a function that converts the row-col coordinates

Listing 2.14: The function `transpose` reflects the square matrix `mA` over its main diagonal.

```
1  void transpose(Matrix mA) {
2    assert(mA.getRows()==mA.getCols());
3    int size = mA.getRows();
4    for (int r=0 ; r<size ; r++) {
5      for (int c=r+1 ; c<size ; c++) {
6        double temp = mA.get(r,c);
7        mA.set(r,c, mA.get(c,r));
8        mA.set(c,r,temp);
9      }
10   }
11 }
```

into an index in an array , to "the other version" effectively changes the representation of a matrix (listing 2.13). Listing 2.13 shows this the transition from row-major order (line 2) to col-major order (lines 4 and 5).

We again compute the product of two square matrices. This time, however, we start with both matrices in the `RowMajorMatrix` representation and change the representation of the second matrix `mB` to column-major order just before the actual multiplication. We use the `transpose` operation to convert `mB`.

This experiment, where the representation changes during the execution, takes 3.61s to multiply two 1024×1024 matrices. First, compare this to the fastest execution from table 2.1, which takes 3.58s, but does not perform a transposition. Second, compare this to the 10.52s needed to multiply two row-major matrices, which also does not perform a transposition but starts with the same representation as this experiment. We conclude that for this example, changing the representation at runtime and performing the multiplication is significantly faster than performing the multiplication with the initial representation. Second, changing the representation at runtime and performing the multiplication is only marginally slower than performing the multiplication with the optimal combination of representations.

> Although it might seem counter intuitive, the matrix multiplication example illustrates the existence of programs where performing both a representation change and an actual computation yields better performance than performing that computation only.

Small Matrices In section 2.2.2 we experimentally determine the best combination of representations with which to call the matrix multiplication method. However, all presented numbers are gathered by measuring the execution time of multiplying two 1024×1024 matrices. For such large matrices that do not fit in the cache, the chosen representation has a significant impact on performance because of cache-effects. Moreover, in section 2.2.3 we showed that transposing the matrix first and then performing

Listing 2.15: `applyStencil` updates the values of its arguments.

```
1  void applyStencil(Matrix src, Matrix tar, int iterations) {
2    for (int i=0 ; i<iterations ; i++) {
3      for (int r=0 ; r<src.getRows() ; r++) {
4        for (int c=0 ; c<src.getCols() ; c++) {
5          tar.set(r,c, heat(src,r,c));
6        }
7      }
8      Matrix tmp = src; src = tar; tar = tmp;
9    }
10 }
```

Listing 2.16: `applyStencil` updates the values of its arguments.

```
1  double heat(Matrix mA, int r, int c) {
2    return ( mA.get(r+0,c+0) +
3             mA.get(r+1,c+0) +
4             mA.get(r+0,c+1) +
5             mA.get(r+1,c+1)   ) / 4.0 * 0.9;
6  }
```

multiplication yields a faster execution than eagerly multiplying without changing the representation.

For small matrices that do fit in the cache, however, cache effects do not influence the execution time. As a corollary, transposing a small matrix introduces overhead but does not lead to improved performance in the actual multiplication. Thus, not only the data access pattern (cf. `mul`) has an influence on which representation to use, but also characteristics of the data itself, here the *size of the matrix*.

2.2.4 Changing Data Characteristics.

We now present a second example where changing the representation of a data object at runtime turns out to be beneficial. The focus in this example will be the characteristics of the data (cf. small matrix) as motivation for a representation change. We present this example, as opposed to the matrix multiplication example which was discussed extensively, in a narrative form only. Later in this chapter, this second example serves as a case to show the limitations of contemporary programming languages in regards to expressing representation changes.

Stencil computations are a second type of commonly used computation involving matrices. Amongst other applications, stencil computations are often used for numerical approximations when solving partial differential equations. A classic implementation, as shown in listing 2.15, updates all values in an output matrix as a function of the values in the neighbourhood of the corresponding element in an input matrix. In the code examples `tar` and `src`, respectively, denote the input and output matrices. This

"neighbourhood function" is called a stencil. Iteratively applying this function on the same two matrices, once as input then as output, models the effect of time.

The example we discuss here, models the heat distribution over time, hence the `heat`-stencil, listing 2.16. Iteratively applying `applyStencil` on any data set visually reveals that the overall temperature converges to zero. This effectively means that the number of zero values has greatly increased over time. First note that "the number of zero values", i. e., sparsity, is a characteristic of the data itself and is not a property of the data representation. However, there is a relation between the two. For instance, it is credible to assume that at the beginning of the computation one of the dense representations is to be preferred, while after "some time" a `SparseMatrix` representation would be a more efficient representation. Thus, ideally the representation changes *during* the computation. In the matrix multiplication example, the representation change is motivated by the change in computation, i. e., a matrix is accessed differently when used as the left or as the right operand. Here, the representation change is motivated by a change in the data itself, i. e., if the matrix becomes sparse, a sparse representation is to be preferred.

Conclusion

In order to improve performance, in the widest sense of the term, we argue in favour of data objects whose data representation can change at runtime. Such a representation change can be favourable when the usage pattern changes (i. e., computation) or when the characteristics of the data change (i. e., data).

2.2.5 Ad-Hoc Representation Changes

Hitherto, we built up a case in favour of dynamic representation changes in order to improve performance. In the second part of this chapter we show how this can be realised in contemporary widespread programming languages. We do this by reimplementing the examples discussed above, using explicit representation changes. Below we discuss three implementation strategies to implement representation changes: by relying on a type system, by resorting to local representation changes, and by building combinations of representations. As we will discuss, each of these techniques to introduce ad-hoc representation changes exhibits problems that hamper their adoption.

2.2.5.1 Relying on a Type System

The conclusion of the matrix multiplication experiment is that the ideal combination of representations with which to call the `mul` method is `RowMajorMatrix` and `ColMajorMatrix`. In a statically-typed language it is possible to encode this conclusion in the signature of the method `mul`, e. g., `Matrix mul(RowMajorMatrix, ColMajorMatrix)`. Then, the *type system* of the language only accepts programs that use `mul` with the intended representations. Thus, the type system ensures that `mul` is always called with matrices that yield the best possible performance (cf. table 2.1). However, we identify three problems with this approach: the burden of changing the representation

lies with the client code, as a result, the logic for changing the representation is scattered and entangled with the application logic, and static type systems are to rigid to allow for "conditional representations".

Putting the Burden on the Client Code. As a result of using a conventional type system, the burden of producing a legal program lies with the code that calls `mul`. The implication is that potentially each call to `mul` has to be surrounded by code responsible for checking the "current representation" changing the current representation to the intended representation if needed. Note that in dynamically typed languages, the burden of passing the "right" arguments also lays with the client code. The difference between a dynamically typed language and a statically languages is that in the former badly typed arguments are detected at runtime only, whereas in the latter badly typed arguments are detected before the execution.

Representation Change Logic is Entangled with Application Logic and Scattered Throughout the Code Base. As already identified above, relying on the static type system implies that the client code has to be *augmented* with logic responsible for checking and changing representations. In software engineering, augmenting code with logic that has a different intent, is known as *code entanglement*. Moreover, because client code is scattered trough the code base, so is the representation change logic. Both entangled and scattered code are considered bad smells, e. g., by Kiczales et al. [49], and language features have been proposed to resolve them. These features have been studied in the field of aspect-oriented programming[49].

Loss of Flexibility. In section 2.2.2 we experimentally determined the best combination of representations with which to call the matrix multiplication method. And in section 2.2.3 we showed that, for large matrices, transposing the matrix and then performing multiplication can yield a shorter execution time than multiplying without a representation change. As a side note, however, we also mentioned that for small matrices this technique does not yield better performance but on the contrary introduces overhead only. Thus, changing the representation of a small matrix before each multiplication could even hamper performance.

The signature of `mul`, `Matrix mul(RowMajorMatrix,ColMajorMatrix)`, which statically encodes the optimal representations, does not take the size of the matrix into account. A static type system, however, is inexorable and the expected type *has* to be passed along. It is not possible to express "conditional types", especially not those that statically reason about dynamic properties such as "the size of a matrix". Also conceptually, the static types are too strict. The multiplication is intended to work with "any kind of matrix", and the static types only *suggest* the ideal representations. In short, static types put too hard a constraint on the program if one simply wants to express "intended" or "preferred" behaviour.

Listing 2.17: Change the representation of the arguments locally.

```
1  Matrix mul(Matrix mA, Matrix mB) {
2    if ( !mA instanceof RowMajorMatrix && (mA.getRows()*mA.getCols())>LARGE )
3      mA = new RowMajorMatrix(A);
4    if ( !mB instanceof ColMajorMatrix && (mB.getRows()*mB.getCols())>LARGE )
5      mB = new ColMajorMatrix(mB);
6    Matrix mC = new RowMajorMatrix(N, N);
7
8    for (int i=0 ; i<N ; i++)
9      for (int j=0 ; j<N ; j++)
10       for (int k=0 ; k<N ; k++)
11         mC.set( i, j, mC.get(i, j) +
12                       mA.get(i, k) * mB.get(k, j) );
13   return mC;
14 }
```

2.2.5.2 Local Representation Change

A second ad-hoc implementation of representation changes is to update the value of a local variable with a "transformed replica" of the original value. For the matrix multiplication example, this would look something like listing 2.17. Note that this approach does allow expressing conditional representation changes, based on dynamic properties. Lines 2 and 4 check both the "current representation" and the "size of the matrix", and react accordingly.

For the stencil computation example, we argued that a representation change could be beneficial when the matrix has become "sufficiently sparse", i. e., after a certain number of iterations. The programs in figure 2.8 achieve this in two different ways. On the left we show a program where the representation change should happen in iteration number `T`. A concrete value for `T` should be determined by a performance engineer, e. g., by trial-and-error. The program on the right is a bit more subtle as it estimates the actual *sparsity* of the matrices using the counter `zeros`. The sparsity of a matrix is a *data characteristic* upon which it can be decided to perform a representation change. Here, when the number of zeros exceeds the threshold `S`, then the sparse representation is chosen and transition to.

Local Representation Changes and Identity Imagine the heat example to be augmented with a graphical interface that continuously updates the thermographic images displayed on the screen. To this end, the thread responsible for updating the screen holds a reference to both matrices. A local representation change, as the name suggests, only affects the local scope. As a result, no updates are visible after a representation change. The problem identified here is that local representation changes not only change the representation of a variable but also its identity. A fresh data object, with a fresh identity, is created and a reference is updated. The original object, however, still exists and retains it original identity. The solution here would be to update the value of all references to the new data object at once. In Smalltalk [36], for instance, this can be realised through

```
for (int i=0 ; i<iterations ; i++) {
    // i counts the number of iterations
    for (int r=0 ; r<src.getRows() ; r++) {
        for (int c=0 ; c<src.getCols() ; c++) {


            tar.set(r,c, heat(src,r,c));
        }
    }
    Matrix tmp = src;
    src = tar;
    if ( i==T )
        tar = new SparseMatrix(...);
    else
        tar = tmp;
}
```

```
for (int i=0 ; i<iterations ; i++) {
    int zeros = 0;
    for (int r=0 ; r<src.getRows() ; r++) {
        for (int c=0 ; c<src.getCols() ; c++) {
            double newHeat = heat(src,r,c);
            if (newHeat==0) zeros++;
            tar.set(r,c, newHeat);
        }
    }
    Matrix tmp = src;
    src = tar;
    if ( zeros>S )
        tar = new SparseMatrix(...);
    else
        tar = tmp;
}
```

Figure 2.8: Left: After `T` iterations switch to sparse. Right: When `zeros` is large switch to sparse.

the use of `become:`. Without language support, however, this approach is not feasible because updating all references is not feasible. Alternatively, one could opt to only apply representation changes on global variables, but this hampers flexibility. Moreover, the use of global variables is generally considered to be bad coding practice.

Application Logic Entangled with Representation Change Logic. Introducing a local representation change does not introduce code scattering, because the change is by definition local. Code entanglement, however, is large. The code fragment in listing 2.17, for instance, consists of two different logical parts. A first part, from line 2 to line 6, expresses the logic responsible for changing the representations of the arguments `mA` and `mB`. Lines 8–12, on the other hand, express the actual application logic.

Figure 2.8 also consists of two logical parts: application logic and representation change logic. Both parts, however, are *entangled*. This is visualised in figure 2.8 where application logic is marked in green (■) and the representation change logic is marked in grey (■).[6] As already mentioned, aspect-oriented programming proposes solutions to these issues [49].

2.2.5.3 Combination of Representations

A third, and final, ad-hoc solution to support data representation changes is the introduction of a *new representation* that explicitly implements representation changes. An example thereof is the matrix implementation in listing 2.9 which allows representation changes by simply replacing the value of field `coord2idx` with a different implementation chosen from listing 2.10. Again, this ad-hoc solution posed two problems: the approach is not scalable and in statically-typed languages this approach is too restrictive.

Not Scalable. This approach only works for dense matrices that store their elements based on some linearisation scheme, e. g., row-major order and column-major order are

[6] The meaning of the other colours (i. e., ■ and ■) is discussed in chapter 8.

both translations from a two-dimensional coordinate of a matrix into a one-dimensional coordinate of an array. In listing 2.9 (on page 27), this linearisation scheme is reified in the function pointer `f`. Implementing a representation that can not be captured as a translation between coordinate systems is thus not possible by replacing `f` with another value. In general, combining heterogeneous representations is not feasible, because it requires significant programmer effort to incorporate each new representation.

Multi Typed. From the "type" point of view, a combined representation is a subtype of all the representations it combines. In practice, this implies the need for a language with support for *multiple inheritance*. In chapter 9 we show that, oddly enough, most approaches related to representation changes adhere to the inverse relation. This inverse relation implies that the different representations are subtypes of one super type. In a statically-typed language, "changeable objects" require the super type as static type. In chapter 4 we present examples where this is too restrictive.

2.2.6 Levels of Granularity

The two example programs discussed in this section, the multiplication and the stencil computation, exemplify the need for data structures with changing representations. Note that in both examples, the decision to change the representation is made on a conceptually different level of granularity. In the matrix multiplication example, the representation change is expressed from the view point of the *computation* that *uses* a matrix, i. e., the `mul` method. In the stencil computation example, on the other hand, the representation change is expressed from the viewpoint of the matrix *itself*, i. e., depending on the sparsity. We discuss this difference in more detail in section 3.2.3 where we conclude that the former describes a static quantification over the code, whereas the latter describes a dynamic quantification over the code [32].

We do not need data structures . . . we need lots of data structures.

In this section we argue that the representation of data objects should be changeable at runtime. Dynamically changing the representation of data is not widely adopted because contemporary programming languages do not provide sufficient support to make the implementation of such "just-in-time data structures" feasible.
A language where representation changes are an integral part of the semantics should support:

- preservation of identity in the occurrence of a representation change;
- the possibility to disentangle application logic and representation change logic;
- the possibility to express both static and dynamic representation change logic.

2.3 Towards A Language with Support for Representation Changes

This chapters draws two conclusions (cf. the boxes on page 25 and page 39): predicting and improving performance is hard and changing the representation of data can be beneficial for performance. Programming languages as such can not reduce the complexity of performance engineering itself. They can, however, make the complexity of performance engineering combined with software engineering manageable. We conclude from section 2.1 that this can be realised by separating application logic for performance related logic, because this allows us to reason about the functional part of the application separate from the performance part of the application. Section 2.2 illustrates that there are applications where changing the representation is beneficial from a performance point of view. However, language support for this kind of representation changes is not yet developed.

We therefore argue in favour of a language with support for representation changes. As suggested in section 2.1, such a language needs to support the separation of *application logic* from *representation change logic* to allow application developers and performance experts to work more independently. Furthermore, as identified in section 2.2, such a language needs to support both *static* and *dynamic* representation change logic, and should allow objects to retain their *identity* after a representation change. **The main technical contribution of this dissertation is the design and implementation of a language that meets the aforementioned requirements: JITds**.

This chapter identifies the need for languages with support for representation changes, mainly with the goal of improving the performance. In the next chapter, we present and discuss a taxonomy for the design space of languages with support for representation changes in general. At the same time this allows us to present more formal definitions of terms, informally introduced in this chapter, such as "'static and dynamic representation changes" and "identity preservation". Then, in chapters 4 and 6 we introduce the language JITds itself.

Chapter 3

Matching Data and Computation

Algorithms + Data Structures = Programs — Niklaus Wirth [88].

Developing software is all about combining *computation* and *representation* according to Wirth [88]. Even though much has changed in software engineering and programming language design, in essence Wirth's statement still applies. Writing programs with good performance can be realised by improving the algorithms to better match the used data structures, or by improving the data structures to better match the used algorithms.

The goal of this chapter is, on the one hand, to introduce *terminology*, and to explore the design space of programming languages where changing the data representation is a core feature. To this end, we first define what a *representation change* exactly is in section 3.1. Then, in section 3.2, we introduce a taxonomy of language design choices w.r.t. data representation changes. This taxonomy consists of seven dimensions and is distilled from the literature.

3.1 Representation Selection or Representation Change.

> Broadly speaking, the history of software development is the history of ever later binding times. — Encyclopedia of Computer Science[67]

In chapter 1 we defined the data representation of a data object as the concrete realisation of a data interface. At execution time, when a program interacts with a data object, it conceptually interacts with a data interface, but it effectively interacts with the concrete data representation of that data object. We also know that it is possible to define different representations that encode the same data interface [88, 53, 1]. Hence, at some point in time, the choice of which representation to use for which data object must be made. This choice, and the strategy that leads to this choice, is what we call *data representation selection*. We now discuss when, in the life time of a data object, such a representation selection can occur. Figure 3.1 shows a time line which represents the life time of a data object.

The left part of figure 3.1 — an adaptation of the *System Development Life Cycle* (SDLC) [68] — shows the life time of a data object before execution (offline). In this

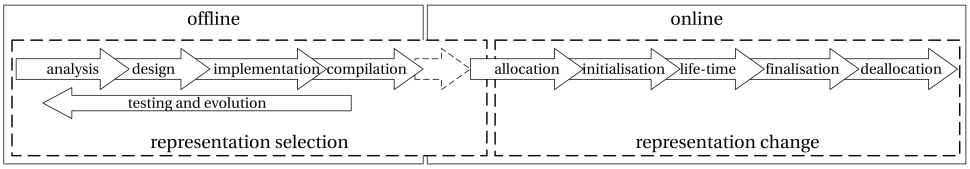


Figure 3.1: The "life time" of a data object: from being identified in the analysis phase of the software development life cycle, through allocation at runtime, to deallocation in the end.

part the data object is introduced in a software product or it is adjusted according to the requirements identified in the development cycles of the software system. When the choice of a data representation is made in this part, we refer to it as *offline representation selection*, because the representation is fixed before execution time.

The right part of figure 3.1 shows the life time of a data object during execution (online). In this part, the data object is consecutively allocated, initialised, "used", finalised, and deallocated. In the evolution of programming languages, ever more dynamic strategies for representation selection have been proposed over the years, effectively delaying the moment during the development cycle where the representation of a data object is fixed. Today, representation selection can be delayed until execution time. In this case we refer to it as *online representation selection* because fixing the data representation is postponed until runtime.

Today, in most programming languages, once a data object is allocated and initialised according to the selected representation, that representation remains fixed. However, some programming languages exist where the representation of a data object can be changed after the initial representation selection. We call such a *renewed* representation selection a *representation change*.

Definition

A data representation change is a renewed data representation selection for a data object during the execution of a program.

Thus, as shown in figure 3.1, there exist three types of representation selection: 1. offline representation selection, 2. online representation selection, and 3. representation change. Note that we do not differentiate between online and offline representation changes. Because we define representation changes as a renewed representation selection for an already existing data object, "Offline Representation Changes" would be a contradiction by definition. Put differently, representation changes inherently happen online. Note that during the feedback loop of a software engineering cycle (right-to-left arrow in figure 3.1), the decision can be made to change the representation of an object. The selection of such a new representation, however, is statically decided, and is hence not considered to be a representation change in our work.

Listing 3.1: Choosing a Data Representation Online

```
1  Matrix c = MatrixFactory.create(rs, cs);
```

Listing 3.3: Selecting the `RowMajorMatrix` representation for `m`, which changes to `ColMajorMatrix` later on.

```
1  m := RowMajorMatrix new.                "Representation Selection"
2  ...                                     "Some Computation"
3  m become: ( ColMajorMatrix new: m ). "Representation Change"
```

Offline Representation Selection *Offline representation selection* is almost as old as programming and software engineering itself and stems from the time when separation of interface and implementation was proposed as best practice. Examples are C/C++ header files, which can be linked and compiled with different C/C++ implementation files. Another straightforward example of offline selection is shown in listing 3.8, where lines 1 and 2 reveal the explicit design choice of a developer to select the representations `RowMajorMatrix` and `TransposableMatrix` for variables `mA` and `mB`, respectively.

Online Representation Selection. With *online representation selection*, the decision of which data representation to use is postponed until runtime. In object-oriented software engineering this is a common practice, as exemplified by various instances of both the factory patten and the more complex builder pattern [34]. Listing 3.1 for instance, makes use of a factory class `MatrixFactory`. An implementation of `MatrixFactory` is given in listing 3.2. Note that `MatrixFactory` decides dynamically which of the representations to use.

Listing 3.2: MatrixFactory

```
1  class MatrixFactory {
2    static Matrix create(int rows, int cols ) {
3      if ( cols>rows )
4        return new RowMajorMatrix(rows,cols);
5      else
6        return new ColMajorMatrix(rows,cols);
7    }
8  }
```

Representation Changes Listing 3.3 shows a snippet of Smalltalk code. On line 1 the `RowMajorMatrix` representation is selected for `m`. On line 2, the representation of `m` is changed to `ColMajorMatrix`. Here, to realise the representation change we used the `become` method. This is a language feature available in Smalltalk which changes the identity of the receiver to its argument[11].

Listing 3.4: On 01/01/2015, John Doe changed the representation of `myList` from `ArrayList` to `LinkedList`

```
1 /*
2  * Used to be ArrayList before 01/01/2015
3  * Changed to LinkedList because of performance reasons.
4  * see report of 26/12/2014 on memory access.
5  * John Doe
6  */
7 List myList = new LinkedList();
```

Offline Representation Change Listing 3.4 shows a piece of Java code where John Doe has "changed" the representation of `myList` during software development. However, as discussed in section 3.1, we do not consider such "offline representation changes" in our work.

3.2 Seven Dimensions of Representation Changes

When studying the literature on programming languages with support for representation changes, we identify *seven questions* that allow us to classify these languages. We argue that the design space of programming languages with support for representation changes is characterised by the answers to the following questions:

- **Q1: Who** is responsible for data representation changes?
- **Q2: How** is a data representation change realised?
- **Q3: When** is a data representation change executed?
- **Q4: Which** data representation changes are possible?
- **Q5: How long** does a data representation change take?
- **Q6: What** is altered after a data representation change?
- **Q7: Why** are data representation changes introduced?

In sections 3.2.1 to 3.2.7 we refine and discuss these questions in more detail.

3.2.1 Q1: Who is responsible for data representation changes?

The representation of a data structure does not automagically change by itself. Either the representation changing functionality is explicitly added by a *developer*, or data representation changes are an inherent part of the *environment* (e. g., a language's semantics or a framework's functionality). We say that data representation changes are the responsibility of either the developer or the environment.

The question of whether the developer or the environment is responsible for representation changes, however, does not necessarily have an "all or nothing" answer. For

instance, in section 3.2.2 we pose the question **how** representation changes are realised and in section 3.2.3 we pose the question **when** representation changes are performed. It is possible, as in Ureche et al. [85] for instance, that the developer is responsible for describing **how** to change the representation, whereas the environment is responsible for deciding **when** a representation is changed. Hence, we could pose the responsibility question in the context of each of the remaining six dimensions of our taxonomy.

Examples. An example where the runtime is responsible for representation changes is *Storage Strategies* [10]. Storage Strategies is a part of the PyPy interpreter, which changes the representation of collections depending on the type of elements. A more thorough discussion on Storage Strategies can be found in section 9.2.1.

Conversely, in Smalltalk representation changes can be realised through a call to `become`. It is the responsibility of the developer to insert the appropriate calls. Let us consider listing 3.3 as one concrete example. This Smalltalk fragment shows both a representation selection (line 1) and a representation change (line 3). In this fragment it is the decision of the developer to select and change the representation of the data object bound to `m`.

3.2.2 Q2: How is a data representation change realised?

A data structure does not automagically know **how** to change its representation. There has to be some part of the program responsible for the actual conversion from one representation to another. The part of the program that expresses this transition is called the *transformation logic*.

3.2.2.1 Orderly Representation Changes.

As we discuss in section 3.2.1, the burden of expressing transformation logic is either put on the developer or it is the responsibility of the environment. Some programming languages that put the burden on the developer provide a *dedicated programming construct* to express the transformation logic. In this case, we say that the language has support for *orderly representation changes*.

Example in LDL and ADRT. We discuss LDL and ADRT in detail in section 9.1.1. As we will show, in LDL and ADRT a *transformation description object* needs to be defined for each pair of representations between which representation changes need to be possible.

Example in Java. In Java, when a data object needs to be displayed on a screen (e. g., `System.out.println`), then the data object is first converted into the `String` representation. By default an object is then printed as the concatenation of its class name, an at-sign, and a hash-code, cf. the default implementation of `toString` in `Object` [60].

Because Java supports method overriding, it is possible to implement a different conversion. Listing 3.5, for instance, shows how to convert an instance of `Point2D` to a string that reveals the coordinates. The JVM does not treat `toString` differently from any other method. However, we do consider `toString` as Java's "special construct" to

Listing 3.5: Converting an instance of `Point2D` to `String`.

```
1  public class Point2D {
2
3    int x, y;
4
5    public String toString() {
6      return "(" + this.x + "," + this.y + ")";
7    }
8  }
```

convert objects into strings. We can argue that Java supports orderly transitions between any object and `String` by means of the `toString`-method.

3.2.2.2 Internal or External Transformation Logic.

The transformation logic can either be a part of the definition of a data structure or not. Hence, we differentiate between data structures with *internal transformation logic* and data structures with *external transformation logic*. In other words, internal transformation logic is transformation logic encapsulated in the definition of a data structure.

3.2.3 Q3: When is a data representation change executed?

A data structure does not automagically know **when** to change its representation. There has to be some part of the program responsible for initiating an actual representation change. We call the part of the program that is responsible for initiating a representation change the *representation change incentive* (code).

3.2.3.1 Internal or External Transformation Incentive.

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 incentives* are initiated by the data structures itself, i. e., as part of their implementation. Conversely, we dub the representation change incentive as *external* when the new representation is imposed on the data structure from the outside of that data structure.

Representation Change Incentive and Transformation Logic: Comparing Internal and External Implementations We use the example of *a set of integers* to illustrate the difference between (internal and external) transformation logic and (internal and external) representation change incentive. Assume there are two representations for a set: as a sorted collection or as an unsorted collection. Then, a sorting algorithm effectively describes how to transform an unsorted collection into a sorted collection, i. e., the sort algorithm is the transformation logic. Invoking such a sorting algorithm effectively causes

Table 3.1: Representation Change Incentive and Transformation Logic: Comparing Internal and External Implementations

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="2">Transformation Logic</th></tr>
<tr><th>internal</th><th>external</th></tr>
<tr><td rowspan="2">Change Incentive</td><td>internal</td><td>User Code
1 myList.add(x);

Representation Code
1 public void add(Object x) {
2 this.data.add(x);
3 this.sort();
4 }</td><td>User Code
1 myList.add(x);

Representation Code
1 public void add(Object x) {
2 this.data.add(x);
3 Collections.sort(this);
4 }</td></tr>
<tr><td>external</td><td>User Code
1 myList.add(x);
2 myList.sort();

Representation Code
1 public void add(Object x) {
2 this.data.add(x);
3 }</td><td>User Code
1 myList.add(x);
2 Collections.sort(myList);

Representation Code
1 public void add(Object x) {
2 this.data.add(x);
3 }</td></tr>
</table>

the transformation to take place, i. e., the call to `sort` is the representation change incentive code. The remaining question is whether the transformation logic or the representation change incentive are part of the definition of the set (internal) or not (external).

Table 3.1 shows all four combinations. The two examples in the top row show how *the list itself* is responsible for ensuring that its elements are sorted (cf. line 7). The examples on the bottom row, on the other hand, show how *the user* of the list invokes some sorting algorithm and enforces the sorted representation on the set (cf. line 3). The examples in the left column exemplify a set representation with its own implementation of a sorting algorithm (cf. `sort()`, used on line 7(top) and line 3 (bottom)). The examples in the right column rely on the sorting functionally implemented external to the set representation (cf. `sort` of `Collection`, used on line 7(top) and line 3 (bottom)).

3.2.3.2 Static or Dynamic Incentive.

Orthogonal to the question of whether the representation change incentive code is internal or external, is the question about at which *moment(s)* during the execution of the program a representation change has to occur. Using the terminology from AOP [32], we can say that representation change incentive code makes quantified statements about the circumstances in which to execute a representation change. From Filman and Fried-

man [32] we know that there exist two kinds of *program quantification*: quantification over the static structure of a program and quantification over the dynamic behaviour and state of a program.

Hence, we differentiate between *static representation change incentive code* and *dynamic representation change incentive code*. The former, static representation change incentive code, identifies lexical places in a program where representation changes have to occur (static). The latter, dynamic representation change incentive code, identifies moments in the execution of a program where representation changes have to occur (dynamic).

3.2.4 Q4: Which data representation changes are possible?

The representation changes for a data object can be modelled naturally by a graph, with representations as vertices, and the possible representation changes as edges between these vertices. We call such a graph a *transition graph*. Consider `OpenFile`, `ClosedFile`, and `LockedFile` as three representations of the abstract data type `File`. An `OpenFile` can be closed and a `ClosedFile` can be opened. Both an `OpenFile` or an `ClosedFile` can be locked, while a `LockedFile` remains locked for ever. Figure 3.2 visualises the *intended representation changes* for a `File` as a transition graph. We say that `File` is the *denominator* of this transition graph.

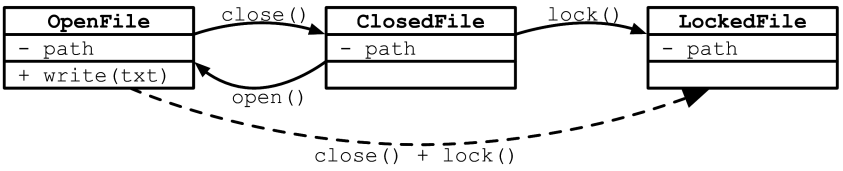


Figure 3.2: The states of a `File`: **O**pen, **C**losed, **L**ocked.

In the most liberal programming language, all data objects (regardless of their representation) can change their representation to any other representation available in the system. In such a programming language it is not possible to restrict the representation changes to only the *intended representation changes*. In the most constraining programming language, no data representation changes are allowed at all. In such a programming language it is not possible to express any representation change.

The design space that describes **which** representation changes are allowed, covers the complete continuum between these two extremes. As a result, each programming language gives rise to transition graphs with different characteristics. We can categorise programming languages according to the *characteristics of the transition graphs* in that programming language. We identify that the characteristics of a transition graph are defined by: the representations that are allowed to be vertices in the transition graph, the (type) relation between the representations in the transition graph, and the possible edges in the transition graph. Orthogonal to which transition graphs are possible in a given programming language, is the question wether the transition graph is explicitly

created by a developer or implicitly defined by the programming language's semantics (cf. the **who**-dimension). First, however, we focus on the characteristics of the transition graphs themselves.

To compare the different characteristics of the transition graphs in different programming languages, we use Smalltalk as the point-zero. Smalltalk proves to be the most liberal programming language in this respect, as the `become` operation effectively can be used on any object and change its representation to any other available representation. Hence, the transition graph in a Smalltalk system is isomorphic with the *complete graph* (K_n), where the vertices are *all* instantiatable classes in the Smalltalk system and where any representation change is possible (i. e., all possible edges).

Listing 3.6: When a `Frog` gets `kissed` it becomes a `Prince`. But `Player john` can never change its representation to `Sword`.
The example in this listing is adapted from [27]. More on Fickle$_{II}$ can be found in section 9.1.4.

```
1  root class Player { ... }
2  root class Item   { ... }
3
4  state class Prince extends Player { ... }
5
6  state class Frog   extends Player {
7    void kissed(){Player} {
8      this⇓Prince; // allowed
9    }
10 }
11
12 state class Sword  extends Item   { ... }
13
14 Player john = new Frog();
15 john.kissed(); // allowed
16 john⇓Sword;    // not allowed
```

3.2.4.1 Transition Graph Vertex Membership.

The vertices in a transition graph are the representations that are "available" for a given data object. Depending on the programming language, all representations (e. g., all classes) or only representations of a certain category are available as possible new representations.

No Restrictions on Representations. As stated before, in Smalltalk the `become` method allows any object to become any other object. Since any class is a viable new representation, we say that Smalltalk puts no restrictions on transition graph membership.

Annotated Representations only. Other languages exist where representation changes are only possible between representations of a *certain kind* of representation. Fickle$_{II}$ [27], for instance, introduces a special kind of class called *state classes*. In Fickle$_{II}$ representation changes are only allowed from one *state class* to another. *Root classes*, on the other hand, are introduced to annotate a class as denominator of a representation changeable type. Listing 3.6 shows the keywords in Fickle$_{II}$ to declare classes as root or

Listing 3.7: `TCPEstablished` is one of the representations of the denominator `TCPConnection`. `TCPEstablished` is a implementation-only class.

```
1  implementationonly class TCPEstablished
2  extends TCPConnection {
3    ...
4  }
```

state classes. Another example is Gilgul, where representation changes are only allowed between *implementation-only classes* [15] (see listing 3.7).

3.2.4.2 Relation between Transition Graph Vertexes.

Orthogonal to the constraints on potential members of a transition graph are the constraints put on the (type) relation between the vertices in any given transition graph. Depending on the programming language, all representations can occur together in a transition graph or only representations that share a certain relation can occur together in a transition graph.

No Restrictions on Relations. In Smalltalk the `become` method allows any object to become any other object. Since any class is a viable new representation, we say that Smalltalk puts no restrictions on the relation between vertices in a transition graph.

Only Sibling Classes. In Fickle$_{II}$, on the other hand, only *state classes* that have the same *root class* as an ancestor are allowed to form a transition graph. Thus, Fickle$_{II}$ puts a constraint on the relationship between the vertices of the transition graphs. Consequently, `Player john`, from listing 3.6, can never change its representation to `Sword`, because `Sword` can never be a part of the transition graph of `Player`.

3.2.4.3 Transition Graph Edges.

Besides vertices, a transition graph also has edges. Each edge represents a representation change that is allowed at runtime. The most liberal languages, with respect to allowed transitions, are those with the implicit assumption that transitions between any two representations are possible. In these languages a transition graph is a *complete graph* (K_n). Alternatively, not all transitions are allowed, either because they have to be developer defined or because the language semantics prohibits certain transitions. These restrictions have an impact on the shape of the transition graph.

No Restrictions on Edges. In Smalltalk the `become` method allows any object to become any other object. Hence, in Smalltalk's transition graphs, there exists an edge between any two representations. We say that Smalltalk does not put any restrictions on the available edges in a transition graph. Moreover, all possible edges are implicitly present in Smalltalk's transition graphs, they are thus by definition complete transition graphs.

Edges Explicitly Defined by Developer. In *Ad hoc Data Representation Transformation* (ADRT) representation changes are defined by a *transformation object* [84]. Such a *transformation object* encapsulates the transformation logic between a high-level representation and a low-level representation. ADRT requires the developer of such transformation objects to provide an implementation for both the `toHigh` and the `toRepr` method, which respectively describe how to transform from a high level to a low level representation and the other way around. Hence, by construction ADRT only supports transition graphs with exactly two edges. More on ADRT can be found in section 9.1.1. Note that the developer is explicitly responsible for defining the transition graph in ADRT.

Edges Implicitly Defined by Language Semantics. Cohen and Gil [13], on the other hand, describe a system with support for representation changes where objects can change their representation but only to a subtype of the current representation. There, the transition graph is homomorphic with the inheritance tree where the "can-transition-to" relation is mapped onto the "is-direct-superclass-of" relation. In between approaches exist as well: effectively all representation changes are allowed but they have to be explicitly facilitated by a developer. This is further discussed in section 9.1.3. Note that the language semantics of Object Evolution (cf. environment) implicitly define what the transition graph looks like.

3.2.4.4 First-Class Transition Graphs.

To the best of our knowledge there are no languages where the transition graph can be considered as a first class citizen in the language. By consequence, we are not aware of any programming language where the transition graph can be changed dynamically by the program or by the programmer.

3.2.5 Q5: How long does a data representation change take?

When it is possible to unambiguously determine the representation of a data object, both before and after a representation change, we say the representation change is *instant*. Conversely, when it is possible that after a representation change a data object is found to be in multiple representations simultaneously, we refer to the representation change as *gradual*.

While instant representation changes are often more straightforward to implement, they can suffer from performance penalties when invoked too often. Gradual representations are usually an attempt to hide the latency of premature changes as they allow for cheap rollback or because they also maintain the old representation, e. g., for modifiers only. Their improved performance measured in execution time is often traded off for more space.

Instant Representation Change. Recall that to have an *instant representation change* it must be possible to unambiguously determine the representation of a data object, both before and after a representation change is invoked. In listing 3.8, the method

Listing 3.8: Internal Transformation Logic.

```
1  Matrix mA = new RowMajorMatrix(rs, cs);
2  Matrix mB = new TransposableMatrix(rs, cs);
3
4  mB.enforceColMajorOrder();
5  Matrix mC = mul(mA, mB);
```

`enforceColMajorOrder` is invoked on a data object `mB` of class `TransposableMatrix`. In listing 3.9, a concrete implementation for the class `TransposableMatrix` is presented. The boolean flag `rowActive` is used to decide which indexing function to use (see section 2.2.1). The effective representation change is realised through the invocation of the method `transpose`, followed by updating the value of `rowActive`. Before and after the call to `enforceColMajorOrder`, the current representation of a transposable matrix is uniquely defined by the value of the flag `rowActive`. Hence, this an example of an *instant representation change.*

Gradual Representation Change. Listing 3.10 presents a different implementation for `TransposableMatrix` where it is not possible to unambiguously determine the current representation of a matrix because two representations are maintained simultaneously. The implementation is roughly equivalent to listing 3.9 except for lines 2, 7, 12, 17, and 19.[1] The main difference in implementation is that there are two arrays in listing 3.10 that both store the data. The mutator `set` updates both arrays, whilst the accessor `get` only consults the active representation's data. This implementation allows for improvements in execution time over the implementation shown in listing 3.9 when the number of `get`s greatly exceeds the number of `set`s. The cost of a representation change comes at the cost of inverting a boolean flag. Of course, maintaining two arrays comes at the cost of doubled memory consumption.

3.2.6 Q6: What is altered after a data representation change?

3.2.6.1 State, Behaviour, or Both.

In the context of object-technology, a "data object" is an object. Moreover, in object-technology, an object encapsulates both state and behaviour. Hence, after a representation change, it is possible that either, or both, *state* and *behaviour* have changed.

3.2.6.2 Genuine Representation Change versus Local Representation Change.

It is important to note that a representation change is defined in terms of a *data object* (cf. section 3.1) and not in terms of a *variable or reference.* We say that a *genuine representation change* keeps the *identity* of its subject intact. This implies that after a genuine

[1] Technically, lines 25 and 27 differ as well, but there the difference is accidental, i. e., because of a variable name.

Listing (3.9) Supporting Multiple Representations

```
1  public class TransposableMatrix {
2    int rows, cols;
3    int data[];
4    boolean rowActive = true;
5
6    public void enforceRowMajorOrder() {
7      if (!rowActive) this.transpose();
8      rowActive = true;
9    }
10
11   public void enforceColMajorOrder() {
12     if (rowActive) this.transpose();
13     rowActive = false;
14   }
15
16   public void set(int r, int c, int v) {
17     if (rowActive)
18       data[r*cols+c] = v;
19     else
20       data[r+c*rows] = v;
21   }
22
23   public int  get(int r, int c) {
24     if (rowActive)
25       return data[r*cols+c];
26     else
27       return data[r+c*rows];
28   }
29 }
```

Listing (3.10) Maintaining Multiple Representations

```
1  public class TransposableMatrix {
2    int rows, cols;
3    int dataRM[], dataCM[];
4    boolean rowActive = true;
5
6    public void enforceRowMajorOrder() {
7
8      rowActive = true;
9    }
10
11   public void enforceColMajorOrder() {
12
13     rowActive = false;
14   }
15
16   public void set(int r, int c, int v) {
17
18     dataRM[r*cols+c] = v;
19
20     dataCM[r+c*rows] = v;
21   }
22
23   public int  get(int r, int c) {
24     if (rowActive)
25       return dataRM[r*cols+c];
26     else
27       return dataCM[r+c*rows];
28   }
29 }
```

Listing 3.11: When x becomes a boat, its alias y has become that same boat.

```
1  x := Car new.
2  y := x.
3  x become: (Boat new).
4  y sailTo: 'England'.
```

representation change, all aliases that refer to the original data object have also changed since they share the same identity. In other words, a genuine representation change **preserves** the identity of a data object.

Most programming languages, however, do not provide language constructs that are powerful enough to express genuine representation changes without introducing an extra layer of indirection (cf. handles). In these languages, *assigning* a new object to a variable can be considered a "poor man's representation change" which we call a *local representation change* in section 2.2.5.2. In other words, a local representation change is an intentional representation change where identity preservation can not be guaranteed, e. g., by local assignment.

A *genuine representation change* alters the representation of a data object, but not its identity. Not many languages support genuine representation changes. Smalltalk, by means of `become`, does. The fragment of Smalltalk code in listing 3.11 makes use of `become` to realise a representation change for the data object bound to `x` from `Car` to `Boat` (line 3). `y`, an alias of the `Car`-object referenced by `x`, has also become a `Boat` — the same exact `Boat` — after the call to `become`. Because Smalltalk's `become` has identity preserving semantics, it can be used to realise a *genuine representation change*.

Other examples of programming languages with support for genuine representation changes are Fickle$_{II}$ [27], Gilgul [15].

Most conventional languages (e. g., Java, C/C++, C#) have no support for genuine representation changes. Listing 3.12 show how *local assignment* can be used to realise a *poor man's representation change*. On line 7, for instance, the parameter `mB` changes from `RowMajorMatrix` to `ColMajorMatrix` by means of a simple assignment. After the representation change, `mB` no longer references the same data object, i. e., the identity is lost. Moreover, because of Java's call-by-value semantics, the `mB` on line 2 still references a data object in the `RowMajorMatrix` representation after the call to `mul`. In this example the representation change does not preserve identity and is only visible locally, cf. *local representation change*.

3.2.7 Q7: Why are Data Representation Changes Introduced?

General Applicability versus Specific Use Cases Finally, we observe that representation changes are introduced based on different motivations. First, we differentiate based on the applicability of the representation changes: one specific use case or generally applicable. Second, we identify a set of use cases where representation changes are the natural solution.

Listing 3.12: Assigning a "derived" value to a local variable.

```
1  Matrix mA = new RowMajorMatrix(rs, cs);
2  Matrix mB = new RowMajorMatrix(rs, cs);
3  Matrix mC = mul(mA, mB);
4
5  public Matrix mul(Matrix mA, Matrix mB) {
6    mA = new RowMajorMatrix(mA);
7    mB = new ColMajorMatrix(mB);
8    ...
9    return mC;
10 }
```

3.2.7.1 Specific or General.

Representation changing techniques can be deployed in two possible ways. First, we see *specific* techniques that are tailored towards a well defined set of scenarios, which can be deployed as-is, off the shelf. These specific 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-arrays" discussion, for instance, applies the same technique to more heterogeneous data [77].

3.2.7.2 Scenarios.

We compiled a non-exhaustive list of scenarios where changing the representation of data can improve performance. In chapter 8 we implement an example program for each of these five scenarios and evaluate the impact on program performance and program structure. Here, we give an overview of these scenarios.

Changing Data Access Pattern On contemporary hardware, the data access pattern has a significant influence on a program's performance. Hence, a data representation is often optimised towards a specific data access pattern. If, however, the data access patterns changes during the execution of the program (e. g., matrix multiplication, see section 2.2.2), then changing the representation to match the new data access pattern might pay off.

Changing Interface Usage Pattern Different representations can exhibit a different algorithmic complexity on the same basic operations. Random access, for instance, is $O(1)$ in a vector-based representation whereas it has a complexity of $O(n)$ in a pointer-based representation. Adding an element to the front, on the other hand, runs in $O(n)$ in a vector-based representation whereas it has a complexity of $O(1)$ in a pointer-based representation. When the usage pattern of a data object changes, e. g., no new elements and frequent reads, changing the representation to match

the new usage pattern might pay off. As opposed to a changing data access pattern, a changing interface usage pattern happens only at the *algorithmic level*.

Changing Functionality While a program executes, the operations that are conceptually applicable on an object may change. This is sometimes called objects with *modes*, where each *mode* is associated with a set of available operations. These objects with modes, have led to an extension of object-oriented programming called typestate-oriented programming [3]. A classic typestate-oriented programming example is a file that can be either open or closed. Depending on its "current state", the `write`-operation may or may not be enabled. This pattern can be easily expressed using representation changes, i. e., "open file" and "close file" are two representations (cf. *modes* or typestates) between which it is possible to transition. Hence, typestate-oriented programming relies on representation changes to change the current functionality of data objects.

Freezing and Thawing Objects Many ADTs have different implementations. Broadly speaking, we can categorise implementations into two groups: those representations that are tailored towards updating, and those representations that are tailored towards querying. "Freezing and thawing" is a well-known pattern, for instance in Haskell[82], to model the transitions between phases of updating the data and phases of querying the data. Both phases prefer their data objects to adhere to a representation that is better-performing for that phase's task, e. g., easily updateable or easily query-able. This is realised by marking a data object as immutable (freeze) or as mutable (thaw) at various places in the program.

Data Specialisation Having more information about the characteristics of input data allows for more specialised representations. In Java, for instance, using `int[]` (a native array of integers) to store a set of integers, performs much better than using `Integer[]` or `Object[]` (an array of references to boxed objects). This is because of auto-boxing, which requires extra space to store the `int` in an object, and which requires extra time to dereference the `int` held by the object. Other characteristics about data (i. e., other than static type) are only known at runtime. In cases where a specialised representation exists for data with an (dynamically) observed characteristic, it might be interesting to change to this specialised representation. For instance, when a matrix is observed to be sparse, using the `SparseMatrix` representation can improve the performance.

From the literature, we identified that representation changes can be beneficial or needed when the *data access pattern changes*, when the *usage pattern changes*, when the *functionality of objects changes*, when the *mutability of objects changes* (cf. freezing and thawing), and when the *(input) data changes*.

3.2.8 Summary

Seven Dimensions of Representation Changes

The design space of programming languages with support for representation changes is characterised by the answers to seven questions:

Q1: Who is responsible for data representation changes?
Developer or Environment

Q2: How is a data representation change realised?

- Orderly Representation Changes
- Internal or External Transformation Logic

Q3: When is a data representation change executed?

- Internal or External Representation Change Incentive
- Static or Dynamic Representation Change Incentive

Q4: Which data representation changes are possible?

- Vertex Membership in a Transition Graph
- Relation between Transition Graph Members in a Transition Graph
- Edges in a Transition Graph
- First-Class Transition Graphs

Q5: How long does a data representation change take?
Instant or Gradual

Q6: What is altered after a data representation change?

- Changing State, Behaviour, or both
- Preserving Identity: Genuine or Local

Q7: Why are data representation changes introduced?

- Specific or General
- Set of Scenarios

This seven-dimension taxonomy is distilled from a survey of contemporary programming languages with support for representation changes. The actual survey is presented in chapter 9. From this taxonomy, we identify a vacuum in the landscape of programming languages, to allow developers to express data representations changes in order to realise *non-functional features*, such as for instance improving performance.

To fill this vacuum we propose JITds, a new programming language with support for online data representation changes. In chapter 4 we show how to define transformation logic as we explain how to *how* to change the representation of object in JITds. In chapter 6 we show how to express representation change incentive as we explain how to *when*

the representation of an object changes in JITds. In chapter 9 we discuss the work related to JITds, according to the taxonomy presented in this chapter.

Chapter 4

Designing Just-in-Time Data Structures: How to Change the Representation

In software engineering, programming against a *data interface* as opposed to programming with a particular *data representation* is considered best practice [53, 88]. This approach is mainly motivated by software engineering advantages such as modularity, maintainability, and evolvability. However, it can also be motivated by productivity and performance. In practice, it is often more productive to start out with a trivial, but working, prototype implementation rather than to invest a lot of time in a highly specialised implementation. Only later in the development cycle, if the initial data representation proves to be a performance bottleneck, the software engineer can decide to implement a more specialised data representation in order to improve performance. This is a software engineering strategy known as "avoiding premature optimisations" (cf. *Premature optimisation is the root of all evil — Donald Knuth [51]*).

The idea of separating *data interface* from *data representation* is almost as old as computer science itself and allows delaying the binding between *data interface* and *data representation* until compilation time. In the early days (i. e., before ±1990) only a single data representation was available per interface which was fixed at compilation and thus also remained fixed during the execution. The advent of object-technology facilitated the existence of *multiple* data representations for a single data interface at runtime. In a class-based object-oriented language, for instance, this is realised when multiple classes extend the same base class. At object allocation time a concrete representation is selected for each object. In chapter 3 we show that this can either be a static selection or a dynamic selection. Either way, with object-technology it is possible to delay the binding between interface and representation until object allocation time.

We developed JITds, a programming language in which the binding between data interface and data representation is *variable*, even after allocation time. In section 2.2 we show some example programs that become more efficient if the data representations changes during the execution of the program. We call such a dynamic transition between representations a *data representation change* in chapter 3. Chapter 3 identifies two concepts when dealing with representation changes. There is *transformation logic*

to describe **how** to change between representations, and there is *representation change incentive* to describe **when** to change between representations. The focus of this chapter is how to express *transformation logic* in JITds. Later, in chapter 6, we focus on how to express *representation change incentive* in JITds.

More concretely, in this chapter we explain how to define a *just-in-time data structure* in JITds. A just-in-time data structure is a *combination* of representations into a single data structure. An instance of a just-in-time data structure always has a "current" representation, but that current representation can change during execution. Although we could realise this through patterns or conventions in a traditional programming languages (see section 7.3.2), in this dissertation we choose to introduce just-in-time Data Structures by means of a new programming language called JITds. JITds is conceived as an extension to a conventional *statically-typed class-based object-oriented programming language*. The remainder of this chapter introduces the actual just-in-time data structures and puts them into context of existing well-studied areas such as object-oriented programming in general, and dynamic object reclassification and multiple inheritance in specific. To conclude this chapter, we show which of the goals presented in chapter 1 are reached by introducing swappable data structures and which goals are not. A motivation of why we chose a statically typed class-based object-oriented language as the base for JITds is given in section 4.1.

4.1 JITds: A Statically Typed Class-based Object-Oriented Language

JITds is an extension to a conventional *statically-typed class-based object-oriented language*, which supports online representation changes. We already motivated the need for data structures with changing representations in section 2.2, the motivation to tackle the subject by language design has been motivated in chapter 1. Here we motivate the choice for a statically-typed class-based object-oriented language as the base language of JITds.

Why an object-oriented language? In object-oriented languages (compound) data objects map one-on-one to "object" objects. At the same time, an object is more than just a compound data object. Its data interface, the set of available operations, is tightly coupled to the object itself. In short, object-oriented languages combine the concept of data representation and data interface into one entity: the object. Therefore, object-orientation proves to be the most natural base for JITds.

Why a class-based language? A *class definition* is a named template for creating objects. Informally, a class specifies which data fields are needed in an object and it specifies how the information can be accessed (read or written) from these data fields. A class effectively maps one-on-one to the concept of *data representation* and has the advantage that it has a name to which can be referred. It is a natural fit to build JITds as a class-based language because it is important for a just-in-time data structure to be able to refer to its representations by name.

Why a statically-typed language? In general, each class defines a new type. Above, we stated that changing the representation of an object in a class-based language ef-

fectively maps to changing the class of an object. Consequently, changing the representation implies changing the type of an object at runtime. Presenting our work in the context of a statically-typed language allows us to explicitly show that in JITds, changing the representation does **not** change the type of an object. From which it follows that executing a valid JITds program will never result in a type error at runtime. We prove this formally in chapter 5.

For convenience, we assume JITds to be a language with automatic memory management (i. e., a language with garbage collection), because this simplifies many discussions. In line with the vision of this dissertation to disseminate the idea of *changing representations*, we opt to present JITds as an extension of Java. We choose Java because it is arguably the best known statically-typed class-based object-oriented language.

4.1.1 JITds versus Java

JITds is a statically-typed class-based, object-oriented programming language with automatic memory management. In section 4.1 we discuss thoroughly why we make this choice. For all intents and purposes, JITds is an extension of of Java. In this chapter and the next, we focus on the *extension* part, i. e., how does JITds facilitates representation changes, something that is not generally possible in statically-typed class-based, object-oriented programming languages. To maintain the focus on the representation changes themselves, we only consider a subset of Java to be the base language of JITds. The following restrictions, for instance, reveal that JITds is not a *proper* extension of Java, but rather an extension of a *subset of Java*.

JITds does not support: 1. interfaces, abstract classes, or enums, only *simple classes*; 2. multi-threading; 3. generic types; 4. visibility modifiers (i. e., all fields are private, all methods are public); 5. packages; 6. overloading (all methods in a class have distinct names).

Syntax: Simple Class

```
class C {

  C_f fieldName;
  ...

  C_r methodName(C_0 a_0, ... , C_n a_n) { ... }
  ...

}
```

4.2 From one to many representations.

Implementing a data structure in a class-based object-oriented language is realised by declaring a new simple class. Such class optionally *extends* another class. Implementing

Listing (4.1) The class `RowMajorMatrix`.

```
1  class RowMajorMatrix  {
2    double[] data;
3    int rows, cols;
4
5    RowMajorMatrix(int rows, int cols) {
6      this.rows = rows;
7      this.cols = cols;
8      this.data = new double[rows*cols];
9    }
10
11   RowMajorMatrix(int rows, int cols,
12                  double[] data) {
13     this.rows = rows;
14     this.cols = cols;
15     this.data = data;
16   }
17
18   int   getRows()  { return rows; }
19   int   getCols()  { return cols; }
20   int[] getArray() { return data; }
21
22   int get(int row, int col) {
23     return data[row*getCols()+col];
24   }
25
26   void set(int row, int col, int val) {
27     data[row*getCols()+col] = val;
28   }
29
30   void transpose(){ ... }
31 }
```

Listing (4.2) The class `ColMajorMatrix`.

```
class ColMajorMatrix  {
  double[] data;
  int rows, cols;

  ColMajorMatrix(int rows, int cols) {
    this.rows = rows;
    this.cols = cols;
    this.data = new double[rows*cols];
  }

  ColMajorMatrix(int rows, int cols,
                 double[] data) {
    this.rows = rows;
    this.cols = cols;
    this.data = data;
  }

  int   getRows()  { return rows; }
  int   getCols()  { return cols; }
  int[] getArray() { return data; }

  int get(int row, int col) {
    return data[row+col*getRows()];
  }

  void set(int row, int col, int val) {
    data[row+col*getRows()] = val;
  }

  void transpose(){ ... }
}
```

a just-in-time data structure in JITds is realised by declaring a new *just-in-time class*. The main difference between a regular class and a just-in-time class is that a just-in-time class *combines* multiple classes into one. We call these classes the *representations* of the just-in-time class. These *representation classes* are simple classes.

Instances of a just-in-time class, just-in-time objects, always have one *current representation*. Such objects look and behave as if they are an instance of their current representation's class. Further, a just-in-time object can be the target of a *swap statement*. This statement instructs the object to change its current current representation to a different representation. To ensure an orderly transition between the old representation `Ri` and the new representation `Rj`, a representation change between representations `Ri` and `Rj` is only allowed if there exists a *transition function* that describes such a transition, i. e., from `Ri` to `Rj`.

Combining multiple representations and allowing dynamic orderly transitions between them is the core idea of JITds. In the remainder of this chapter we gradually introduce the language constructs needed to express the aforementioned combinations and transitions. We describe these constructs and provide illustrating examples. Every time a new construct is introduced, its syntax is presented in a *syntax box*, such as the one that shows the syntax of a *Simple Class*.

4.2.1 Combining Representations

To combine multiple representations into one just-in-time class, the developer defines a new class. Instead of specifying a super-class (cf. `extends` as in Java) and a list of interfaces (cf. `implements` as in Java), the definition of a just-in-time class is augmented with a comma-separated list of representation classes. This list is preceded by the keyword `combines`. To introduce new syntactic constructs, or in abstract examples, we often use `T` as the name of a just-in-time class, and the names of representation classes are usually denoted as `R` with an optional subscript. To instantiate a just-in-time class the developer calls a constructor of the preferred initial representation. Syntactically, this is realised by qualifying the name of the just-in-time class, with the name of the intended initial representation.

Syntax: Defining a just-in-time Class and creating a just-in-time object.

```
class T combines R0, ... , Rn { ... }

T t = new T.Ri(a0, ... , an);
```

We now illustrate the definition of just-in-time classes with two examples: a just-in-time `Matrix` and a just-in-time `File`. For now, we do not consider the body of the class definitions and the members defined within.

Matrix. Listing 4.3 shows on line 2 how `Matrix` combines two representation classes, `RowMajorMatrix` and `ColMajorMatrix`, into a single just-in-time class. Listings 4.1

and 4.2 present the implementation of the two classes `RowMajorMatrix` and the `ColMajorMatrix`. These implementations are slightly different from the implementation shown in section 2.2.1. The methods `getArray` and `transpose` are added. Moreover, the class hierarchy is flattened for convenience, i. e., there is no common super-class with the shared behaviour.

Listing 4.3: The JIT-class `Matrix` combines two representations.

```
1  class Matrix
2  combines RowMajorMatrix, ColMajorMatrix  {
3    ...
4  }
```

File. A second example, listing 4.4, combines three different representations into a single just-in-time class `File` (line 1). The three representations correspond to the three "states" a file can be in: open, closed, or locked (forever closed).

4.3 Representation Changes

Hitherto, we introduced how to combine multiple representations into a single just-in-time class and we showed how to create a new just-in-time object that adheres to a given initial representation. The core idea of JITds is, of course, to be able to change the representation of such objects at runtime. In this section we introduce the constructs to describe how a transition between two representations should be realised (cf. *transition logic*, chapter 3) and how to initiate such a transition (cf. *transition incentive code* in chapter 3). To this end, JITds introduces two new constructs:

- The *transition function* is a new kind of class member that encapsulates the transition logic. A transition function basically describes how to transition from one representation to an other.

- The *swap statement* is a new kind of statement to express a representation change (i. e., change incentive code). A swap statement basically calls a transition function on an object.

Listing 4.4: The class `File` combines three representations.

```
1  class File combines OpenFile, ClosedFile, LockedFile  {
2    ...
3  }
```

4.3.1 Transition Functions

JITds ensures orderly transitions between representations by requiring transition logic to be expressed as a *transition function*. Such a *transition function* is a member of a just-in-time class that describes the transition between two of its representations. Syntactically, this is expressed by stating the *source representation* and the *target representation* separated by the keyword `to` and followed by a sequence of statements between curly braces, which we call the *body of a transition function*.

Optionally, a transition function can be named, as shown in the second syntax case. A named transition function behaves like a parameterless method, i. e., when it is invoked, it causes the representation change described by the transition function to occur.

Syntax: Transition Function

```
R_source to R_target { /* body statements */ }

R_source to R_target as name { /* body statements */ }
```

A transition function resembles a parameterless constructor as they are known from conventional class-based languages. Within the body of a transition function two pseudo-variables can be used: `target` and `source`. These keywords 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 bound to `source`;
2. The first statement in the body invokes a constructor of the *target representation* and binds the newly created object to `target`;
3. During the execution of the body, both `target` and `source` exist as separate objects of a simple class;
4. After the execution of the body, the current representation is updated, i. e., the newly created object, denoted by `target`, replaces the *old* current representation (denoted by `source`).

Chapter 7, where a formal specification is given, discusses these semantics more concretely and two different prototype implementations are discussed in chapter 7.

In short, transition functions describe how a transition between two representations of a just-in-time class is realised. In theory, the potential number of transition functions grows quadratically with the number of combined representations. We argue in section 4.6, however, that in practice the number of transition functions that need to be implemented can be kept manageable.

We now revisit the two examples just-in-time classes: `Matrix` and `File`, which we introduced in section 4.2.1, and focus on their transition functions.

Matrix. Listing 4.5 shows a transition function (lines 4–9) that describes how to change the representation of a just-in-time `Matrix` from `RowMajorMatrix` to `ColMajorMatrix`. The first statement in the body, `target(...)`, (lines 5–7) creates a new object using the constructor from `ColMajorMatrix`. Note that the initialisation is supplied with data from the original representation (cf. `source.getRows(),...`). As explained in section 2.2.3, transposing a matrix in the `RowMajorMatrix` representations, yields a matrix in the `ColMajorMatrix` representation. Thus, in order to represent the same matrix as before, the newly created matrix is transposed (line 8).

Listing 4.5: The JIT-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.getArray());
8      target.transpose();
9    }
10
11   ColMajorMatrix to RowMajorMatrix {
12     target(source.getCols(),
13            source.getRows(),
14            source.getArray());
15     target.transpose();
16   }
17 }
```

File. In the definition of the JIT class `File` (listing 4.6), there are three *named transition functions* defined. `myFile.close()` causes a `File` in the open state to transition to the closed stated. In other words, calling the parameterless method `close` incurs a representation change from `OpenFile` to `ClosedFile`.

From the file example in listing 4.6 it is possible to construct a finite state graph with the representations as vertices and the transition functions as edges between those representations. We called such a graph the *transition graph* in chapter 3. The transition graph of `File` (see figure 4.2) shows that it is not possible to transition from a locked file to any other representation.

Listing 4.6: The class `File` combines three representations.

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

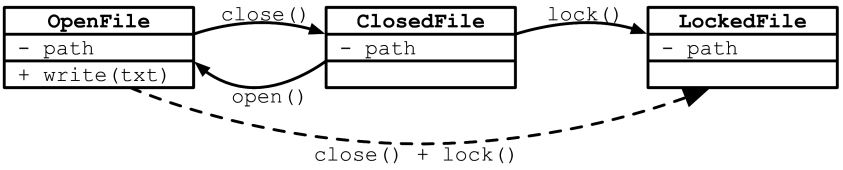


Figure 4.2: The states of a `File`: **O**pen, **C**losed, **L**ocked. (Image adapted from [78])

4.3.2 Swap Statement

The *swap statement* is JITds's primitive for expressing an explicit representation change. Syntactically, a swap statement consists of a *subject* and a *target representation* $\texttt{R}_{target}$ combined with the infix keyword `to`. The *subject* can be any expression from the base language that evaluates to an object. The *target representation* is restricted to the name of a class. Because a swap statement only makes sense for instances of a just-in-time class, the *subject* has to be an instance of a just-in-time class. Moreover, it only makes sense to swap between the representations combined by this just-in-time class. Thus, the *target representation* has to be one of the representations in this just-in-time class. Both restrictions are formally discussed in chapter 5, where we present the type system of JITds.

If the current representation of a just-in-time object is $\texttt{R}_i$, then the swap to $\texttt{R}_i$ is a null-operation by definition, i. e., nothing happens. Otherwise, a sequence of transitions functions, i. e., a *path in the the transition graph*, has to be executed, which effectively changes the representation of the subject to the requested representation. When no such path can be found an `UnsupportedSwapException` is thrown at runtime.

Syntax: Swap Statement

```
subject to R_target;
```

We now present two examples of how to use the swap statement. In the first example we change the representation of a `Matrix`, and in the second example we change the representation of a `File`.

Matrix. In section 2.2.3 we showed that a matrix multiplication, when implemented using 3 nested loops, yields better performance when the first operand is stored in row-major order and the second operand is stored in column-major order. Listing 4.7 puts this theory into practice, as it changes the first operand `a` to the `RowMajorMatrix` representation, and the second operand `b` to the the `ColMajorMatrix` representation. Lines 5 and 6 of listing 4.7 show how this is realised by means of a *swap statement*.

File. Listing 4.8 shows how create and use a just-in-time `File`. The code in listing 4.8 does not use any swap statements explicitly. On line 1, a new `File` is created in the

Listing 4.7: Using `Matrix`.

```
1  int N = 1024;
2  Matrix mA= new Matrix(N,N);
3  Matrix mB = new Matrix(N,N);
4
5  mA to RowMajorMatrix;
6  mB to ColMajorMatrix;
7
8  Matrix mC = mul(mA, mB);
```

`ClosedFile` representation. The statements on lines 3 and 5 invoke a named transition. Thus, executing lines 3 and 5 has the exact same effect as if the respective statements `file to OpenFile` and `file to ClosedFile` would have been used.

Listing 4.8: Using `File`.

```
1  File myFile= new File.ClosedFile("~/docs/out.txt");
2
3  myFile.open(); // myFile to OpenFile
4  myFile.write("lorem␣ipsum");
5  myFile.close(); // myFile to ClosedFile
```

4.4 Member fields

Syntax: Member field declaration

```
class T combines R0, ... , Rn {
  Ti vni; ...
  static Ts vns; ...
  ...
}
```

JITds allows the declaration of member fields in a just-in-time class, just like in regular classes. The value of a regular field, defined by a type and a name, can differ per just-in-time instance. The value of `static` fields belong to the just-in-time class and is unique for that class (cf. Java). Both instance fields and static fields that are defined in the just-in-time class, remain unchanged when a representation changes. The need for static fields and instance fields becomes clear in the next chapter, when we discuss the scoping rules of swap rules.

4.5 Intended Usage

> *When a programmer makes use of an abstract data object, he is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation. —*
> Liskov and Zilles [53]

When Liskov and Zilles [53] proposed "programming with *abstract data types*", they had a clear notion of *behavioural subtyping* in mind, i. e., "Liskov substitutability". Liskov substitutability states that, regardless of implementation, all objects of the same abstract data type behave identically. Note that this requirement is not enforced by the language but rather remains the responsibility of the developer. In JITds, we make a similar assumption. We assume that the set of representations of a single just-in-time class are Liskov substitutable. Thus, under the assumption of Liskov substitutable representations, **the programmer does not have to be concerned with which representation that is actively used at any point in time**. This follows directly from Liskov's substitution principle, because an object should behave the same, regardless of the current representation.

However, the intended usage of just-in-time data structures diverges on two fronts from the intention of Liskov and Zilles [53]:

- we allow a *specialised data representation* to have a *specialised data interface*, and
- we allow just-in-time data structures to combine non-Liskov substitutable representations to make state changes explicit (cf. typestates [3]).

We now discuss both use-cases in detail and present the language features that are related. We first discuss *specialised representations*, then we discuss *state representations*.

4.5.1 Specialised Representations

Until now, we presented just-in-time objects as objects that are, when instructed, able to transition between the representations of a just-in-time class. Our running example combines the classes `RowMajorMatrix` and `ColMajorMatrix` into one just-in-time class `Matrix`. Both `RowMajorMatrix` and `ColMajorMatrix` implement the exact same data interface and are also behaviourally identical. Hitherto, we did not pay much attention to these properties because they align with Liskov's and Zilles's [53] view on abstract data types and data interfaces.[1] Moreover, in Liskov and Zilles [53] view "identical interfaces" would be a prerequisite of being combinable into a just-in-time class. We now make the case for why JITds does *not* require the representations of a JIT class to implement the exact same data interface.

A new representation for an abstract data type is often developed to better cope with a specific situation. The matrix, for instance, can be represented by an array that stores the elements in row-major order, or it can be represented by an array that stores the elements in col-major order. When the number of (non-zero) elements in the matrix is low,

[1] In [53] data interface is referred to as the set of "characterising operations for a type".

however, a third storage scheme could outperform the two aforementioned representations in both access time as well as memory footprint. Listing 4.9 introduces such a third representation in the form of the class `SparseMatrix`. A full implementation of the compressed row storage scheme (CRS) is omitted here, but can be found in appendix C.

Listing 4.9: The class `SparseMatrix`.

```
1  class SparseMatrix  {
2
3    int rows, cols;
4
5    int   getRows()  { return rows; }
6    int   getCols()  { return cols; }
7
8    int       get(int row, int col) { ... }
9    void      set(int row, int col, int val) { ... }
10   Iterator nonZeroIterator(){ ... }
11   int       nonZeroCount(){ ... }
12 }
```

The interface of `SparseMatrix` is roughly the same as the interface of `RowMajorMatrix` and `ColMajorMatrix` (cf. listings 4.1 and 4.2). All three representations provide an implementation for `getRows()`, `getCols()`, `get(int, int)`, and `set(int, int, double)`. However, there is one method that is specific to `SparseMatrix`, i. e., `nonZeroIterator()`. `nonZeroIterator()` is expected to return an iterator over all non-zero values, i. e., functionality that only makes sense depending on the sparsity of the matrix. For `SparseMatrix` it is interesting to provide such functionality because it is tightly coupled with its intended properties.

While the `nonZeroIterator()` operation makes sense in the context of `SparseMatrix`, it is unnecessary to provide such functionality in the context of a "dense" representation such as `RowMajorMatrix` or `ColMajorMatrix`. The two dense representations, on the other hand, provide an operation to access the raw data array, i. e., `getArray()`. Conversely, the `getArray`-operation does not make sense in the context of `SparseMatrix`.

If we define a JIT class `Matrix` that combines all three representations (listing 4.10) and we require that all representations implement the exact same data interface, then we put a burden on the developer to "fill the holes". This burden easily results in stub implementations as illustrated in listing 4.11. Conversely, JITds explicitly allows developers to combine representation classes that do not necessarily implement the exact same data interface.

Listing 4.10: The class `Matrix` combines three representations.

```
1  class Matrix combines RowMajorMatrix, ColMajorMatrix, SparseMatrix {
2    ...
3  }
```

Listing 4.11: The method stubs `nonZeroIterator()` and `nonZeroCount()` are introduced to fill the holes in a dense matrix representation to comply with the data interface of `SparseMatrix`.

```
1  Iterator nonZeroIterator(){
2    // Should not be called
3    throw new UnsupportedOperationException();
4  }
5
6  int      nonZeroCount(){
7    // Should not be called
8    throw new UnsupportedOperationException();
9  }
```

		RowMajorMatrix	ColMajorMatrix	SparseMatrix
core	`getRows()`	✓	✓	✓
core	`getCols()`	✓	✓	✓
core	`get(int row, int col)`	✓	✓	✓
core	`set(int row, int col, double val)`	✓	✓	✓
specialised	`getArray()`	✓	✓	✗
specialised	`transpose()`	✓	✓	✗
specialised	`nonZeroIterator()`	✗	✗	✓
specialised	`nonZeroCount()`	✗	✗	✓

Table 4.1: The methods of `Matrix` and their occurrence in the three representations classes.

Thus, in JITds it is possible to combine optimised implementations, which provide a specialised data interface, into a single just-in-time class. As a corollary we can differentiate between two types of operations, based on their occurrence in the different representation classes. The *core operations*, or *core* for short, are the set of operations implemented in all representations. *Specialised operations*, on the other hand, are only implemented in a subset of the representations. Table 4.1 summarises which operations are a member of `Matrix`'s core, and which are not.

Conclusion: The first intended usage of just-in-time classes is to combine *behaviourally identical representations*. This implies that representations should implement the exact same data interface.
To support specialisation, however, *specialised operations* are allowed, such that *specialised representations* can be implemented. **The rule of thumb, however, is to maximise the *core* and to minimise the number of specialised operations.**

4.5.1.1 Specialised Swaps

In the previous paragraph we have defined the core as the set of operations that are implemented by all representations of a just-in-time class. Thus, invoking a core operation on a just-in-time object never poses any problems because all representation classes have an implementation for it, regardless of the current representation. The semantics of JITds simply states that the corresponding operation from the current representation has to be executed. When a specialised operation is invoked, however, it is not guaranteed that the intended operation has an implementation in the current representation. We identify two possibilities. Either the current representation does provide an implementation, then the same semantics apply as if it were a core operation. Alternatively, if the current representation does not provide an implementation, JITds is free to automatically change the representation of the just-in-time object to a different representation which does provide an implementation. We refer to this implicit representation change as a *specialised swap*.

Example. Consider the code fragment in listing 4.12, which relies on `nonZeroIterator` to count the number of non-zero elements. Because `SparseMatrix` is the only representation of `Matrix` that foresees the method `nonZeroIterator`, JITds guarantees that right before the execution of the method (line 3), the representation of `m` is changed to `SparseMatrix`.

The rationale behind the specialised swap is to ensure that a just-in-time object is in the correct representation at the moment a method is invoked. To this end, JITds 1. searches a path p from the current representation to *any* representation in which the invoked method has in implementation[2], 2. executes the transition functions that correspond to the edges in p, 3. (re-)invokes the method. Because JITds executes the transi-

[2] Assume, for now, that the transition graph is fully connected.

tion functions needed to transition to a representation with an implementation for the requested method, step 3 is guaranteed to succeed.

When no such path exists, this either means the transition is not intended to happen (cf. `LockedFile` in `File`, figure 4.2), or the developer forgot to implement a specialised transition function. Chapter 5 shows how a static analysis of the code can inform the developer of either of these cases, e. g., by means of a compiler warning or a type error.

Alternatively, it is possible that multiple paths exist. This happens when multiple representations exist with an implementation for the specialised method. Since representations of the same just-in-time class are intended to be behaviourally identical, it is irrelevant, at least with respect to the behaviour of the program, which of the paths is chosen. For performance, however, the choice might be very relevant. The current JITds specification (see chapter 5) and implementations (see chapter 7) gives precedence to the first representation in the list of representations.

Listing 4.12: 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.nonZeroIterator();
4    while(it.hasNext()) {
5      it.next();
6      count++;
7    }
8    return count;
9  }
```

4.5.1.2 The Performance Pitfall of Unintended Specialised Swaps

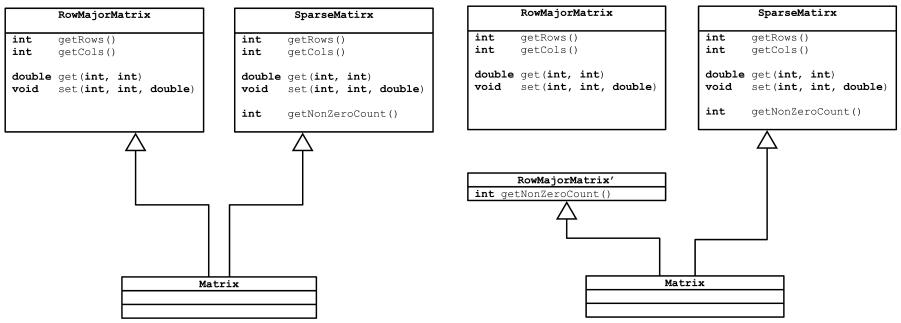
In 2014, we conducted an empirical study of the use of the Java Fork/Join framework [22]. One of the conclusions of this work was that the majority of developers write inefficient Fork/Join code because they either lack sufficient knowledge about the framework (cf. section 2.1), or because it is "easier" to write inefficient code than to write efficient code (cf. "path of least resistance"). We foresee a similar performance pitfall in JITds, where it is easy to write a program with a lot of unintended specialised swaps, which might hamper the performance of an application. Programmers that are not aware of the potential cost of calling specialised operations can be tempted to invoke them whether appropriate or not. This is because the *user* of a just-in-time class does not necessarily know whether he is using a core method or a specialised method. Especially since IDE's with autocompletion are commonplace today , the developer that acts as a mere user of a just-in-time data structure can be oblivious to the representation changes that using specialised method potentially incurs. In section 4.5.1.3 we present a solution to avoid unwanted specialised swaps, in the intermezzo on page 74 we briefly sketch the findings of De Wael et al. [22].

Anecdotical Intermezzo:
The majority of developers follow the "path of least resistance".

In an empirical study of the use of the Java Fork/Join framework, we observed that the majority of developers follow the "path of least resistance" when implementing functional requirements [22]. The study considers (all) 120 GitHub projects that make use of the Java Fork/Join framework and reveals that in 20%, 13%, and 15% of the programs that use the framework, the respective anti-patterns *heavyweight split*, *heavyweight merge*, and *inappropriate sharing* are present (see [22] for a description of these anti-patterns). These are high numbers if one knows that the heavyweight split and heavyweight merge anti-pattern can only occur in certain restricted scenarios (i. e., map and fold operations on collections).

```
1  public List buildRange(int n) {
2     return buildRange(0,n);
3  }
4
5  public List buildRange(int from, int to) {
6     if ( (to-from) < 1 ) {
7        return Collections.emptyList();
8     } else if ( (to-from) == 1 ) {
9        return Collections.singletonList(from);
10    } else {
11       int mid = (from+to)/2;
12
13       List result = new ArrayList();
14
15       result.addAll( buildRange(from, mid) );
16       result.addAll( buildRange(mid , to ) );
17
18       return result;
19    }
20 }
```

The code fragment above builds a `List` of integers using the divide-and-conquer approach. In the non-base case (lines 10-19), the readily available operation `addAll` on `List` is used to combine two partial results. First, note that within the execution of `buildRange(n)`, the operation `addAll` is executed $O(\log(n))$ times. Second, note that, in Java, `addAll` is an expensive operation on `ArrayList` because one invocation can trigger up to two calls to `System.arraycopy`. When `buildRange` is called with a large n, this implementation can be considered to be an anti-pattern.
De Wael et al. [22] conclude that heavyweight merge is a common pitfall for programmers using the Java Fork/Join framework, because the current collections framework does not inform programmers about the high cost incurred by recursively executing expensive operations.



(a) A call to `getNonZeroCount()` causes a specialised swap because `getNonZeroCount()` is a specialised method of `SparseMatrix`.

(b) A call to `getNonZeroCount()` does not cause a specialised swap because `getNonZeroCount()` is bypassed.

Figure 4.3: The just-in-time class `Matrix` inherits methods from both `RowMajorMatrix` and `SparseMatrix`. `getNonZeroCount()` is a specialised method because it is only implemented by `SparseMatrix`.

4.5.1.3 Bypass Methods

In order to bypass expensive or frequent representation changes caused by unintended specialised swaps, we introduce a new type of class member in just-in-time classes: the *bypass method*. The bypass method is a construct adopted from Ureche et al. [84], who propose a similar solution to avoid conversions from stack values to heap values. Concretely, a bypass method provides a generic implementation of a specialised operation which is executed en lieu of a potentially costly representation change.

Syntax: Bypass Methods

```
class T combines R0, ... , Rn {
  public X Ri.methodNamea(){ ... }
  public X T.methodNameb(){ ... }
}
```

Bypass methods are implemented as if they were regular methods. Thus, they appear as members in the definition of a just-in-time class. The syntactic difference between bypass methods and overridden methods is that the name of a bypass method is qualified with the name of the representation for which it acts as a bypass. Alternatively, when the qualifier is the name of the just-in-time class itself, then the bypass method acts as a bypass for all specialised swaps. Conceptually, a bypass method extends one (or more) of the representation classes, without altering the original base class.

Listing 4.13: A call to `getNonZeroCount()` does not cause a specialised swap because `getNonZeroCount()` is bypassed.

```
1  class Matrix combines RowMajorMatrix, SparseMatrix {
2
3    public int RowMajorMatrix.getNonZeroCount() {
4      int count = 0;
5      for (int r=0 ; r<this.getRows() ; r++)
6        for (int c=0 ; c<this.getCols() ; c++)
7          if ( this.get(r,c) == 0.0 )
8            count++;
9      return count;
10   }
11
12 }
```

Matrix Example. The `getNonZeroCount` operation is one of the specialised methods of the `SparseMatrix` representation. This is shown in figure 4.3a. When this method is invoked on an object that has a dense representation as current representation, that object changes its representation to `SparseMatrix` in order to be able to execute the method invocation. However, this representation change can be *unwanted and expensive*, for instance when the matrix has a lot of non-zero values. Listing 4.13 shows (a part of) the implementation of `Matrix` which provides an bypass method for the `getNonZeroCount` operation. Figure 4.3a shows conceptually how this is realised in JITds.

4.5.2 Functional Swaps

In section 4.5.1 we discussed *specialisation* as a first use-case where we want to combine representations with different data interfaces. A second scenario where we would want to combine representations with different data interfaces is when the different representations model *complementary behaviour*. In such a scenario, it makes sense for an object to support data interface `A` at one point in time and data interface `B` at a different point in time. In general, this scenario where the *role* of an object changes during its lifetime is explored in typestate-oriented programming [3]. The `File` example, which we introduced in section 4.2.1, is one of the running examples from typestate-oriented programming. We now discuss this file example in the context of *specialisation*.

Consider the three classes from listing 4.14 as the representation classes of the just-in-time class `File` from listing 4.6. When we look at the three data interfaces, we observe that `write` is a specialised method only available for files in the "open" state. Recall that listing 4.6 introduces three named transition functions: `open`, `close`, and `lock`. Those named transition functions change the representation of a file f from `ClosedFile` to `OpenFile`, from `OpenFile` to `ClosedFile`, and `ClosedFile` to `LockedFile`, respectively.

The representation of a just-in-time Data object can, as introduced in section 4.5.1.1, also change due to a specialised swap. `OpenFile` is the only representation of `File`

Listing 4.14: Three implementations for a file, that represent the closed, locked, and open state, respectivly.

```
class ClosedFile {
  String getPath() { ... }
}

class LockedFile {
  String getPath() { ... }
}

class OpenFile {
  String getPath() { ... }
  void   write(String str) { ... }
}
```

which provides an implementation for `write()`. Hence, invoking the `write` operation on a `File` in the "closed" state implicitly opens the file by changing the representation to `OpenFile`. Note that, because of the shape of the transition graph, an implicit state change is not possible for files in the "locked" state, as there is no transition possible from `LockedFile` to `OpenFile`. Further note that this approach is more dynamic than the approach used in languages with support for typestate-oriented programming because code that potentially changes the state of an object does not need to be explicitly annotated as such. Plaid and Fickle$_{II}$, for instance, both require explicit annotations in the presence of a representation change [78, 27].

Conclusion: The second intended usage of just-in-time classes is to combine *behaviourally complementary representations*. To support dynamic typestate-oriented programming, the different states are implemented as different representations. Invoking state specific operations causes a dynamic change in representation (cf. specialised swap). **Again, the rule of thumb when combining representations is to maximise the *core* and to minimise the number of specialised operations.**

4.5.2.1 Transient Information

This section discusses an issue that can occur when programming with behaviourally complementary representations. When two representations `A` and `B` are behaviourally identical, then all information present in `A` is also present in `B`, and vice versa. In a class-based object-oriented language, this implies that the fields of `A` and `B` encode the same information, but store the data differently. Consequently, a representation change from `A` to `B` can change the data representation, but preserves information.

However, when two representations `A` and `B` are behaviourally complementarily, it is still possible that `A` and `B` independently store extra information. Consequently, a representation change from `A` to `B`, or vice versa, can lose information.

Listing 4.15: A `Bat` is both a `Mammal` and a `FlyingAnimal`.

```
1  class Mammal       {
2    int feedingCount;
3
4    void feedOffspring() { feedingCount++; }
5    int getFeedingCount() { return feedingCount; }
6  }
7
8  class FlyingAnimal {
9    void fly() { ... }
10 }
11
12 class Bat combines FlyingAnimal, Mammal { ... }
```

Listing 4.16: Using methods of both representations.

```
1  Bat dracula = new Bat.Mammal();
2  dracula.feedOffspring();                // Mammal
3  int before = dracula.getFeedingCount(); // Mammal
4  dracula.fly();                          // FlyingAnimal
5  int after  = dracula.getFeedingCount(); // Mammal
6  assert(before==after);
```

We now present a new just-in-time class `Bat` which combines `Mammal` and `FlyingAnimal` (see listing 4.15). Consider `dracula` (see listing 4.16) as an instance of `Bat`. The call to `feedOffspring()` implies that `dracula` adheres to the `Mammal` representation, and results in the field `feedingCount` being incremented by one. Similarly, to execute the call to `getFeedingCount()`, `dracula` adheres to the `Mammal` representation. Just before the next feeding moment (line 5), `dracula` flies around, causing a representation change to `FlyingAnimal`. The second call to `getFeedingCount()` again causes a representation change, this time to `Mammal`. Conceptually, one would expect the assertion on line 6 to hold. However, because the state is transient in the context of a representation change, this is not guaranteed in JITds.

In a JITds program, the transition functions between `Mammal` and `FlyingAnimal` are responsible for *transforming* the data from the source representation into the format (i. e., available fields) of the target representation. The `Bat` example reveals that a transition between behaviourally complementary representations can result in the unintended loss of information (cf. `feedingCount`). To avoid the unintended loss of information, JITds foresees the possibility to annotate fields as `persistent`.

Syntax: Persistent Fields

```
class T combines R0, ... , Rn {
  persistent Ri.f;
  ...
}
```

To declare a field as persistent for a given just-in-time class `T`, it suffices to add a persistent declaration to the declaration of a field `f` in the definition of `T`. The syntax of such a declaration is the keyword `persistent` followed by the name of the field quantified with the name of the representation it occurs in, for instance $\texttt{R}_i\texttt{.f}$. The semantics of a representation change in the context of a persistent field are as follows: When a just-in-time object `obj` abandons the $\texttt{R}_i$ representation, the values of each persistent field $\texttt{f}_i$ are evacuated from their original slots and temporarily stored elsewhere, i. e., memorised. When `obj` transitions back to the $\texttt{R}_i$ representation, directly or indirectly, then the memorised values for $\texttt{f}_i$ are assigned back to their original slots. Both the "evacuation" and the "'reinstallation" of the values of persistent fields happens *before* the execution of the actual transition function. This allows the transition functions to make use of these values or to correct unintended behaviour. Listing 4.17 shows an implementation of `Bat` where the field `feedingCount` from the `Mammal` representation is made persistent. With this implementation of `Bat`, the assertion on line 6 in listing 4.16 succeeds.

Listing 4.17: The field `Mammal.feedingCount` is made persistent in `Bat`.

```
1 class Bat combines FlyingAnimal, Mammal {
2   persistent Mammal.feedingCount;
3 }
```

4.6 Managing the Number of Transition Functions

When a JIT class combines N representations, then there are $O(N^2)$ different transition functions that can be implemented.[3] Thus, when the number of representations grows, the number of possible transition functions grows fast. Writing that many transition functions becomes impractical. This is an obvious critique to JITds in theory. We argue, however, that the combinatorial explosion in the number of transition functions is not an issue in practice.

- 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 general-purpose implementations for the `List` interface (i. e., `ArrayList`, `Linked-`

[3] $N(N-1)$ to be exact

Listing 4.18: 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  }
```

`List`, and `Vector`).[4] In such a case, 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 *generic transition function* for the `Matrix` example is shown in listing 4.18. Such a generic transition function can replace all other *specialised transition functions*. Of course, from a performance perspective, specialised transition functions are likely to be preferred. An example of a specialised transition function in the matrix example is the `transpose` function which expresses the transition from a `RowMajorMatrix` to a `ColMajorMatrix`, and vice versa. These specialised transition functions are shown in listing 4.5 (lines 3–8 and 10–15).

- Third, the set of available specialised 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 specialised and direct transition function is likely to be preferred in terms of performance.

- A final argument to counter the "transition function explosion" is that some transitions between two representations are unlikely to occur, or semantically even nonsensical. Implementing a specialised transition function in such cases serves no practical purpose. For instance, we do not want to allow transitions from `LockedFile` to any other representation, because conceptually a `LockedFile` is ought to remain locked. Hence, omitting transition functions from the `LockedFile` representation to any other representation, encodes this conceptual constraint.

[4] Other "known implementing classes" of `List` include `AbstractList`, `AbstractSequentialList`, `AttributeList`, `CopyOnWriteArrayList`, `RoleList`, `RoleUnresolvedList`, `Stack`. These are either abstract classes (i. e., `AbstractList` and `AbstractSequentialList`), list specialised for certain objects (i. e., `AttributeList`, `RoleList`, `RoleUnresolvedList`), or are lists with complimentary behaviour (i. e., `Stack` which provides a stack interface or `CopyOnWriteArrayList` which provides a thread-safe implementation that "may be more efficient when traversal operations vastly outnumber mutations").

4.7 Just-in-Time Data Structures put into Context

From the previous sections we know that JITds is designed as a statically typed, class-based object-oriented language. The core idea of JITds is that objects can change their representation (i. e., class) at runtime. To realise such representation changes, JITds allows a class to `combine` multiple other classes into one. In this section we put both concepts in context: First, we discuss how JITds's representations changes are a special case of *dynamic object reclassification*. Second, we discuss the relation between combine multiple classes into one and *multiple inheritance*. Finally, we discuss how JITds is positioned in the landscape of object-oriented programming languages in general, using the terminology introduced by Lieberman et al. [52].

4.7.1 Homomorphic Reclassification

When an object changes its representation in JITds, this effectively means that its current class is changed. Changing the class of an object at runtime is known as *dynamic object reclassification* [27]. This term, however, is very general. For instance, changing the `__class__` field in Python is a discouraged practice — from a software engineering point of view — but it is still classified as reclassification. Other, more restricted forms of reclassification exist. Cohen and Gil [13], for instance, study a form of reclassification where objects can gain properties but never lose properties. They refer to this restricted form of reclassification as *monotonic reclassification*.

A just-in-time object always support the union of all methods of all of its representations. Thus, also in JITds an object never *loses* properties. Hence, JITds's representation changes are a form of monotonic reclassification. Moreover, we see that an object in JITds never *gains* properties either. In other words, the data interface of a just-in-time object never changes. This is because specialised swaps ensure[5] that a just-in-time object is always able to invoke any method of any representation. What is described above, is thus an even more restricted from of reclassification than *monotonic reclassification*, which we will call *homomorphic reclassification*.

> *Homomorphic reclassification* is a restricted form of dynamic object reclassification where changing the representation of an data object does not change the data interface of the object. In other words, homomorphic reclassification does not add nor remove properties from a data object.

4.7.1.1 Two Implementation Techniques.

From the work on monotonic reclassification we infer that there are two base approaches to implement dynamic object reclassification: *inline* reclassification or reclassification by *forwarding*. Other techniques are either variations of the latter two or require more

[5] The guarantee only holds when the appropriate transition functions exist.

advanced language features (e. g., skake-ins and mix-ins) [13]. In section 7.3 and section 7.4 we discuss two implementations of JITds that use reclassification by forwarding and inline reclassification, respectively.

Reclassification by forwarding uses a handle to point to an object with the current representation. When the object is reclassified, it suffices to update the handle to point to a freshly allocated object, with the new representation. This fresh object can reside anywhere in memory. This technique is used in our JITds to Java compiler and is discussed in section 7.3.

Inline reclassification on the other hand does not require a handle but transparently stores an object at a fixed memory location. To support reclassification, however, enough memory has to be allocated to fit the *largest possible representation*. This technique is used in our JITds to C compiler and is discussed in section 7.4.

4.7.2 Multiple Inheritance

A just-in-time class combines multiple representation classes into one, inheriting methods from all representations. In this paragraph we discuss the relation between this approach and *multiple inheritance*.

Consider a just-in-time class `T`, which combines two representations `A` and `B`. We already explained how an instance of `T` is able to respond to the specialised methods of both `A` and `B` (see section 4.5.1.1). In our matrix example, for instance, listing 4.19 is a sound program fragment. Thus, a just-in-time class effectively inherits members of more than one class. This is known as *multiple inheritance* and is a widely studied feature in the literature on object-oriented programming languages. However, multiple inheritance is also a widely criticised feature.

One of the critiques of multiple inheritance is that when multiple parents want to pass on a member with the same signature to its shared offspring, it is ambiguous which of the parents wins the evolutionary fight. The ambiguity problem in multiple inheritance, often referred to as the *diamond problem*, has been tackled in various ways. Appendix D identifies *Ambiguity Rejection*, *Protocols Linearisation*, *Select and Rename*, *Explicit Inheritance*, *Method Combination*, *Most Specific Argument*, *Meta-object protocol*, and *Virtual Inheritance* as the conventional techniques to tackle the diamond problem.

Listing 4.19: Code fragment with two specialised swaps.

```
1  Matrix m = new Matrix.RowMajorMatrix(3,4);
2  m.getRows();                // Core Method
3  m.nonZeroIterator();        // Specialized in SparseMatrix
4  m.getRows();                // Core Method
5  m.getDataArray();           // Specialized in Row- and ColMajorMatrix
6  m.getRows();                // Core Method
```

In JITds it is unambiguous which method (i. e., of which representation) will be executed at-runtime, because a just-in-time object always has a current representation. Consider the core of any just-in-time class. This set of operations has a different implementation in each of the representations. Generally speaking, in languages with multiple inheritance it is unclear which of these implementations to execute. Various programming languages try to resolve the ambiguity in various ways (see appendix D). In JITds,

however, there is *no ambiguity at runtime*, because a just-in-time object simply executes the implementation found in its *current representation*. For example, a `Matrix m` can be in one of three representations, i. e., `RowMajorMatrix`, `ColMajorMatrix`, or `SparseMatrix`. And a call to the core method `getRows()`, can potentially be executed by one of these three different implementations. The specific invocation of `getRows()` on line 4 in listing 4.19 is guaranteed to execute the implementation of `SparseMatrix`, because we know `m.nonZeroIterator()` (line 3) forces `m` to adhere to the `SparseMatrix` representation (specialised swap).

We observe that the inheritance mechanism of the language JITds relies purely on dynamic information to decide which branch of the static inheritance hierarchy to use. According to the survey of ambiguity mitigating techniques, JITds is the first language which inherently resolves ambiguity based on run-time information. Note that we by no means claim that JITds solves the problems emerging from multiple inheritance in general. We merely claim that JITds is a unique programming language when it comes to the dynamic semantics of lookups in the hierarchy chain, especially since JITds is a statically typed programming language.

The reason why JITds is the first language to do this is because the intent of inheriting from multiple representations is different in JITds compared to other languages. In general, "`T extends A, B`" implies that an instance `t` of `T` is **both** an `A` and a `B` at any point in time. In JITds, however, "`T combines A, B`" implies that an instance `t` of `T` can be **either** an `A` or a `B` at any point in time[6].

Furthermore, compared to other programming languages with support for representation changes (e. g., Gilgul, Fickle$_{II}$ or Plaid), JITds is the only language that treats the representations as a super type of the changeable data structure. Concretely, in the matrix example this means that in JITds we consider a just-in-time Matrix to be a subtype of `RowMajorMatrix`, `ColMajorMatrix`, and `SparseMatrix`, i. e., multiple inheritance. Whereas the same example in Gilgul, Fickle$_{II}$ or Plaid, can only be realised where `RowMajorMatrix`, `ColMajorMatrix`, and `SparseMatrix` are *states* (i. e., some kind of subtype) of the general type matrix. The latter approach is fine when the representations are never to be used as types (cf. implementation only classes in Gilgul) or when the current representation is statically known (cf. Fickle$_{II}$ and Plaid). In JITds, on the other hand, representations can be used as stand-alone types. Furthermore, the just-in-time class introduces a new representation, i. e., a representation that can benefit from all representations it combines. Finally, the current representation of a just-in-time object is only known at runtime. Hence, to model that the same just-in-time matrix object can behave both as an `RowMajorMatrix` as well as an `SparseMatrix`, the type relation a just-in-time class and its representations is to be modelled as a sub-type relation (instead of an super-type relation as in e. g., Gilgul, Fickle$_{II}$ or Plaid).

We compare the types of the matrix classes for an implementation in a classic object-oriented language without multiple inheritance (here Java) JITds. This is visualised in figure 4.4. The interface `IMatrix` collects all methods shared by all representations of

[6] This statement seems to be false at first for the `Bat` example: a `Bat` is *conceptually* both a `Mammal` and a `FlyingAnimal`. From the *programming language's* point of view, however, an instance of `Bat` is always either in the `Mammal` or in the `FlyingAnimal` representation.

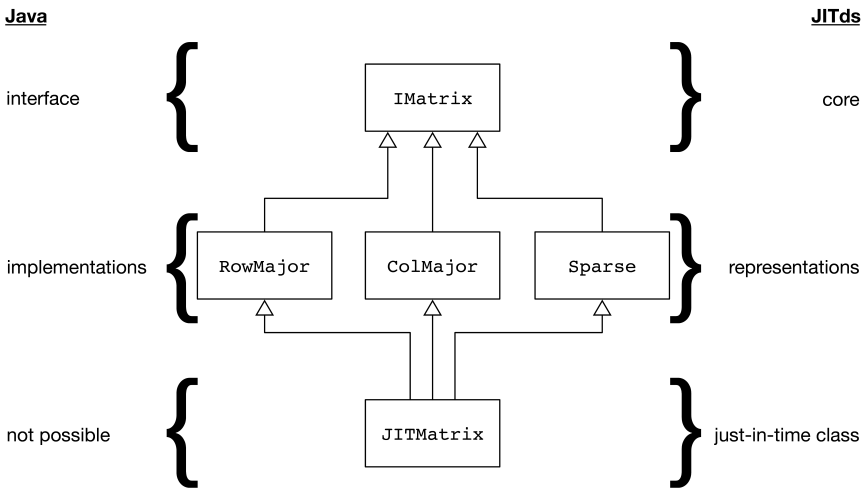


Figure 4.4: The types of matrix considered from both sides.

a matrix (e. g., `getRows` or `set`). Note that in JITds, we called this set of methods that are shared by all representation the *core*. All representations *implement* the `IMatrix` (Java) interface. Potentially, a single representation has some *specialised behaviour*, e. g., `nonZeroIterator` in `SparseMatrix`. In JITds, as opposed to Java, it is possible to combine all these representations into a *new typeadvantage*. This has to be a new type because it supports the behaviour of the core (for which super type would suffice), but it supports all the specialised behaviour as well, for which only the sub-type suffices. To realise the same behaviour in Java, one would also need to define a fourth implementation of `IMatrix`, e. g., with the name `ComboMatrix`. The discussion on the types of representations and just-in-time classes is revisited and discussed more formally in chapter 5.

4.7.3 Treaty of Orlando

Lieberman et al. [52] concluded from a decade of work on object-technology that there are three dimensions in which code sharing (i. e., inheritance) in object-oriented languages is realised.

> ... the following three independent dimensions along which this sharing mechanism - which some call delegation and others inheritance ... - can be examined, namely: FIRST, whether STATIC or DYNAMIC: When does the system require that the patterns of sharing be fixed?... SECOND, whether IMPLICIT or EXPLICIT: Does the system have an operation that allows a programmer to explicitly direct the patterns of sharing between objects, or does the system do this automatically and uniformly? ... and THIRD. whether

> PER OBJECT or PER GROUP: Is behavior specified for an entire group of objects at once, ..., or can idiosyncratic behavior be attached to an individual object? — *Henry Lieberman, Lynn Stein, and David Ungar [52].*

Using this terminology we can categorise JITds as a programming language where sharing is *dynamic*, *implicit*, and *per group*. This categorisation is clear, however, we do make some side remarks.

In JITds, sharing is *dynamic* because a just-in-time object can change its representation which effectively means that the implementation that is inherited, changes after allocation time. Even though sharing is *dynamic*, the set of potential classes from which code can be inherited is *statically* known (i. e., the representations).

In JITds, the sharing mechanism is *implicit* because it is uniformly defined for all just-in-time objects that the current representation defines from which representation class the invoked method is inherited. Also the mechanism of specialised swaps is uniformly defined for all just-in-time objects. A swap statement can be used to *explicitly* select the representation from which to inherit the code.

Just-in-time classes define for all there instances, i. e., *per group*, from which representation classes code can be inherited. A swap statement can be used to select the representation from which to inherit the code *per instance*.

4.8 Conclusion

In JITds, a just-in-time class is defined by combining multiple representations. Instances of such a class can change their representation during the execution of a program without losing their identity.

Orderly transitions between the representations are realised by transition functions, a special kind of member of a just-in-time class. The transition logic is encapsulated and thus clearly separated from the application logic.

Invoking a swap statement causes an actual representation change. These statements can be used anywhere in the code. Thus, in general, representation change incentive code is entangled with the application logic.

Chapter 5

JIT$\Delta\sigma$: A Formal Specification of JITds

In this chapter we present JIT$\Delta\sigma$, a formal semantics for JITds with a focus on the *type system*. We also present a type system for JITds and prove that this type system is sound. A sound type systems gives us the guarantee that when a JITds program is well-formed (i. e., type checked) it will never encounter type errors at runtime, concretely we can guarantee that invoking a method never results in a "method not found" exception and that accessing a field never results in a "field does not exist" exception.

In chapter 4 we motivate why JITds is designed as an extension of Java. Hence, the formalism we present in this chapter is designed as an extension of a formalism of Java. Concretely, JIT$\Delta\sigma$ is largely based on Lightweight Java [75, Chapter 3] and [76], a minimal imperative core calculus of Java that is still a proper subset of Java.

In general, the idea of a core calculus is to reason over a complex language, by defining a subset of that language in order to ignore irrelevant details [8, 46, 76, 62]. Igarashi et al. [46] say it is sensible to design a formal model that favours compactness over completeness, because such a model offers maximum insight for minimum investment. Lightweight Java, for instance, does not model type casts, local variables, field hiding, interfaces, method overloading, explicit constructors, literals, operators, or any of the more advanced language features of Java. Hence, these features are not modelled in JIT$\Delta\sigma$ either.

For the sake of compactness, we further simplify the model of Lightweight Java and also exclude *public field access* (i. e., all fields are private) and *single inheritance* from JIT$\Delta\sigma$. Excluding public field access has no significant impact on the completeness of JIT$\Delta\sigma$ because it can be implemented by providing simple getters and setters for (each) field. Ignoring single inheritance in the formal specification of a Java subset, however, is less obvious. This simplification allows us to focus on the peculiarities that stem from JITds's representation changes, as opposed to the peculiarities that stem from inheritance. In section 5.6, we discuss the changes needed to JIT$\Delta\sigma$ in order to support single inheritance as well.

In the remaining sections we show the user syntax of JIT$\Delta\sigma$ (section 5.1), the operational semantics of JIT$\Delta\sigma$ (section 5.2), the type system (section 5.4), type checking (section 5.4.4), and a proof of type soundness (section 5.5). In section 5.6, we sketch the

Listing 5.1: User Syntax of JITds.

```
P          : cd*

cd         : class Cn { fd* md* }
           | class Cn_jit combines Cn_rep* { td* }

fd         : Cn fn;

md         : Cn_ret mn ( pd* ) { stmt* return x; }
pd         : Cn vn

td         : Cn_src to Cn_tar { stmt* }

stmt       : { stmt* }
           | vn = x ;
           | vn = this . fn ;
           | this . fn = y ;
           | if ( x==y ) stmt_t else stmt_f
           | vn = new Cn() ;
           | vn = x . mn(a*) ;

           | vn = new Cn_s.Cn_d() ;
           | x to Cn_tar ;
           | vn = source . fn ;
           | target . fn = x ;

x, y, a    : vn
           | this
```

impact on JIT$\Delta\sigma$ if single inheritance where to be included. Finally, in section 5.7 we summarise the results presented in this chapter.

5.1 User Syntax

This section describes the user syntax of JIT$\Delta\sigma$, which is shown in listing 5.1. We adhere to the following conventions: keywords are coloured, declarations end with a "d", and names end with an "n". Sequences are represented with a bar over the elements. All names (i. e., `Cn`, `fn`, `mn`, and `vn`) range over valid Java identifiers. Finally, subscripts indicate indexes.

Program In JIT$\Delta\sigma$, a program P is a sequence of class definitions $\overline{\texttt{cd}}$. A class definition cd is either a *simple class* or a *just-in-time class.*

Simple class A simple class defines a class with a name Cn and a list of field and method definitions, i. e., $\overline{\texttt{fd}}$ and $\overline{\texttt{md}}$, respectively.

A field definition fd is simply a combination of a class name Cn and a field name fn. Similarly, a parameter definition pd is a combination of a class name Cn and a variable name vn. Here, a class name defines the static type of a field or a parameter, respectively.

A method definition md, combines the return type in the form of a class name Cn with a method name mn, a list of parameter definitions $\overline{\texttt{pd}}$, and a method body. The method body, between curly braces, combines in turn a list of statements $\overline{\texttt{stmt}}$ and a return statement.

Just-in-time class A just-in-time class defines a class with a name $\texttt{Cn}_{jit}$ and a list of names of representation classes $\overline{\texttt{Cn}_{rep}}$. While it is not enforced in the specification of the syntax, the number of representations must be at least two. Furthermore, a just-in-time class definition contains a list of transition function definitions $\overline{\texttt{td}}$. A transition function definition td combines the class name of the source representation $\texttt{Cn}_{src}$ and the class name of the target representation $\texttt{Cn}_{tar}$ with a sequence of statements $\overline{\texttt{stmt}}$ that describe the actual transition.

Statements The syntax for statements stmt forms a language on its own. Up to the four last statements, the syntax of JITds statements are a proper subset of Java's syntax for statements. We refrain from discussing these in detail and only enumerate the statements specific for JITds. To assign a fresh just-in-time object to the variable vn the keyword new is followed by the name of a just-in-time class $\texttt{Cn}_s$ and the class name of the initial representation $\texttt{Cn}_d$. Just like Lightweight Java, we do not model user defined constructors. Hence, creating a new object does not require any arguments. The swap statement orders a variable x to change its representation to the class Cn. The last two statements are adaptations of field read and field write to be used in the body of a transition function. This restriction cannot syntactically be enforced, but is implied in the type checking phase by means of well-formedness rules (see section 5.4.4).

5.2 Operational Semantics

We define the semantics of a JIT$\Delta\sigma$ program in terms of reductions from one *configuration* to another, for instance $c \longrightarrow c'$. These reductions are described in section 5.3. However, before we can discuss these rules, we need to describe the constructs and auxiliary functions used in these rules: *configurations* (see section 5.2.1), *selection functions* (see section 5.2.2), and transition graphs (see section 5.2.3).

5.2.1 Configurations, Heaps, Stacks, Objects, and Values

In JIT$\Delta\sigma$ a configuration c is a triplet (S, H, P) consisting of a stack S, a heap H, and — as discussed in section 5.1 — a program P. A heap H is a mapping between object identifiers *objId* and actual objects. A stack S is a, possibly empty, list of frames, where the leftmost frame is the *top frame* or *current frame*. Each frame contains the information needed to continue a computation: i. e., a *sequence of statements* $\overline{\texttt{stmt}}$, a *local store* L, and an *invocation context* I. $\overline{\texttt{stmt}}$ is a list of statements that remain to be executed. A local store L is a mapping from variables `x` to values v (or w). An invocation context I reveals whether the ongoing computation — top of stack — is executed in the context of a method invocation or in the context of a transition function. In the former case, the invocation context is a pair containing the method name `mn` and the defining class `Cn` ($I = (\texttt{Cn}, \texttt{mn})$). In the latter case, the invocation context is a triplet containing the just-in-time class $\texttt{Cn}_{jit}$, the source representation $\texttt{Cn}_{src}$, and the target representation $\texttt{Cn}_{tar}$ ($I = (\texttt{Cn}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar})$).

The top stack frame can, instead of statements, also contain an *exception* as the first element. JIT$\Delta\sigma$ inherits one type of exception from Lightweight Java, i. e., the null pointer exception `NPE`; and introduces a new type of exception, i. e., the unsupported swap exception `USE`.

Values There are only two kinds of values modelled in JIT$\Delta\sigma$: a value v is either **`null`** or it is a an object identifier *objId*.

Objects In JIT$\Delta\sigma$ an object is modelled as a triplet $(\texttt{Cn}_s, \texttt{Cn}_d, F)$. $\texttt{Cn}_s$ and $\texttt{Cn}_d$ respectively denote the *static class* and the *dynamic class* of the object. The *static class* is the class of which the object is an instance, the *dynamic class* is the current representation. The discussion on the difference between static/dynamic *class* and static/dynamic *type*, is deferred to section 5.4. F is a mapping between field names `fn` and values v. In section 5.4 we discuss in greater detail the resemblances and the differences between types and class (names) in JIT$\Delta\sigma$. For just-in-time objects the static class is the name of a just-in-time class and the dynamic class is the class name of the current representation. For a simple object $obj = (\texttt{Cn}_s, \texttt{Cn}_d, F)$ $\texttt{Cn}_s$ and $\texttt{Cn}_d$ are the same, i. e., the name of the simple class of which *obj* is an instance.

Mappings H, L, and F model a heap, a local store, or a field binding, respectively. Listing 5.2 shows the production rules for updating and extending these functions, e. g., to add a new variable to a local store. Furthermore, listing 5.2 gives an overview of the syntax of the constructs introduced in this section.

5.2.2 Selection Functions

Selection functions are auxiliary functions used to select a specific piece of information from a construct such as, for instance, statements or a *configuration*. The find-class(P, `Cn`) function, which tries to find a class definition `cd` for a given class name `Cn` in the program P. This function is a simple linear search trough the sequence of class definitions

Listing 5.2: Syntax of JIT$\Delta\sigma$ to describe the state of a JITds program.

$$
\begin{array}{lll}
\textit{config} & : & (S, H, P) \\
\\
H & : & [\ ] \\
 & | & H[\textit{objId} \mapsto \textit{obj}] \\
\\
S & : & \varepsilon \\
 & | & \langle \overline{\texttt{stmt}}, L, I \rangle \ : \ \texttt{S} \\
 & | & \langle \textit{Exception}, L, I \rangle \ : \ \texttt{S} \\
\\
I & : & (\texttt{Cn},\texttt{mn}) \\
 & | & (\texttt{Cn}_{jit},\texttt{Cn}_{src},\texttt{Cn}_{tar}) \\
\\
L & : & [\ ] \\
 & | & L[\texttt{x} \mapsto v] \\
\\
\textit{obj} & : & (\texttt{Cn}_s,\texttt{Cn}_d,F) \\
\\
F & : & [\ ] \\
 & | & F[\texttt{fn} \mapsto v] \\
\\
v, w & : & \textbf{null} \\
 & | & \textit{objId} \\
\\
\textit{Exception} & : & \textbf{NPE} \\
 & | & \textbf{USE}
\end{array}
$$

found in P. Most selection functions are equally trivial, mostly because we do not consider single inheritance.[1] We therefore omit from this discussion, most of the auxiliary functions that are used to inspect the program state. These omitted functions can be found, however, in appendix B.

5.2.3 Transition Graph

In JITds, the representations of a just-in-time class form a directed graph, which we use to model the relationships between representations of a just-in-time class, and we use the graph's paths to describe a sequence of transitions needed to go from one representation to another. We call such a graph a *transition graph*. For each just-in-time class `Cn`, we can construct its transition graph $\mathcal{G}_{cn} = (V, E)$. The vertices V of $\mathcal{G}_{cn}$ are the representations of `Cn`. Further, the set of edges E can be constructed from the set of

[1] Finding the class inheritance path is the only source of complexity in Lightweight Java's "lookup functions" [76].

defined transition functions $\overline{\texttt{td}}$. Iff $\texttt{Cn}_{from}$ **to** $\texttt{Cn}_{to}$ {...}$\in \overline{\texttt{td}}$ then $(\texttt{Cn}_{from}, \texttt{Cn}_{to}) \in E$ (cf. adjacent). These relations are given by TRANS-GRAPH in figure 5.2.

To avoid ambiguity later in this section, we first introduce an *ordering on paths* (see figure 5.1) and we define an ordering on the representations of a just-in-time class. We define the latter to be the precedence in the list of representations, i. e., $\texttt{Cn}_i < \texttt{Cn}_j$ iff $\texttt{Cn}_i$ precedes $\texttt{Cn}_j$ in the list of representations $\overline{\texttt{Cn}}$. The former is defined in figure 5.1. Informally, a shorter path is always "smaller than" a longer path; for equally long paths the ordering of the representations is used as a tie breaker. This relation is a *total order* (proof omitted).

$$\frac{\mathsf{length}(p) < \mathsf{length}(p')}{p \leq_{\mathcal{G}} p'} \leq\text{-LENGTH}$$

$$\frac{}{[\,] \leq_{\mathcal{G}} [\,]} \leq\text{-EQUAL-EMPTY}$$

$$\frac{a < a' \qquad \mathsf{length}(rest) = \mathsf{length}(rest')}{[a \mid rest] \leq_{\mathcal{G}} [a' \mid rest']} \leq\text{-COMBINATION}$$

$$\frac{a = a' \qquad \mathsf{length}(rest) = \mathsf{length}(rest') \qquad rest \leq_{\mathcal{G}} rest'}{[a \mid rest] \leq_{\mathcal{G}} [a' \mid rest']} \leq\text{-COMBINATION-2}$$

Figure 5.1: The ordering $\leq_{\mathcal{G}}$ on paths in $\mathcal{G}$

In chapter 4, we describe how transition functions can be used transitively. Hence, all legal transition sequences can be modelled as a path p in $\mathcal{G}_{cn}$. We call such a path a *transition path*. We need these transition paths in two cases, either in the context of a swap statement to express a representation change from one representation $\texttt{Cn}_{src}$ to another $\texttt{Cn}_{tar}$, or alternatively, in the context of a specialised swap when the current representation $\texttt{Cn}_d$ does not have an implementation for the method mn, and a transition to a representation that does have an implementation for mn needs to be found and transitioned to.

For the first case, TRANS-PATH-CLASS finds the shortest transition path between $\texttt{Cn}_{src}$ and $\texttt{Cn}_{tar}$ in $\mathcal{G}_{cn}$. For the second case, TRANS-PATH-METHOD finds the shortest transition path between $\texttt{Cn}_d$, the current representation, and another representation of the just-in-time class $\texttt{Cn}_s$ in which there exists an implementation for mn. These rules are described in figure 5.2.

Note that, in practice, the shortest path is not necessarily the fastest path. For instance, it is possible that executing the transitions from A to B to C takes longer than executing the transitions A to X to Y to Z to C. Currently, it is not possible to express such information in JITds. To tackle this problem, JITds could ask the developer to mark each transition function with an estimated cost. Then JIT$\Delta\sigma$ needs a weighted graph to model

the transition graph, and the shortest path then needs to take these costs into account. This extension, however, is future work.

$$\frac{\mathscr{G}_{cn} = (V, E) \qquad V = \overline{\texttt{Cn}_{rep}} \qquad \overline{\texttt{td}} = \overline{\texttt{Cn}_{from}\ \texttt{to}\ \texttt{Cn}_{to}\ \{...\}} \qquad E = \overline{(\texttt{Cn}_{from}, \texttt{Cn}_{to})}}{\text{transition-graph}(\texttt{class}\ \texttt{Cn}_{jit}\ \texttt{combines}\ \overline{\texttt{Cn}_{rep}}\{\ \overline{\texttt{td}}\ \}) = \mathscr{G}_{cn}}\ \textsc{Trans-Graph}$$

$$\frac{\text{find-class}(P, \texttt{Cn}) = \texttt{cd} \qquad \text{transition-graph}(\texttt{cd}) = \mathscr{G}_{cn} \qquad \overline{p} = \text{path}(\mathscr{G}_{cn}, \texttt{Cn}_{src}, \texttt{Cn}_{tar}) \qquad p = \min_{\leq_{\mathscr{G}}}(\overline{p})}{\text{transition-path}(P, \texttt{Cn}, \texttt{Cn}_{src}, \texttt{Cn}_{tar}) = p}\ \textsc{Trans-Path-Class}$$

$$\frac{\begin{array}{c}\text{find-class}(P, \texttt{Cn}_s) = \texttt{cd} \qquad \text{transition-graph}(\texttt{cd}) = \mathscr{G} \qquad \text{class-representation-names}(\texttt{cd}) = \overline{\texttt{Cn}_{rep}} \\ \overline{\texttt{Cn}'_r} = \{\texttt{Cn}_r \mid \texttt{Cn}_r \in \overline{\texttt{Cn}_{rep}} \wedge \text{find-method}(P, \texttt{Cn}_r, \texttt{mn})\} \\ \overline{p} = \left\{p \mid \exists \texttt{Cn}'_r \in \overline{\texttt{Cn}'_r} : p \in \text{path}(\mathscr{G}, \texttt{Cn}_d, \texttt{Cn}'_r)\right\} \qquad p = \min_{\leq_{\mathscr{G}}}(\overline{p})\end{array}}{\text{transition-path}(P, \texttt{Cn}_s, \texttt{Cn}_d, \texttt{mn}) = p}\ \textsc{Trans-Path-Method}$$

Figure 5.2: Finding a transition path in a transition graph.

5.3 Reduction Semantics for Statements

Reductions rules in JIT$\Delta\sigma$ are of the form $c \longrightarrow c'$. In section 5.2.1, we defined a configuration c to be the triple (S, H, P). For a reduction, the first statement found in the top frame of the stack is often the most important because that statement defines which reduction can be applied. Hence, our reduction rules make those explicit by expanding the top stack frame, and the first statement, e. g., $\left(\left\langle \texttt{stmt}_1\ \overline{\texttt{stmt}}, L, I\right\rangle, H, P\right)$.

Most reduction rules — i. e., block statements, variable assignment, and if statements — in JIT$\Delta\sigma$ are fairly straightforward adaptations of the reduction rules for statements in Lightweight Java. We refrain from discussing them in detail but they are shown for the sake of completeness in figure 5.3. The same holds for the reduction rules for field access (i. e., read and write), which are shown in figure 5.4. Note, however, that field access can cause a null pointer exception when the variable `x` dereferences to the value **`null`**. The reductions for object creation, method invocation, and swap statements are less trivial and are discussed separately in sections 5.3.1 to 5.3.3, respectively.

$$\frac{}{(\langle\{\overline{\texttt{stmt}_b}\}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P) \longrightarrow (\langle\overline{\texttt{stmt}_b}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P)}\ \text{R-BLOCK}$$

$$\frac{L(\texttt{x}) = v \qquad L' = L[\texttt{vn} \mapsto v]}{(\langle\texttt{vn = x ;}\overline{\texttt{stmt}}, L, I\rangle : S, H, P) \longrightarrow (\langle\overline{\texttt{stmt}}, L', I\rangle : S, H, P)}\ \text{R-VARIABLE-ASSIGN}$$

$$\frac{L(\texttt{x}) = v \qquad L(\texttt{y}) = w \qquad v = w}{(\langle\texttt{if (x==y) stmt}_t\ \texttt{else stmt}_f\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P) \longrightarrow (\langle\texttt{stmt}_t\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P)}\ \text{R-IF-T}$$

$$\frac{L(\texttt{x}) = v \qquad L(\texttt{y}) = w \qquad v \neq w}{(\langle\texttt{if (x==y) stmt}_t\ \texttt{else stmt}_f\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P) \longrightarrow (\langle\texttt{stmt}_f\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P)}\ \text{R-IF-F}$$

Figure 5.3: Reduction rules for block statements, variables assignment, and if statements.

$$\frac{L(x) = objId \qquad H(objId) = (\texttt{Cn}_s, \texttt{Cn}_d, F) \qquad F(\texttt{fn}) = v \qquad L' = L[\texttt{vn} \mapsto v]}{(\langle\texttt{vn = x.fn;}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P) \longrightarrow (\langle\overline{\texttt{stmt}}, L', I\rangle : S, H, P)}\ \text{R-FIELD-READ}$$

$$\frac{L(x) = \textbf{null}}{(\langle\texttt{vn = x.fn;}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P) \longrightarrow (\langle\textbf{NPE}, L, I\rangle : S, H, P)}\ \text{R-FIELD-READ-NPE}$$

$$\frac{L(x) = objId \qquad H(objId) = (\texttt{Cn}_s, \texttt{Cn}_d, F) \qquad L(y) = v \qquad F' = F[\texttt{fn} \mapsto v] \qquad H' = H[objId \mapsto (\texttt{Cn}_s, \texttt{Cn}_d, F')]}{(\langle\texttt{x.fn = y;}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P) \longrightarrow (\langle\overline{\texttt{stmt}}, L, I\rangle : S, H', P)}\ \text{R-FIELD-WRITE}$$

$$\frac{L(x) = \textbf{null}}{(\langle\texttt{x.fn = y;}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P) \longrightarrow (\langle\textbf{NPE}, L, I\rangle : S, H, P)}\ \text{R-FIELD-WRITE-NPE}$$

Figure 5.4: Reduction rules for field reads and field writes.

5.3.1 Reduction Object Construction

We already introduced the triplet representation of objects in JIT$\Delta\sigma$. Creating an object has as net effect that the heap is extended with a new entry, i. e., a mapping from a fresh

identifier *objId* to such a triplet $(H\left[objId \mapsto (\texttt{Cn}_s, \texttt{Cn}_d, F)\right])$. We discuss the rule R-NEW-JIT, which creates a new just-in-time object, in detail, and show how the rule R-NEW-SIMPLE, which creates a plain old object, is a simplification of the former. These rules are shown in figure 5.5.

The statement `vn = new` $\texttt{Cn}_s$`.`$\texttt{Cn}_d$`();` denotes that the variable `vn` has to be bound to a newly created just-in-time object. This new object is an instance of the just-in-time class $\texttt{Cn}_s$ and has $\texttt{Cn}_d$ as its initial representation. Note that only the representation is able to change at runtime, hence the subscript d which is a shorthand notation for "dynamic". The just-in-time class, on the other hand, is fixed, hence the subscript s for "static".

The first steps in the object creation rule R-NEW-JIT are to look up the class definition of the initial representation $\texttt{Cn}_d$, and to extract the field names $\overline{\texttt{fn}}$ from this class definition. To create a new unambiguous entry in the heap, a fresh *objId* is created and a new entry with that identifier is added to the original heap. As stated in section 5.2.1, the object is a triplet containing static class, dynamic class, and a sequence of field-value mappings. For a just-in-time object the static class and dynamic class are respectively the just-in-time class $\texttt{Cn}_s$ and the initial representation $\texttt{Cn}_d$. Because we do not model explicit constructors, all fields are initially mapped to the value **`null`**. Finally, the local store L is extended with a binding of `vn` with the new object identifier *objId*. The computation can then proceed with reducing the statements that are still left, i. e., $\overline{\texttt{stmt}}$. These further reductions, however, happen in the context of the updated heap H' and updated local store L'.

The creation of a new simple object (i. e., an instance of a simple class), as described by R-NEW-SIMPLE, is analogous to R-NEW-JIT up to the classes of the new object. In a just-in-time object the static class and the dynamic class always refer to two distinct classes. In a simple object the static class and the dynamic class are the same, namely the class being instantiated. This difference is thus only observable in the rules, where the object's triplet is created and added to the heap.

$$\frac{\textsf{find-class}(P, \texttt{Cn}_d) = \texttt{cd}_d \qquad \textsf{class-field-names}(\texttt{cd}_d) = \overline{\texttt{fn}} \qquad F = \overline{[\texttt{fn} \mapsto \textbf{null}]} \qquad objId \notin \textsf{dom}(H) \qquad H' = H[objId \mapsto (\texttt{Cn}_s, \texttt{Cn}_d, F)] \qquad L' = L[\texttt{vn} \mapsto objId]}{\left(\langle \texttt{vn = new } \texttt{Cn}_s\texttt{.}\texttt{Cn}_d\texttt{();} \ \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow \left(\langle \overline{\texttt{stmt}}, L', I\rangle : S, H', P\right)} \text{ R-NEW-JIT}$$

$$\frac{\textsf{find-class}(P, \texttt{Cn}) = \texttt{cd} \qquad \textsf{class-field-names}(\texttt{cd}) = \overline{\texttt{fn}} \qquad F = \overline{[\texttt{fn} \mapsto \textbf{null}]} \qquad objId \notin \textsf{dom}(H) \qquad H' = H[objId \mapsto (\texttt{Cn}, \texttt{Cn}, F)] \qquad L' = L[\texttt{vn} \mapsto objId]}{\left(\langle \texttt{vn = new } \texttt{Cn}\texttt{();} \ \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow \left(\langle \overline{\texttt{stmt}}, L', I\rangle : S, H', P\right)} \text{ R-NEW-SIMPLE}$$

Figure 5.5: Reduction rules for creating new objects.

5.3.2 Reduction Method Invocation and Method Return

Method Invocation The method invocation statement yields the most interesting reduction rules (shown in figure 5.6) as they prove to differ the most from the rules generally found in Java like semantics. The first rule, however, is not. R-MI-NPE simply covers the case where a method is invoked on the value **`null`**, i. e., when `x` is **`null`**.

The second rule R-MI-DIRECT formalises the case where a method is actually found and thus can be directly executed. To extract the body of a method, first the correct method definition `md` has to be found: 1. Look up the object identifier in the local store, and 2. track down the identified object in the heap, 3. find the definition of the dynamic class of the object, and 4. search for the method with the name `mn` . Then, when all previous steps succeed, the rule shows how to prepare a new local store L_m in which to execute the method's body. This local store contains a binding from **`this`** to the object identifier of the receiver and a binding for each parameter name with the corresponding argument's value. Together the method body $\overline{\texttt{stmt}_B}$, the local store L_m, and a new invocation context I', form a new frame that is pushed on top of the old stack S'. Note that S' is the full stack as it found in the original configuration, i. e., S preceded by the stack frame containing the method invocation. This implies that the method invocation statement is still present in the top frame of S'. We keep the method invocation statement because later, wen we return from the method invocation, we need the know the variable name `vn`, to which the result of the invocation has to be bound (cf. R-MR-DIRECT).

The static type checker (see section 5.4) prevents the invocation of methods on simple objects whose class does not provide a method definition with the right name. For just-in-time objects, however, it is possible that the dynamic class, i. e., the current representation, does not provide the required method definition. In section 4.5.1.1 we discuss how JITds then changes the representation of the just-in-time object to a representation that does have an implementation of the requested method. Section 5.2.3 discusses how to compute a transition path, which in its simplest form is the shortest sequence of transitions needed to reach such a representation (TRANS-PATH-METHOD). The resulting transition path is a sequence of intermediate representations $\overline{\texttt{Cn}_{tar}}$. Transitively changing the representation of `x` to the representations in $\overline{\texttt{Cn}_{tar}}$ causes `x` to end up in a representation with an implementation for `mn`. These transitions can be realised by a sequence of swap statements in JITds, i. e., $\overline{\texttt{x to Cn}_{tar}\texttt{;}}$. Hence, the reduction in R-MI-INDIRECT replaces the original method invocation statement by a block statement $\texttt{stmt}_t = \{\overline{\texttt{x to Cn}_{tar}\texttt{;}}\,\texttt{vn = x.mn(}\overline{\texttt{a}}\texttt{);}\}$, which starts with a sequence of swap statements and ends with the original method invocation statement. If no transition path can be found, as is modelled by R-MI-USE, then the reduction for the method invocation ends in an exception configuration because the required representation changes are unsupported (unsupported swap exception **`USE`**).

$$\frac{L(x) = \texttt{null}}{\left(\langle \texttt{vn = x.mn(}\overline{\texttt{a}}\texttt{)} \ ; \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow \left(\langle \textbf{NPE}, L, I\rangle : S, H, P\right)} \text{ R-MI-NPE}$$

$$\frac{\begin{array}{c} L(x) = \mathit{objId} \qquad H(\mathit{objId}) = (\texttt{Cn}_s, \texttt{Cn}_d, F) \qquad \text{find-class}(P, \texttt{Cn}_d) = \texttt{cd}_d \\ \text{find-method}(\texttt{cd}_d, \texttt{mn}) = \texttt{md} \qquad \texttt{md} = \texttt{Cn}_{ret}\ \texttt{mn}\ (\overline{\texttt{pd}})\ \ \{\overline{\texttt{stmt}_B}\ \textbf{return}\ \texttt{x;}\ \} \\ \overline{\text{parameter-name}(\texttt{pd}) = \texttt{pn}} \qquad \overline{L(a) = v} \qquad L_m = [\textbf{this} \mapsto \mathit{objId}][\overline{\texttt{pn} \mapsto v}] \\ S' = \langle \texttt{vn = x.mn(}\overline{\texttt{a}}\texttt{)} \ ; \overline{\texttt{stmt}}, L, I\rangle : S \qquad I' = (\texttt{Cn}_d, \texttt{mn}) \end{array}}{\left(\langle \texttt{vn = x.mn(}\overline{\texttt{a}}\texttt{)} \ ; \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow \left(\langle \overline{\texttt{stmt}_B}\textbf{return}\ \texttt{x;}, L_m, I'\rangle : S', H, P\right)} \text{ R-MI-DIRECT}$$

$$\frac{\begin{array}{c} L(x) = \mathit{objId} \qquad H(\mathit{objId}) = (\texttt{Cn}_s, \texttt{Cn}_d, F) \qquad \text{find-class}(P, \texttt{Cn}_d) = \texttt{cd}_d \\ \neg\text{find-method}(\texttt{cd}_d, \texttt{mn}) \qquad \text{transition-path}(P, \texttt{Cn}_s, \texttt{Cn}_d, \texttt{mn}) = \overline{\texttt{Cn}_{tar}} \\ \texttt{stmt}_t = \{\overline{\texttt{x}\ \textbf{to}\ \texttt{Cn}_{tar}\texttt{;}}\texttt{vn = x.mn(}\overline{\texttt{a}}\texttt{)} \ \texttt{;}\} \end{array}}{\left(\langle \texttt{vn = x.mn(}\overline{\texttt{a}}\texttt{)} \ ; \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow \left(\langle \texttt{stmt}_t\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right)} \text{ R-MI-INDIRECT}$$

$$\frac{\begin{array}{c} L(x) = \mathit{objId} \qquad H(\mathit{objId}) = (\texttt{Cn}_s, \texttt{Cn}_d, F) \qquad \text{find-class}(P, \texttt{Cn}_d) = \texttt{cd}_d \\ \neg\text{find-method}(\texttt{cd}_d, \texttt{mn}) \qquad \neg\text{transition-path}(P, \texttt{Cn}_s, \texttt{Cn}_d, \texttt{mn}) \end{array}}{\left(\langle \texttt{vn = x.mn(}\overline{\texttt{a}}\texttt{)} \ ; \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow \left(\langle \textbf{USE}, L, I\rangle : S, H, P\right)} \text{ R-MI-USE}$$

Figure 5.6: Reduction rules method invocation.

Method Return As discussed above, the reduction of a valid method invocation pushes a new frame on top of the stack and continues the execution of a method's body, with a new local store and a method invocation context. By construction, the last statement of a method's body is a return statement, cf. the first statement in the reductions shown in figure 5.7. When such a return statement is observed as first statement in the top frame, we know we have reached the end of a method invocation. R-MR-DIRECT describes how to reduce such a configuration. The top frame of S, thus the second frame on the current stack, reveals that the result of the method invocation is to be bound to the variable `vn`, and that $\overline{\texttt{stmt}}$ is the remaining computation after the method invocation. The value of `y` is to be returned, hence, the computation should be continued in a local store where `vn` is bound the value of `y`. This is realised in the construction of L'. Thus, the reduction continues with the execution of $\overline{\texttt{stmt}}$ in the updated local store and the stack $\langle \overline{\texttt{stmt}}, L', I'\rangle : S'$.

This reduction, however, is only allowed when it can be guaranteed that the caller in the calling context (i. e., either another method or an transition function) still has the same dynamic class as at the time of the current method's invocation. Consider, for instance, the method `f` defined in the class `R0` as shown in listing 5.3. On line 6, the field `a` is accessed. In JITds, and any other Java-like programming language, the type system can guarantee that this field exists and holds a value of the expected type. On line 7, a the

method `foo` is invoked. On line 8, the field `a` is accessed again. In any other Java-like programming language, this is trivially possible as well. In JITds, however, we need to consider the possibility that the caller of `foo`, here an instance of `R0`, has undergone a representation change.

Listing 5.3: Method `f` in class `R0`, that accesses the field `a` before and after a call to `foo`.

```
1  class R0 {
2
3    A a;
4
5    void f(B b, C c) {
6      this.a = new A();
7      c = b.foo();
8      this.a = new A();
9    }
10
11 }
```

This is possible when the caller is not actually an instance of R_0, but rather an instance of `T` which was in the R_0 representation at the moment of invoking `foo`. If the caller has changed its representation to R_1 during the execution of `foo`, it is not longer possible to access the field `a` after the invocation, because R_1 does not have a field `a`. Accessing a non-existing field at runtime is a type error, hence to avoid this kind of errors we have to ensure that a caller does not change its representation or if it does, that it changes it representation back to the original representation.

Listing 5.4: Just-in-time class `T` combines R_0 and R_1. R_1 does not have a field `a` of type `A`.

```
1  class R1 { }
2  class T combines R0, R1 { ... }
```

The auxiliary function caller-has-expected-type (shown in figure 5.8) checks whether the original caller still has the expected dynamic class. When this is not the case, an appropriate swap statement is inserted before the return statement as is modelled by R-MR-INDIRECT.

$$\frac{S = \langle \texttt{vn = x.mn(a); } \overline{\texttt{stmt}}, L_c, I_c \rangle : S' \qquad \text{caller-has-expected-type}(L_c, I_c, H) \qquad L(\texttt{y}) = w \qquad L' = L_c[\texttt{vn} \mapsto w] \qquad I' = I_c}{(\langle \textbf{return } \texttt{y;}, L, I \rangle : S, H, P) \longrightarrow \left(\langle \overline{\texttt{stmt}}, L', I' \rangle : S', H, P\right)} \text{ R-MR-DIRECT}$$

$$\frac{S = \langle \texttt{vn = x.mn(a); } \overline{\texttt{stmt}}, L_c, I_c \rangle : S' \qquad \neg\text{caller-has-expected-type}(L_c, I_c, H) \qquad \text{caller}(I_c) = c \qquad \text{expected-caller-type}(I_c) = \texttt{Cn} \qquad \texttt{x}_c \notin \text{dom}(L) \qquad L' = L[x_c \mapsto objId_c]}{(\langle \textbf{return } \texttt{y;}, L, I \rangle : S, H, P) \longrightarrow \left(\langle \texttt{x}_c \textbf{ to } \texttt{Cn; } \textbf{return } \texttt{y;}, L', I \rangle : S, H, P\right)} \text{ R-MR-INDIRECT}$$

Figure 5.7: Reduction rules of method return.

$$\frac{}{\text{caller}((\texttt{Cn}, \texttt{mn})) = \textbf{this}} \text{ caller-method}$$

$$\frac{}{\text{caller}\left((\texttt{Cn}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar})\right) = \textbf{source}} \text{ caller-tf}$$

$$\frac{}{\text{expected-caller-type}((\texttt{Cn}, \texttt{mn})) = \texttt{Cn}} \text{ expected-caller-type-method}$$

$$\frac{}{\text{expected-caller-type}\left((\texttt{Cn}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar})\right) = \texttt{Cn}_{src}} \text{ expected-caller-type-tf}$$

$$\frac{\text{caller}(I) = c \qquad \text{expected-caller-type}(I) = \texttt{Cn}_d \qquad L(c) = objId \qquad H(objId) = (\texttt{Cn}_s, \texttt{Cn}_d, F)}{\text{caller-has-expected-type}(L, I, H)} \text{ caller-has-expected-type}$$

Figure 5.8: Auxiliary Functions.

5.3.3 Reduction Swap Statement and Transition Function Return

Swap Statement The final statement for which we present reduction rules is the swap statement. `x to Cn;`, for instance, changes the representation of a just-in-time object `x` to `Cn`. Whether `x` is actually a just-in-time object, and whether `Cn` is a valid representation for `x` is checked by the type system (see section 5.4). All other cases that can occur at runtime are formalised in the rules found in figure 5.9.

The first swap reduction rule, R-SWAP-NPE, handles the case where the value of `x` is **`null`**. In that case, the reduction yields a configuration with a null pointer exception

(**NPE**). The second swap reduction rule, R-SWAP-USE, models the case where the intended transition, i. e., from the dynamic class of x to the new representation $\texttt{Cn}_{tar}$, is not possible. This happens when there is no direct transition function defined, or no transition path exists. In that case, the reduction yields a configuration with an unsupported swap exception (**USE**). When no *direct* transition function is defined, but there exists a transition path $\overline{\texttt{Cn}_t}$, the transition to $\texttt{Cn}_{tar}$ can be realised by a sequence of swap statements $\overline{\texttt{x to Cn}_{tar}\texttt{;}}$, as shown by R-SWAP-INDIRECT.

R-SWAP-IDEMPOTENT and R-SWAP-DIRECT both describe the reduction of a swap statement that fully succeeds. R-SWAP-IDEMPOTENT, on the one hand, describes the reduction of a swap statement where no representation change is needed because the target representation is the current representation $\texttt{Cn}_d$, i. e., the dynamic class remains unchanged.

R-SWAP-DIRECT, on the other hand, describes the reduction of a swap statement that does require an actual representation change and for which a transition function exists. To extract the body of a transition function, first the correct transition function definition td has to be found: 1. Look up the object identifier for x in the local storage, and 2. track down the identified object in the heap, 3. find the definition of the static class of the object, and 4. search for the transition function with from $\texttt{Cn}_{src}$ to $\texttt{Cn}_{tar}$. Then, when all previous steps succeed, the rule stipulates to prepare a new local store L' in which to execute the transition function's body. In L', **source** refers to the original object and **target** refers to a newly created object in H'. The transition function body $\overline{\texttt{stmt}'}$, the local store L', and a new invocation context I', form a new frame that is pushed on top op the old stack S'.

Transition Function Return As discussed above, executing a swap statement pushes a new frame on top of the stack and continues the execution of a transition function's body, with a new local store and an transition function invocation context. When the sequence of statements in the top frame of the stack is empty ($\emptyset$), we know we have reached the end of a transition function invocation.[2] When such an empty sequence is observed R-TFR-DIRECT describes how to remove the top frame of the stack and continue with the remainder of the computation. But first, the original object referred to by $L_c(\texttt{x})$ has to be replaced by the new object referred to by $L(\textbf{target})$. Hence, the updated heap H'.

Returning from a transition function is, similar to returning from a method invocation (see earlier example), only allowed when it can be guaranteed that the caller in the calling context (i. e., either another transition function or a method) still has the same dynamic class as at the time of the current transition function's invocation. This is checked by the auxiliary function caller-has-expected-type, shown in figure 5.8. The latter can not be guaranteed when the caller of the transition function has undergone a representation change. In that case, as modelled by R-TFR-INDIRECT, an appropriate swap statement is inserted before the return statement.

[2] The alternative, the end of a method invocation, ends by construction with a return statement.

$$\frac{L(x) = \textbf{null}}{\left(\langle \texttt{x to Cn}_{tar}\texttt{;}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow (\langle \textbf{NPE}, L, I\rangle : S, H, P)}\ \text{R-SWAP-NPE}$$

$$\frac{\begin{array}{c} L(x) = objId \qquad H(objId) = (\texttt{Cn}_{jit}, \texttt{Cn}_{src}, F_{src}) \\ \text{find-class}(P, \texttt{Cn}_{jit}) = \texttt{cd}_{jit} \qquad \neg\text{find-transition-function}(\texttt{cd}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar}) \\ \neg\text{transition-path}(\texttt{cd}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar}) \end{array}}{\left(\langle \texttt{x to Cn}_{tar}\texttt{;}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow (\langle \textbf{USE}, L, I\rangle : S, H, P)}\ \text{R-SWAP-USE}$$

$$\frac{L(x) = objId \qquad H(objId) = (\texttt{Cn}_{s}, \texttt{Cn}_{tar}, F)}{\left(\langle \texttt{x to Cn}_{tar}\texttt{;}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow \left(\langle \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right)}\ \text{R-SWAP-IDEMPOTENT}$$

$$\frac{\begin{array}{c} L(x) = objId \qquad H(objId) = (\texttt{Cn}_{jit}, \texttt{Cn}_{src}, F_{src}) \\ \text{find-class}(P, \texttt{Cn}_{jit}) = \texttt{cd}_{jit} \qquad \neg\text{find-transition-function}(\texttt{cd}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar}) \\ \text{transition-path}(\texttt{cd}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar}) = \overline{\texttt{Cn}'_{tar}} \\ \texttt{stmt}_t = \left\{\overline{\texttt{x to Cn}'_{tar}\texttt{;}}\right\} \end{array}}{\left(\langle \texttt{x to Cn}_{tar}\texttt{;}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow \left(\langle \texttt{stmt}_t\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right)}\ \text{R-SWAP-INDIRECT}$$

$$\frac{\begin{array}{c} L(x) = objId \qquad H(objId) = (\texttt{Cn}_{jit}, \texttt{Cn}_{src}, F_{src}) \qquad \text{find-class}(P, \texttt{Cn}_{jit}) = \texttt{cd}_{jit} \\ \text{find-transition-function}(\texttt{cd}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar}) = \texttt{td} \\ \texttt{td} = \texttt{cn}_{src}\textbf{to}\ \texttt{cn}_{tar}\,\{\overline{\texttt{stmt}'}\} \\ \text{find-class}(P, \texttt{Cn}_{tar}) = \texttt{cd}_{tar} \qquad \text{class-field-names}(\texttt{cd}_{tar}) = \overline{\texttt{fn}_{tar}} \\ objId_{tar} \notin \text{dom}(H) \qquad L' = [\textbf{source} \mapsto objId][\textbf{target} \mapsto objId_{tar}] \\ F_{tar} = \overline{[\texttt{fn}_{tar} \mapsto \textbf{null}]} \qquad H' = H[objId_{tar} \mapsto (\texttt{Cn}_{jit}, \texttt{Cn}_{tar}, F_{tar})] \\ I' = (\texttt{Cn}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar}) \qquad S' = \langle \texttt{x to Cn}_{tar}\texttt{;}\ \overline{\texttt{stmt}}, L, I\rangle : S \end{array}}{\left(\langle \texttt{x to Cn}_{tar}\texttt{;}\ \overline{\texttt{stmt}}, L, I\rangle : S, H, P\right) \longrightarrow \left(\left\langle \overline{\texttt{stmt}'}, L', I'\right\rangle : S', H', P\right)}\ \text{R-SWAP-DIRECT}$$

Figure 5.9: Reduction rules for the swap statement.

$$\frac{S = \langle \texttt{x to } \texttt{Cn}_{tar}\texttt{;}\ \overline{\texttt{stmt}}, L_c, I_c\rangle : S' \quad \neg\text{caller-has-expected-type}(L_c, I_c, H) \quad \text{caller}(I_c) = c \quad L_c(c) = objId_c \quad x_c \notin dom(L) \quad L' = L[x_c \mapsto objId_c] \quad \text{expected-caller-type}(I_c) = \texttt{Cn}}{(\langle \emptyset, L, I\rangle : S, H, P) \longrightarrow \left(\langle \texttt{x}_c \texttt{ to cn;}, L', I\rangle : S, H, P\right)}\ \text{R-TFR-INDIRECT}$$

$$\frac{S = \langle \texttt{x to } \texttt{Cn}_{tar}\texttt{;}\ \overline{\texttt{stmt}}, L_c, I_c\rangle : S' \quad \text{caller-has-expected-type}(L_c, I_c, H) \quad L_c(\texttt{x}) = objId_x \quad L(\textbf{target}) = objId_{tar} \quad H(objId_{tar}) = (\texttt{Cn}_{jit}, \texttt{Cn}_{tar}, F_{tar}) \quad H' = H[objId_x \mapsto (\texttt{Cn}_{jit}, \texttt{Cn}_{tar}, F_{tar})]}{(\langle \emptyset, L, I\rangle : S, H, P) \longrightarrow \left(\langle \overline{\texttt{stmt}}, L_c, I_c\rangle : S', H', P\right)}\ \text{R-TFR-DIRECT}$$

Figure 5.10: Reduction rules of returning from a transition function.

5.4 A Type System for JIT$\Delta\sigma$

Before executing a JITds program, it ought to be checked for static type errors. Static type errors are errors in the definition of a program that are detectable before the execution of a program and that would cause the execution to stop, e. g., accessing a field that does not exist. The *type system* of a language defines the subset of (syntactically correct) programs that will never encounter these errors. This section presents a static type system for JITds. First, however, we informally discuss the difference between static/dynamic types and static/dynamic classes.

5.4.1 Static Types and Dynamic Types

Listing 5.5: A method `foo` which returns a new JIT object.

```
Cn_ret foo( Cn_arg x )
  x = new Cn_s.Cn_d();
  return x;
}
```

In statically-typed languages, such as JITds, a difference is made between the static type and the dynamic type of a variable. The *static type* of a variable is the type which is taken into account during the static type-checking phase. In listing 5.5, for instance, the variable `x` is statically known to hold a value of type $\texttt{Cn}_{arg}$, $\texttt{Cn}_{arg}$ is the *static type* of `x`. At runtime, however, `x` can refer to any object that is a sub-type of $\texttt{C}_{arg}$. The actual type of `x` at runtime is the *dynamic type*. At line 2 in listing 5.5, for instance, `x` is assigned a new object of type $\texttt{Cn}_s$, here a just-in-time class.

In JITds, once an object is created its dynamic type never changes. Now note that the dynamic type of an object in JITds is the same as its static class. The dynamic class of an

object, however, can change. Listing 5.5 is a code fragment that allows us to explain all terms: $\texttt{Cn}_s$ is the static type of `x`, $\texttt{Cn}_d$ is the dynamic type of `x` (after line 3).

5.4.2 Types (Cn), Subtypes (Cn <: Cn′), and Valid Types ($P \vdash$ Cn)

In JITds each class definition uniquely defines a *type*. Therefore, a type is either the class name of a simple class definition, or the class name of a JIT class definition. In the remainder of this text, because type and class name can be used interchangeably, we refrain from introducing a new symbol to denote types. Furthermore, we define a *valid type* as a type `Cn` for which there exists a class definition in `P`. The rule to determine type validity in `P`, $P \vdash \texttt{Cn}$ is given in figure 5.11.

$$\frac{\textsf{find-class}(P, \texttt{Cn}) = \texttt{cd}}{P \vdash \texttt{Cn}}\ \textsf{valid-type}$$

$$\frac{P \vdash \texttt{Cn}}{P \vdash \texttt{Cn} <: \texttt{Cn}}\ \text{ST-REFLECTIVE}$$

$$\frac{\textsf{find-class}(P, \texttt{Cn}_{jit}) = \texttt{cd} \qquad \texttt{Cn}_{rep} \in \textsf{class-representation-names}(\texttt{cd}) \qquad P \vdash \texttt{Cn}_{rep}}{P \vdash \texttt{Cn}_{jit} <: \texttt{Cn}_{rep}}\ \text{ST-JIT}$$

Figure 5.11: Subtyping rules in JIT$\Delta\sigma$.

In a type system for an object-oriented programming language, the subtyping relation defines whether a value of type `Cn` can be used when a value of type $\texttt{Cn}'$ is expected. Concretely, this is allowed when `Cn` is a subtype of $\texttt{Cn}'$. $\texttt{Cn} <: \texttt{Cn}'$ denotes the subtype relation between `Cn` and $\texttt{Cn}'$. As in most type systems, the subtype relation in JITds is both reflective as well as transitive. Thus $\texttt{Cn} <: \texttt{Cn}$ always holds; and if $\texttt{Cn} <: \texttt{Cn}'$ and $\texttt{Cn}' <: \texttt{Cn}''$ hold then $\texttt{Cn} <: \texttt{Cn}''$ also holds. The actual subtyping rules are given in figure 5.11, which state that, in a given program `P`, the type of a class is a subtype of itself if it has a definition in `P` (reflective). Alternatively, the type associated with a JIT class is a subtype of all its representations.

Besides defining the subtyping relation between types, we also define the subtyping relation between *values* and types (see figure 5.12). $v <: \texttt{Cn}$ hold when the value v is a *member* of the type `Cn`. We need this relation in section 5.4.4 when we verify whether a runtime value is a member value of the type of the variable it is bound to. The *type member relation for variables* states that **`null`** is a member of any valid type in P. A non-null value *objId* is a member of `Cn` iff the static class of the object associated with *objId* is a subtype of `Cn`.

$$\frac{P \vdash \texttt{Cn}}{P, H \vdash \texttt{null} <: \texttt{Cn}}\ \text{ST-NULL}$$

$$\frac{P \vdash \texttt{Cn} \qquad H(objId) = (\texttt{Cn}_s, \texttt{Cn}_d, F) \qquad P \vdash \texttt{Cn}_s <: \texttt{Cn}}{P, H \vdash objId <: \texttt{Cn}}\ \text{ST-DYNAMIC-CLASS}$$

$$\frac{P \vdash \texttt{Cn} \qquad H(objId) = (\texttt{Cn}_s, \texttt{Cn}, F)}{P, H \vdash objId =: \texttt{Cn}}\ \text{ST-DYNAMIC-CLASS-EQUAL}$$

Figure 5.12: Subtyping rules for values in JIT$\Delta\sigma$.

5.4.3 Local Type Environment (Γ)

In the soundness proof, we rely on a *global type environment* Γ, which is a mapping from methods and transition functions to local type environments. We come back to global type environments in more detail in section 5.5. A *local type environment* Γ is a mapping between variables `x` and their static type `Cn`. In JIT$\Delta\sigma$, such a type environment is used to type check the body of a method or a transition function in a JITds program.

Listing 5.6: Syntax of JIT$\Delta\sigma$ to encode a type environment.

$$\begin{array}{lcl} \Gamma & : & [\ ] \\ & | & \Gamma[x \mapsto \texttt{Cn}] \end{array}$$

5.4.4 Type Checking

Type checking a JITds program before its execution prevents type errors during the execution. The type checking is defined in terms of well-formedness rules. Most of these rules are similar to those found in [76, Chapter 3] and are presented here only for completeness' sake. The rules that are JIT$\Delta\sigma$ specific are discussed in more detail. The auxiliary functions used in these rules can be found in appendix B. Note that a prerequisite of well-formedness of a program is that the program adheres to the syntax defined in listing 5.1, even though it is not mentioned explicitly in what follows.

5.4.4.1 Programs

A JITds program P is well-formed if all its classes have different names ($\mathsf{distinct}(\overline{\texttt{Cn}})$) and if all its classes are themselves well-formed with respect to P ($\overline{P \vdash \texttt{cd}}$).

$$\frac{P = \overline{\texttt{cd}} \qquad \overline{\text{class-name}(\texttt{cd}) = \texttt{Cn}} \qquad \text{distinct}(\overline{\texttt{Cn}}) \qquad \overline{P \vdash \texttt{cd}}}{\vdash P}\ \text{WF-PROGRAM}$$

Figure 5.13: Well-formedness rule for programs.

5.4.4.2 Simple Classes

$$\frac{\overline{\text{field-name}(\texttt{fd}) = \texttt{fn}} \qquad \text{distinct}(\overline{\texttt{fn}}) \qquad \overline{\text{field-type}(\texttt{fd}) = \texttt{Cn}_\texttt{f}} \qquad \overline{P \vdash \texttt{Cn}_\texttt{f}} \qquad \overline{\text{method-name}(\texttt{md}) = \texttt{mn}} \qquad \text{distinct}(\overline{\texttt{mn}}) \qquad \overline{P, \texttt{Cn} \vdash \texttt{md}}}{P \vdash \textbf{class}\ \texttt{Cn}\ \{\overline{\texttt{fd}}\ \overline{\texttt{md}}\}}\ \text{WF-SIMPLE-CLASS}$$

$$\frac{\overline{\text{parameter-name}(\texttt{pd}) = \texttt{pn}} \qquad \text{distinct}(\overline{\texttt{pn}}) \qquad \overline{\text{parameter-type}(\texttt{pd}) = \texttt{Cn}_p} \qquad \overline{P \vdash \texttt{Cn}_p} \qquad \Gamma = [\textbf{this} \mapsto \texttt{Cn}]\overline{[\texttt{pn} \mapsto \texttt{Cn}_p]} \qquad P, \Gamma \vdash \overline{\texttt{stmt}} \qquad P \vdash \Gamma(x) <: \texttt{Cn}_r}{P, \texttt{Cn} \vdash \texttt{Cn}_r\ \ \texttt{mn}\ \ (\overline{\texttt{pd}})\ \{\overline{\texttt{stmt}}\ \ \textbf{return}\ \texttt{x};\ \}}\ \text{WF-METHOD}$$

Figure 5.14: Well-formedness rules for simple classes and their methods.

A simple class `Cn` is well-formed with respect to P, if all its fields have distinct names and well-formed types, and if all methods have distinct names and are well-formed with respect to the program and the simple class ($\overline{P, \texttt{Cn} \vdash \texttt{md}}$).

A method `md` is well-formed if all its parameters have distinct names and have well-formed types. Further, the body $\overline{\texttt{stmt}}$ must be well-formed with respect to the program P and a local type environment Γ. Finally, we must ensure that `x`, the value returned by the methods, is a member of the return type $\texttt{Cn}_{ret}$.

5.4.4.3 Just-in-Time Classes

A just-in-time class combines a set of representations and inherits the methods defined in these representations. We explain in chapter 4 that this does not lead to ambiguity because there is always one active representation which is the prime supplier of method definitions. For well-formedness, however, it is important that combining the methods of multiple representations does not introduce ambiguity in the type (signature) of these methods.

Consider the three classes in listing 5.7. The simple classes `R1` and `R2` both implement a method `foo` and a method `bar`. The just-in-time class `T` combines both `R1` and

Listing 5.7: The classes R1, R2, and T

```
1  class R1 {
2    A1 foo(B1 b) { return null; }
3    A1 bar(B1 b) { return null; }
4  }
5
6  class R2 {
7    A1 foo(B1 b) { return null; }
8    A2 bar(B2 b) { return null; }
9  }
10
11 class T combines R1, R2 { }
```

R2. As a result, T inherits a foo from both R1 and R2, and T inherits a bar from both R1 and R2.

Chapter 4 explains that, depending on the current representation, the implementation of R1 or the implementation of R2 is executed, i. e., without any ambiguity. Here, we focus on the type of the methods foo and bar of T. Because the type of foo is B1 → A1 in both R1 and R2, it is fair to say that the type in T is B1 → A1 as well. The types B1 → A1 and B2 → A2, from bar in R1 and R2, respectively, are not unifiable. Hence, we do not allow these two simple classes to be combined into one just-in-time class. Alternatively, it is possible to extend JITds such that methods with the same name but different types denote different methods (overloading). However, taking overloading into account, needlessly overcomplicates (method) look-up functions, reduction rules, well-formedness rules, and the proofs that rely on these rules. LightweightJava, for instance, does not allow method overloading either. Hence, we define that two methods are combinable ($\uplus$) into a just-in-time class if either their names differ, or if their types are identical. These two auxiliary rules are shown in figure 5.15.

$$\frac{\mathsf{method\text{-}name}(\mathtt{md}_1) \neq \mathsf{method\text{-}name}(\mathtt{md}_2)}{\mathtt{md}_1 \uplus \mathtt{md}_2}\ \text{WF-AUX-COMB-METHOD1}$$

$$\frac{\mathsf{method\text{-}name}(\mathtt{md}_1) = \mathsf{method\text{-}name}(\mathtt{md}_2) \quad \mathsf{method\text{-}type}(\mathtt{md}_1) = \mathsf{method\text{-}type}(\mathtt{md}_2)}{\mathtt{md}_1 \uplus \mathtt{md}_2}\ \text{WF-AUX-COMB-METHOD2}$$

Figure 5.15: Auxiliary rules to check if two methods are "combinable".

Now that we have established which methods are combinable, it is possible to define what a well-formed just-in-time class looks like. Foremost, all representations $\overline{\mathtt{Cn}_{rep}}$ must be distinct and should denote well-formed types in P. Furthermore, all methods

inherited from any representation have to be pairwise combinable (cf. $\overline{\overline{\mathtt{md}_1 \uplus \mathtt{md}_2}}$). The transition functions $\overline{\mathtt{td}}$ have to define distinct transitions and have to be well-formed themselves.

Finally, a transition function is well-formed if it defines a transition between two distinct representations of `Cn` and if the body is well-formed with respect to a local typing environment Γ (WF-TRANSITION-FUNCTION).

$$\frac{\begin{array}{c}\mathsf{distinct}(\overline{\mathtt{Cn}_{rep}}) \\ \overline{P \vdash \mathtt{Cn}_{rep}} \quad \overline{\mathsf{find\text{-}class}(P, \mathtt{Cn}_{rep}) = \mathtt{cd}_{rep}} \quad \overline{\mathsf{simple\text{-}class}(\mathtt{cd}_{rep})} \\ \overline{\mathsf{class\text{-}methods}(\mathtt{cd}_{rep}) = \overline{\mathtt{md}_1}} \quad \overline{\mathsf{class\text{-}methods}(\mathtt{cd}_{rep}) = \overline{\mathtt{md}_2}} \quad \overline{\overline{\mathtt{md}_1 \uplus \mathtt{md}_2}} \\ \overline{\mathtt{td} = \mathtt{Cn}_{from}\ \mathbf{to}\ \mathtt{Cn}_{to}\ \{...\}} \quad \mathsf{distinct}\left(\overline{(\mathtt{Cn}_{from}, \mathtt{Cn}_{to})}\right) \quad \overline{P, \mathtt{Cn}_{jit} \vdash \mathtt{td}}\end{array}}{P \vdash \mathbf{class}\ \mathtt{Cn}_{jit}\ \mathbf{combines}\ \overline{\mathtt{Cn}_{rep}}\{\ \overline{\mathtt{td}}\ \}}\ \text{WF-JIT-CLASS}$$

$$\frac{\begin{array}{c}\mathtt{Cn}_{from}, \mathtt{Cn}_{to} \in \mathsf{class\text{-}representation\text{-}names}(\mathtt{Cn}) \\ \Gamma = [\mathbf{source} \mapsto \mathtt{Cn}_{from}][\mathbf{target} \mapsto \mathtt{Cn}_{to}] \quad \overline{P, \Gamma \vdash \mathtt{stmt}}\end{array}}{P, \mathtt{Cn} \vdash \mathtt{Cn}_{from}\ \mathbf{to}\ \mathtt{Cn}_{to}\ \{...\}}\ \text{WF-TRANSITION-FUNCTION}$$

Figure 5.16: Well-formedness rules for Just-in-Time classes and their transition functions.

5.4.4.4 Statements

Statements in JITds have to be well-formed with respect to a program P and a local typing environment Γ. Γ is used to look up and verify expected types of variables, P is used to consult the program's structures (e. g., classes and their members).

Most well-formedness rules are fairly straightforward. A block statement, for instance, is well-formed if all the statements in the block are well-formed (cf. WF-BLOCK). An assignment statement is well-formed if the value of `x` is a member of the type of `vn` in Γ (cf. WF-VAR-ASSIGN). An if statement is well-formed if both the consequent and the alternative are well-formed in Γ, and if `x` and `y` are type-wise related in Γ (cf. WF-IF). WF-BLOCK, WF-VAR-ASSIGN, and WF-IF are shown in figure 5.17.

$$\frac{\overline{P,\Gamma \vdash \texttt{stmt}}}{P,\Gamma \vdash \overline{\texttt{stmt}}}\ \text{WF-BLOCK}$$

$$\frac{P \vdash \Gamma(\texttt{x}) <: \Gamma(\texttt{vn})}{P,\Gamma \vdash \texttt{vn = x;}}\ \text{WF-VAR-ASSIGN}$$

$$\frac{(P \vdash \Gamma(\texttt{x}) <: \Gamma(\texttt{y}) \vee P \vdash \Gamma(\texttt{y}) <: \Gamma(\texttt{x})) \qquad P,\Gamma \vdash \texttt{stmt}_t \qquad P,\Gamma \vdash \texttt{stmt}_f}{P,\Gamma \vdash \textbf{if}\ (\ \texttt{x==y}\ )\ \texttt{stmt}_t\ \textbf{else}\ \texttt{stmt}_f}\ \text{WF-IF}$$

Figure 5.17: Well-formedness rules for block statements, variables assignment, and if statements.

The field read statement `vn = x.fn` is well-formed if it can be verified in Γ that the type of `fn` in the expected type of `x` is a subtype of the type of `vn`. Conversely, the field write `x.fn = vn` is well-formed if it can be verified in Γ that the type of `vn` is a subtype of the type of `fn` in the expected type of `x`. Both rules, WF-FIELD-READ and WF-FIELD-WRITE, are shown in figure 5.18.

$$\frac{\Gamma(\texttt{x}) = \texttt{Cn} \qquad \textsf{find-class}(P,\texttt{Cn}) = \texttt{cd} \qquad \textsf{find-field}(\texttt{cd},\texttt{fn}) = \texttt{fd} \qquad \textsf{field-type}(\texttt{fd}) = \texttt{Cn}' \qquad P \vdash \texttt{Cn}' <: \Gamma(\texttt{vn})}{P,\Gamma \vdash \texttt{vn = x.fn;}}\ \text{WF-FIELD-READ}$$

$$\frac{\Gamma(\texttt{x}) = \texttt{Cn} \qquad \textsf{find-class}(P,\texttt{Cn}) = \texttt{cd} \qquad \textsf{find-field}(\texttt{cd},\texttt{fn}) = \texttt{fd} \qquad \textsf{field-type}(\texttt{fd}) = \texttt{Cn}' \qquad P \vdash \Gamma(\texttt{y}) <: \texttt{Cn}'}{P,\Gamma \vdash \texttt{x.fn = y;}}\ \text{WF-FIELD-WRITE}$$

Figure 5.18: Well-formedness rules for rules for field read and field write statements.

The creation of an object is well-formed if the static class of the created object is a subtype of the variable it is being assigned to. To create a simple object the static class `Cn` must denote a simple class from P. To create a just-in-time object, the static class $\texttt{Cn}_s$ must be a just-in-time class and the dynamic class $\texttt{Cn}_d$ must be a representation thereof ($\texttt{Cn}_d \in \textsf{class-representation-names}(\texttt{cd})$). This is formalised in the rules WF-NEW-SIMPLE and WF-NEW-JIT, respectively. Both rules are shown in figure 5.19.

$$\frac{P \vdash \texttt{Cn} <: \Gamma(\texttt{vn}) \qquad \textsf{find-class}(P, \texttt{Cn}) = \texttt{cd} \qquad \textsf{simple-class}(\texttt{cd})}{P, \Gamma \vdash \texttt{vn = new Cn();}}\ \text{WF-NEW-SIMPLE}$$

$$\frac{P \vdash \texttt{Cn}_s <: \Gamma(\texttt{vn}) \qquad \textsf{find-class}(P, \texttt{Cn}_s) = \texttt{cd} \qquad \texttt{Cn}_d \in \textsf{class-representation-names}(\texttt{cd})}{P, \Gamma \vdash \texttt{vn = new Cn}_s\texttt{.Cn}_d\texttt{();}}\ \text{WF-NEW-JIT}$$

Figure 5.19: Well-formedness rules for rules for object creation statements.

A method invocation is well-formed if it can be verified in Γ that the expected types of all arguments $\overline{\texttt{a}}$ are subtypes of the parameter types of `mn` and that the return type of `mn` is a subtype of the type of the variable `vn` (see figure 5.20).

$$\frac{\Gamma(\texttt{x}) = \texttt{Cn} \qquad \textsf{method-type}(P, \texttt{Cn}, \texttt{mn}) = \overline{\texttt{Cn}_{arg}} \rightarrow \texttt{Cn}_{ret} \qquad \overline{P \vdash \Gamma(\texttt{a}) <: \texttt{Cn}_{arg}} \qquad P \vdash \texttt{Cn}_r <: \Gamma(\texttt{vn})}{P, \Gamma \vdash \texttt{vn = x.mn}(\overline{\texttt{a}})\texttt{;}}\ \text{WF-METHOD-INVOCATION}$$

Figure 5.20: Well-formedness rules for rule for method invocations.

WF-SWAP states that a swap statement is well-formed if it can be verified in Γ and P that the type of `x` is a just-in-time class [3] and that $\texttt{Cn}_{tar}$ is a representation of that just-in-time class.[4]

$$\frac{P \vdash \Gamma(\texttt{x}) <: \texttt{Cn}_{jit} \qquad \textsf{find-class}(P, \texttt{Cn}_{jit}) = \texttt{cd} \qquad \texttt{Cn}_{tar} \in \textsf{class-representation-names}(\texttt{cd})}{P, \Gamma \vdash \texttt{x to Cn}_{tar}\texttt{;}}\ \text{WF-SWAP}$$

Figure 5.21: Well-formedness rule for rules for swap statements.

5.5 Proof of Soundness

The type system described in section 5.4 is a traditional object-oriented type system in the sense that it is designed to help programmers to avoid the runtime type errors

[3] Implicit from $\texttt{Cn}_{tar} \in$ class-representation-names(cd)
[4] Explicit from $\texttt{Cn}_{tar} \in$ class-representation-names(cd)

"method not found" (cf. "message not understood" in SmallTalk) and "field not found". To show that our type system actually does what it claims to do, it must be proven *sound*. In general, a type system is said to be sound if well-formed programs do not generate runtime type errors [89]. One can prove that a type system is sound, if one can show that any well-formed program state (cf. configuration) never gets stuck by performing a legal reduction defined in the operation semantics (i. e., to prove progress) and and if one can also show that performing a legal reduction defined in the operational semantics keeps the program state well-formed (i. e., to prove preservation). Before we prove both *progress* and *preservation*, we define what it means for *configurations*, *heaps*, *local stores*, and *objects* to be well-formed.

5.5.1 Well-formed Configurations, Heaps, Local Stores, and Objects

Well-formedness of a configuration is checked in terms of a *global type environment* Γ. A global type environment is a mapping from invocation contexts to local type environments. Because an invocation context uniquely defines a method or a transition function, $\Gamma(I)$ can be used to find the local type environment as found in the well-formedness rules WF-METHOD and WF-TRANSITION-FUNCTION, for methods and transition functions, respectively.

Configurations are well-formed in a global type environment Γ when the program P is well-formed, when the heap H is well-formed, and when the stack's *top frame* is well-formed. A frame with invocation context I is well-formed if the local store is well-formed in the local type environment $\Gamma(I)$ and if, when applicable, the statements are well-formed in the local type environment $\Gamma(I)$ (see figure 5.22).

Note that we use Γ to denote both global type environment and local type environments. This generally does not introduce ambiguity, because we usually do not consider both at the same time. When we do, however, we explicitly take the effort to avoid the ambiguity.

$$\frac{\vdash P \qquad P \vdash H \qquad P, \Gamma(I), H \vdash L \qquad \overline{P, \Gamma(I) \vdash \texttt{stmt}}}{\Gamma \vdash (\langle \overline{\texttt{stmt}}, L, I \rangle : S, H, P)}\ \text{WF-CONFIG}$$

$$\frac{\vdash P \qquad P \vdash H \qquad P, \Gamma(I), H \vdash L}{\Gamma \vdash (\langle \mathit{Exception}, L, I \rangle : S, H, P)}\ \text{WF-CONFIG-EX}$$

Figure 5.22: Well-formedness rule for rules for configurations.

To show that a heap H is well-formed it is sufficient to show that its domain is finite and that all objects in the heap are well-formed. A local store L is well-formed when it has a finite domain and when it can be shown that all variables in Γ have types that are supertypes of the types of values associated with them in L. Finally, an object is well-formed when it has values for all the fields $\overline{\texttt{fn}}$ defined by the dynamic class of the object

and if these values are members of the field types defined by the dynamic class of the object, i. e., subtypes of $\overline{\texttt{Cn}_\texttt{f}}$.

$$\frac{\mathsf{finite}(\mathsf{dom}(H)) \qquad \forall \mathit{objId} \in \mathsf{dom}(H) : P, H \vdash H(\mathit{objId})}{P \vdash H}\ \text{WF-HEAP}$$

$$\frac{I = (\texttt{Cn}, \texttt{mn}) \qquad \mathsf{finite}(\mathsf{dom}(L)) \qquad \forall \texttt{x} \in \mathsf{dom}(\Gamma) : P, H \vdash L(\texttt{x}) <: \Gamma(\texttt{x}) \qquad P, H \vdash L(\textbf{this}) =: \Gamma(\textbf{this})}{P, \Gamma, H, I \vdash L}\ \text{WF-LOCAL-STORE-M}$$

$$\frac{I = (\texttt{Cn}_{jit}, \texttt{Cn}_{src}, \texttt{Cn}_{tar},) \qquad \mathsf{finite}(\mathsf{dom}(L)) \qquad \forall \texttt{x} \in \mathsf{dom}(\Gamma) : P, H \vdash L(\texttt{x}) <: \Gamma(\texttt{x}) \qquad P, H \vdash L(\textbf{source}) =: \Gamma(\textbf{source}) \qquad P, H \vdash L(\textbf{target}) =: \Gamma(\textbf{target})}{P, \Gamma, H, I \vdash L}\ \text{WF-LOCAL-STORE-TF}$$

$$\frac{\mathsf{find\text{-}class}(P, \texttt{Cn}_d) = \texttt{cd} \qquad \mathsf{class\text{-}field\text{-}names}(\texttt{cd}) = \overline{\texttt{fn}} \qquad \mathsf{dom}(F) = \overline{\texttt{fn}} \qquad \mathsf{class\text{-}field\text{-}types}(\texttt{cd}) = \overline{\texttt{Cn}_\texttt{f}} \qquad \overline{P, H \vdash F(\texttt{fn}) <: \texttt{Cn}_\texttt{f}} \qquad \mathsf{valid\text{-}classes}(P, \texttt{Cn}_s, \texttt{Cn}_d)}{P, H \vdash (\texttt{Cn}_s, \texttt{Cn}_d, F)}\ \text{WF-OBJECT}$$

$$\frac{\mathsf{find\text{-}class}(P, \texttt{Cn}_s) = \texttt{cd}_s \qquad \mathsf{simple\text{-}class}(\texttt{cd}_s) \qquad \texttt{Cn}_s = \texttt{Cn}_d}{\mathsf{valid\text{-}classes}(P, \texttt{Cn}_s, \texttt{Cn}_d)}\ \text{WF-AUX-VALID-CLASSES-SIMPLE}$$

$$\frac{\mathsf{find\text{-}class}(P, \texttt{Cn}_s) = \texttt{cd}_s \qquad \mathsf{jit\text{-}class}(\texttt{cd}_s) \qquad \texttt{Cn}_d \in \mathsf{class\text{-}representation\text{-}names}(\texttt{cd}_s)}{\mathsf{valid\text{-}classes}(P, \texttt{Cn}_s, \texttt{Cn}_d)}\ \text{WF-AUX-VALID-CLASSES-JIT}$$

Figure 5.23: Well-formedness rules for heaps, local stores, and objects.

5.5.2 Progress

A first property that must hold for a type system to be sound is that it should guarantee that a well-formed program does not get stuck if there are still statements to be executed. In other words, this property, called *progress*, states that a well-formed configuration can always be reduced (eq. (5.1)). There are two kinds of configurations that cannot be reduced: a final configuration $(\langle \emptyset, L, I \rangle : \varepsilon, H, P)$ or an exception configuration

$(\langle Exception, L, I\rangle : S, H, P)$.

$$\Gamma \vdash c \;\Rightarrow\; \exists c' : c \longrightarrow c' \tag{5.1}$$

During the proof we know that the initial configuration c is well-formed in some Γ, cf. the left hand side of eq. (5.1). Consequently, because we can rely on c to be well-formed, we can also assume the program, the local store, the heap, and all statements (including the first) to be well-formed. Then, we use structural induction on the sequence of statements $\overline{\texttt{stmt}}$ found in the top frame of the stack S, to prove that for any non-final, non-exception configuration it is possible to find a next configuration c' using the reduction rules from section 5.2 (cf. the right hand side of eq. (5.1)). The structural induction consist of eleven cases, depending on the first statement found in $\overline{\texttt{stmt}}$:

$$\underline{\{\,\overline{\texttt{stmt}}\,\}}$$

The application of R-BLOCK is always possible.

$$\underline{\textbf{if}\ (\ \texttt{x==y}\ )\ \texttt{stmt}_t\ \textbf{else}\ \texttt{stmt}_f}$$

From WF-IF we know $\texttt{x},\texttt{y} \in dom(\Gamma)$. Because the local store is well-formed as well (cf. WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF) we are assured that $\texttt{x},\texttt{y} \in dom(L)$. The latter guarantees the existence of two values v and w such that $L(\texttt{x}) = v$ and $L(\texttt{y}) = w$. If $v = w$ then R-IF-T is applicable, otherwise when $v \neq w$ R-IF-F is applicable.

$$\underline{\texttt{vn = x;}}$$

From WF-VAR-ASSIGN we know that $\texttt{x} \in dom(\Gamma)$; from WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF we then know that $\texttt{x} \in dom(L)$; thus, we know there must be a v such that $L(\texttt{x}) = v$. $L(\texttt{x}) = v$ is a sufficient condition to apply R-VARIABLE-ASSIGN.

$$\underline{\texttt{x.fn = y;}}$$

From WF-FIELD-WRITE we know that $\texttt{x},\texttt{y} \in dom(\Gamma)$ and from WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF we know that $\texttt{x},\texttt{y} \in dom(L)$. Either $L(\texttt{x}) = \textbf{null}$ or $L(\texttt{x}) = objId$. In the former case R-FIELD-WRITE-NPE is applicable. In the latter case, R-FIELD-WRITE is applicable.

$$\underline{\texttt{vn = x.fn;}}$$

From WF-FIELD-READ we know that x has a known type Cn in Γ, further from WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF it follows that x has a value in L because $\texttt{x} \in dom(L)$. Either $L(\texttt{x}) = \textbf{null}$ or $L(\texttt{x}) = objId$. In the former case R-FIELD-READ-NPE is applicable. In the later case, R-FIELD-READ is applicable because from WF-LOCAL-STORE-M (or WF-LOCAL-STORE-TF), WF-OBJECT and ST-DYNAMIC-CLASS it follows that: 1. an object exists in the heap ($objId \in \mathrm{dom}(H)$), 2. the object is well-formed ($P, H \vdash H(objId)$), and 3. thus must have a value v bound to the field fn.

$$\underline{\texttt{vn = new Cn();}}$$

From WF-NEW-SIMPLE we know that Cn is a valid type (find-class(P,Cn = cd)), and that cd is a simple class (simple-class(cd)). From which it trivially follows that we can retrieve the field names associated with that simple class. WF-HEAP requires the heap to be finite which means that there must exist a fresh object identifier *objId* such that *objId* $\notin$ dom(H). As a result, all preconditions are met to legally apply R-NEW-SIMPLE.

$$\underline{\texttt{vn = new Cn}_s\texttt{.Cn}_d\texttt{();}}$$

From WF-NEW-JIT we know that $\texttt{Cn}_s$ is a valid type (find-class(P,$\texttt{Cn}_s$ = cd) and that cd is a JIT class ($\texttt{Cn}_d \in$ class-representation-names(cd)). From the latter follows, indirectly, that $\texttt{Cn}_d$ is a valid type that refers to a simple class (representations of just-in-time classes are simple classes). As a result it must be possible to retrieve the field names associated with that simple class $\texttt{Cn}_d$. WF-HEAP requires the heap to be finite which means that there must exist a fresh object identifier *objId* such that *objId* $\notin$ dom(H). Now, all preconditions are met to legally apply R-NEW-JIT.

$$\underline{\texttt{vn = x.mn(}\overline{\texttt{a}}\texttt{);}}$$

From WF-METHOD-INVOCATION we know that x has a known type Cn in Γ (Γ(x) = Cn), then from WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF it follows that x has a value in L (x $\in dom(L)$). Either L(x) = **null**, in which case R-FIELD-READ-NPE is applicable, or L(x) = *objId*. If the latter is true, then from WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF and WF-OBJECT it follows that an object *obj* exists in the heap (*objId* $\in$ dom(H)). Moreover, the object *obj* is guaranteed to be well-formed. Thus *obj* has a dynamic class $\texttt{Cn}_d$ which refers to a simple class definition cd. By mutual exclusion we differentiate between three cases: either there is a method md in cd, or there is no such method but we can find a transition path, or there is no such method and we cannot find a transition path. We now discuss each of these cases separately:

- find-method(cd,mn) = md
  From WF-METHOD-INVOCATION and WF-LOCAL-STORE-M (or WF-LOCAL-STORE-TF) we know that all arguments $\overline{\texttt{a}}$ have a corresponding value in L. This is sufficient to construct a new stack frame and apply the rule R-MI-DIRECT.

- ¬find-method(cd,mn), but transition-path(P,$\texttt{Cn}_s$,$\texttt{Cn}_d$,mn) = $\overline{\texttt{Cn}_t}$
  Then R-MI-INDIRECT is directly applicable.

- ¬find-method(cd,mn) and ¬transition-path(P,$\texttt{Cn}_s$,$\texttt{Cn}_d$,mn)
  Then R-MI-USE is directly applicable.

$$\underline{\texttt{return y;}}$$

By construction, it is guaranteed that when a return statement is encountered, the first expression of the *second* stack frame is a method invocation expression `vn = x.mn(`$\overline{\texttt{a}}$`);`. Then, by mutual exclusion, caller-has-expected-type(L_c, I_c, H) either succeeds or fails. If it succeeds, R-MR-DIRECT is applicable. R-MR-DIRECT only requires $\texttt{y} \in dom(L)$ which is possible because of WF-METHOD, which requires exactly that $\texttt{y} \in dom(\Gamma)$ and because of WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF it is thus known that $\texttt{y} \in dom(L)$. If caller-has-expected-type(L_c, I_c, H) fails, R-MR-INDIRECT is applicable. Because L is finite, an $\texttt{x}_c$ such that $\texttt{x}_c \notin dom(L)$, can be found.

$$\underline{\texttt{x to Cn}_{tar}\texttt{;}}$$

From WF-SWAP we know that x has a known type $\texttt{Cn}_{jit}$ in Γ ($\Gamma(\texttt{x}) = \texttt{Cn}_{jit}$), then from WF-LOCAL-STORE-M follows that x has a value in L $\texttt{x} \in dom(L)$. Either $L(\texttt{x}) = \textbf{null}$, in which case R-SWAP-NPE is applicable, or $L(\texttt{x}) = objId$. If the latter is true, then from WF-LOCAL-STORE-M and WF-OBJECT it follows that one such object exists in the heap ($objId \in \mathrm{dom}(H)$), that is well-formed ($P, H \vdash (\texttt{Cn}_s, \texttt{Cn}_{src}, \overline{\texttt{fn} \mapsto v})$). If $\texttt{Cn}_{src} = \texttt{Cn}_{tar}$ then R-SWAP-IDEMPOTENT trivially applies. Otherwise, we try to find a transition function td from $\texttt{Cn}_{src}$ to $\texttt{Cn}_{tar}$ in $\texttt{cd}_{jit}$, where $\texttt{cd}_{jit}$ is the class definition of $\texttt{Cn}_{jit}$. Then, by mutual exclusion, we differentiate between three cases: either td does not exist and no transition path exists between $\texttt{Cn}_{src}$ and $\texttt{Cn}_{tar}$, or td does not exist and a transition path between $\texttt{Cn}_{src}$ and $\texttt{Cn}_{tar}$ can be found, or td does exist. In the first case, R-SWAP-USE is directly applicable. In the second case, R-SWAP-INDIRECT is directly applicable. In the last case, R-SWAP-DIRECT is applicable.

$$\underline{\emptyset}$$

By construction, it is guaranteed that when the empty statement sequence is encountered, the first expression of the *second* stack frame is a swap statement `x to Cn`$_{tar}$`;`. Then, by mutual exclusion, caller-has-expected-type(L_c, I_c, H) either succeeds or fails. If it succeeds, R-TFR-DIRECT is applicable. R-TFR-DIRECT only requires $\texttt{y} \in dom(L)$ which is possible because of WF-METHOD, which requires exactly that $\texttt{y} \in dom(\Gamma)$ and because of WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF it is thus known that $\texttt{y} \in dom(L)$. If caller-has-expected-type(L_c, I_c, H) fails, R-TFR-INDIRECT is applicable. Because L is finite, an $\texttt{x}_c$ such that $\texttt{x}_c \notin dom(L)$, can be found.

We showed for each of the eleven possible occurrences of a well-formed configuration c we can find a configuration c' by applying reduction of the operation semantics only. Hence, we have proven *progress*. □

5.5.3 Preservation

The second property for a type system to be sound is that it should guarantee that the execution of a well-formed program maintains the well-formedness of configurations during the execution. In other words, this property called *preservation*, states that each legal reduction of a well-formed configuration again produces a well-formed configuration. This is expressed in eq. (5.2), which literally states that when a configuration c

is well-formed in some global type environment Γ and when that configuration can be reduced to c', then there must exists a global type environment Γ' that encodes the same information as Γ (or more), in which c' is well-formed. Using structural induction on the 23 reduction rules, we can show that eq. (5.2) always holds in JIT$\Delta\sigma$.

$$\Gamma \vdash c \wedge c \longrightarrow c' \Rightarrow \exists \Gamma' : \Gamma \subseteq \Gamma' \wedge \Gamma' \vdash c' \tag{5.2}$$

R-BLOCK, R-IF-T, R-IF-F, and R-SWAP-IDEMPOTENT (simple rewrite reductions)

c' is trivially well-formed because of WF-CONFIG and the respective statement well-formedness rules, i. e., WF-BLOCK for R-BLOCK, and WF-IF for R-IF-T and R-IF-F. RSWAPIDEM does not rely on another rule.

R-FIELD-READ-NPE, R-FIELD-WRITE-NPE, R-MI-NPE, R-SWAP-NPE

R-MI-USE, and R-SWAP-USE

All reductions that lead to an exception configuration, result in a well-formed exception configuration. This follows from WF-CONFIG-EX because P, H, and L remain unchanged.

R-VARIABLE-ASSIGN

This reduction removes the first statement and leaves all other statements untouched. The only other difference between c and c' is the local store. Hence, for $\Gamma \vdash c'$ to hold we must show that $P, \Gamma, H \vdash L[\texttt{vn} \mapsto v]$ with $L(\texttt{x}) = v$. R-VARIABLE-ASSIGN does not change the domain of L and thus $dom(L)$ remains finite. Moreover, R-VARIABLE-ASSIGN does not change anything in L except the value of `vn`. Hence, what is left to show is that $P, H \vdash v <: \Gamma(\texttt{vn})$. This is trivially true when v is **`null`**, because of ST-NULL. Otherwise, v must be some $objId \in \mathsf{dom}(H)$ for which we know that $H(objId) = (\texttt{Cn}_s, \texttt{Cn}_d, \overline{\texttt{fn} \mapsto u})$ (WF-HEAP, WF-OBJECT), which simplifies our goal to show that $P, H \vdash \texttt{Cn}_s <: \Gamma(\texttt{vn})$. From WF-VAR-ASSIGN we know that $P, H \vdash \Gamma(\texttt{x}) <: \Gamma(\texttt{vn})$. From the transitivity of $<:$ it follows that $P, H \vdash \texttt{Cn}_s <: \Gamma(\texttt{vn})$.

R-FIELD-READ

This reduction removes the first statement and leaves all other statements untouched. The only other difference between c and c' is the local store. Hence, for $\Gamma \vdash c'$ to hold we must show that $P, \Gamma, H \vdash L[\texttt{vn} \mapsto v]$ with $L(\texttt{x}) = objId_x$ and with $H(objId_x, \texttt{fn}) = v$. R-FIELD-READ does not change the domain of L and thus $dom(L)$ remains finite. Moreover, R-FIELD-READ does not change anything in L except the value of `vn`. Hence, what is left to show is that $P, H \vdash v <: \Gamma(\texttt{vn})$. This is trivially true when v is **`null`**, because of ST-NULL.

Otherwise, v must be some $objId_f \in \mathsf{dom}(H)$ for which we know that $H(objId_f) = (\mathtt{Cn}_{sf}, \mathtt{Cn}_{df}, F_f)$ (WF-HEAP, WF-OBJECT), which simplifies our goal to show that $P, H \vdash \mathtt{Cn}_{sf} <: \Gamma(\mathtt{vn})$. From WF-FIELD-READ, on the one hand, it follows that $\Gamma(\mathtt{x}) = \mathtt{Cn}_x$, that $\mathsf{find\text{-}class}(P, \mathtt{Cn}_x) = \mathtt{cd}_x$, that $\mathsf{find\text{-}field}(\mathtt{cd}_x, \mathtt{fn}) = \mathtt{fd}_{xf}$, that $\mathsf{field\text{-}type}(\mathtt{fd}_{xf}) = \mathtt{Cn}_{xf}$, and that $P \vdash \mathtt{Cn}_{xf} <: \Gamma(\mathtt{vn})$. From WF-HEAP and WF-OBJECT, on the other hand, we know that $objId_x \in \mathsf{dom}(H)$, that $H(objId_x) = (\mathtt{Cn}_{sx}, \mathtt{Cn}_{dx}, F_x)$, that $\mathsf{find\text{-}class}(P, \mathtt{Cn}_{dx}) = \mathtt{cd}_{dx}$, that $\mathsf{find\text{-}field}(\mathtt{cd}_{dx}, \mathtt{fn}) = \mathtt{fd}_{dxf}$, and that $\mathsf{field\text{-}type}(\mathtt{fd}_{dxf}) = \mathtt{Cn}_{dxf}$ and $P \vdash \mathtt{Cn}_{sf} <: \mathtt{Cn}_{dxf}$. In the context of a field read, the syntax of JITds only allows `x` to be either **`this`** or **`source`**. Hence, we know from WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF that $P, H \vdash L(\mathtt{x}) =: \Gamma(\mathtt{x})$. In other words, we can show that $\mathtt{Cn}_x = \mathtt{Cn}_{dx}$ and thus $\mathtt{Cn}_{xf}$ and $\mathtt{Cn}_{dxf}$ are equal as well. From the transitivity of <: it then follows that $P, H \vdash \mathtt{Cn}_{sf} <: \Gamma(\mathtt{vn})$, which was left to prove.

R-FIELD-WRITE

This reduction removes the first statement and leaves all other statements untouched. The only other difference between c and c' is the heap. To be more precise it is one object in that heap, i. e., $H(objId_x) = (\mathtt{Cn}_{sx}, \mathtt{Cn}_{dx}, F_x)$, with $L(x) = objId_x$, where the value of a field has changed. Hence, to show that $P \vdash H$, it suffices to show that $P, H \vdash (\mathtt{Cn}_s, \mathtt{Cn}_d, F)$ with $F(\mathtt{fn}) = v$ and $L(\mathtt{y}) = v$ after the application of R-FIELD-WRITE. Because $\mathsf{find\text{-}class}(P, \mathtt{Cn}_{dx}) = \mathtt{cd}_{dx}$, $\mathsf{find\text{-}field}(\mathtt{cd}_{dx}, \mathtt{fn}) = \mathtt{fd}_{dxf}$, and $\mathsf{field\text{-}type}(\mathtt{fd}_{dxf}) = \mathtt{Cn}_{dxf}$ it suffices to show that $P, H \vdash v <: \mathtt{Cn}_{dxf}$. This is trivially true when v is **`null`**, because of ST-NULL.

Otherwise, v must be some $objId_y \in \mathsf{dom}(H)$ for which we know that $H(objId_y) = (\mathtt{Cn}_{sy}, \mathtt{Cn}_{dy}, F_y)$ (WF-HEAP, WF-OBJECT), which simplifies our goal to show that $P, H \vdash \mathtt{Cn}_{sy} <: \mathtt{Cn}_{dxf}$. From WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF we know that $P \vdash \mathtt{Cn}_{sy} <: \Gamma(\mathtt{y})$. From WF-FIELD-WRITE it follows that $\Gamma(\mathtt{x}) = \mathtt{Cn}_x$, that $\mathsf{find\text{-}class}(P, \mathtt{Cn}_x) = \mathtt{cd}_x$, that $\mathsf{find\text{-}field}(\mathtt{cd}_x, \mathtt{fn}) = \mathtt{fd}_{xf}$, $\mathsf{field\text{-}type}(\mathtt{fd}_{xf}) = \mathtt{Cn}_{xf}$, and that $P \vdash \Gamma(\mathtt{y}) <: \mathtt{Cn}_{xf}$. In the context of a field write, the syntax of JITds only allows `x` to be either **`this`** or **`target`**. Hence, we know from WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF that $P, H \vdash L(\mathtt{x}) =: \Gamma(\mathtt{x})$. Hence, $\mathtt{Cn}_x$ is equal to $\mathtt{Cn}_{dx}$ and thus is $\mathtt{Cn}_{xf} = \mathtt{Cn}_{dxf}$. Then $P, H \vdash \mathtt{Cn}_{sy} <: \mathtt{Cn}_{dxf}$ follows from the transitivity of <:.

R-NEW-SIMPLE

We need to show two things: first, we need to show that $P, H \vdash (\mathtt{Cn}, \mathtt{Cn}, F)$ (well-formed object); and second, we need to show that $P, H[objId \mapsto (\mathtt{Cn}, \mathtt{Cn}, F)] \vdash objId <: \Gamma(\mathtt{vn})$; both with $F = \overline{[\mathtt{fn} \mapsto \mathbf{null}]}$. To show that $(\mathtt{Cn}, \mathtt{Cn}, F)$ is a well-formed object we need to show that $P \vdash \mathbf{null} <: \mathtt{Cn}_f$ holds for each field `fn` of `Cn`. This is trivially true because of ST-NULL. Because of ST-DYNAMIC-CLASS, proving $P, H[objId \mapsto (\mathtt{Cn}, \mathtt{Cn}, F)] \vdash objId <: \Gamma(\mathtt{vn})$ reduces to showing that $P, H[objId \mapsto (\mathtt{Cn}, \mathtt{Cn}, F)] \vdash \mathtt{Cn} <: \Gamma(\mathtt{vn})$, which is trivially true from WF-NEW-SIMPLE.

R-NEW-JIT

We need to show two things: first, we need to show that $P, H \vdash (\texttt{Cn}_s, \texttt{Cn}_d, F)$ (well-formed object); and second, we need to show that $P, H[\mathit{objId} \mapsto (\texttt{Cn}_s, \texttt{Cn}_d, F)] \vdash \mathit{objId} <: \Gamma(\texttt{vn})$; both with $F = \overline{[\texttt{fn} \mapsto \textbf{null}]}$. To show that $(\texttt{Cn}_s, \texttt{Cn}_d, F)$ is a well-formed object we need to show that $P \vdash \textbf{null} <: \texttt{Cn}_f$ holds for each field $\texttt{fn}$ of $\texttt{Cn}_d$. This is trivially true because of ST-NULL. Because of ST-DYNAMIC-CLASS, proving $P, H[\mathit{objId} \mapsto (\texttt{Cn}_s, \texttt{Cn}_d, F)] \vdash \mathit{objId} <: \Gamma(\texttt{vn})$ reduces to showing that $P, H[\mathit{objId} \mapsto (\texttt{Cn}_s, \texttt{Cn}_d, F)] \vdash \texttt{Cn}_s <: \Gamma(\texttt{vn})$, which is trivially true from WF-NEW-JIT.

R-MI-INDIRECT

The only difference between c and c' is the set of statements. In c', a block statement $\texttt{stmt}_t$ is added. Hence, to show that c' is well-formed, we need to show that $\texttt{stmt}_t$ is well-formed under Γ, which reduces to showing $P, \Gamma \vdash \texttt{stmt}$ for all statements in $\texttt{stmt}_t$. First, note that this is trivially true for the last statement of $\texttt{stmt}_t$, i. e., the method invocation which is well-formed because of WF-METHOD-INVOCATION. All other statements in $\texttt{stmt}_t$ are of the form $\texttt{x}$ **to** $\texttt{Cn}_{tar}$**;**, for which it is left to show that $\texttt{x} \in dom(\Gamma)$, and that $\texttt{Cn}$ is one of the representations of $\texttt{Cn}_{jit}$, with $P \vdash \Gamma(\texttt{x}) <: \texttt{Cn}_{jit}$. From WF-METHOD-INVOCATION of the initial statement we know $\texttt{x} \in dom(\Gamma)$. From the existence of the transition path $\overline{\texttt{Cn}_t}$, it follows that all $\texttt{Cn}_t$ are representations of $\texttt{Cn}_{jit}$.

R-MI-DIRECT

We must show that the new stack frame is well-formed. Because the method is well formed, the statements in its body $\overline{\texttt{stmt}_B}$ are also well-formed. From WF-LOCAL-STORE-M or WF-LOCAL-STORE-TF we know that $\overline{v <: \Gamma(\texttt{a})}$ with $L(\texttt{a}) = v$, and from WF-METHOD-INVOCATION we know that $\overline{\Gamma(\texttt{a}) <: \texttt{Cn}_a}$ with $\texttt{Cn}_a$ the type of the respective arguments. Hence, by transitivity of $<:$, we know $v <: \texttt{Cn}_a$ for each argument. Because the method $\texttt{mn}$ of $\texttt{cd}_d$ is well-formed in P, its body statements $\overline{\texttt{stmt}_B}$ are well-formed in some typing environment $\Gamma' = [\textbf{this} \mapsto \texttt{Cn}_d][\overline{\texttt{vn} \mapsto \texttt{Cn}_a}]$. Then, $P, \Gamma', H \vdash L_m$ is true because: First, $dom(L_m)$ is finite by construction; Second, it is trivial to see that $L(x) <: \Gamma(x)$ in Γ'; and Third, $L(\textbf{this}) =: \texttt{Cn}_d$ is trivially true as well.

R-MR-DIRECT

In this reduction we need to show that new top stack frame is well-formed. Because, the statement $\overline{\texttt{stmt}}$ and the invocation context I' have not been changed since they have become the second frame, they are still well-formed. What is left to show is that L' is well-formed, i. e., we must show that $P, \Gamma, H \vdash L_c[\texttt{vn} \mapsto w]$ with $L(\texttt{y}) = w$. Because L_c is well-formed and because R-MR-DIRECT does not change the domain of L_c, caller-has-expected-type guarantees that the calling context has the required types. Hence, the only thing to show is that $P, H \vdash v <: \Gamma(\texttt{vn})$. This is trivially true when v is **null**, because of ST-NULL. Otherwise, v must be some $\mathit{objId} \in \mathsf{dom}(H)$ for which we know that $H(\mathit{objId}) = (\texttt{Cn}_s, \texttt{Cn}_d, \overline{\texttt{fn} \mapsto u})$ (WF-HEAP, WF-OBJECT), which simplifies our goal to show that $P, H \vdash \texttt{Cn}_s <: \Gamma(\texttt{vn})$. From WF-METHOD we know that $P, H \vdash \Gamma(\texttt{y}) <: \Gamma(\texttt{vn})$. Then, from the transitivity of $<:$ it follows that $P, H \vdash \Gamma(\texttt{Cn}_s) <: \Gamma(\texttt{vn})$.

R-MR-INDIRECT

The difference between c and c' is the set of statements and the local store. In c', a statement of the form $\texttt{x}_c$ **to** $\texttt{Cn}_{tar}$**;** is added and the local store is extended with a new binding. Hence, to show that c' is well-formed, we need to show that the statement is well-formed under some Γ' and that the new local store is well-formed under that same Γ'. Hence, what is left to show is that $\texttt{x}_c \in dom(\Gamma')$, that $\texttt{Cn}_{tar}$ is one of the representations of $\texttt{Cn}_{jit}$, with $P \vdash \Gamma'(\texttt{x}_c) <: \texttt{Cn}_{jit}$, and that $P, \Gamma', H \vdash L(x_c) <: \Gamma'(x_c)$. From the existence of the transition path $\overline{\texttt{Cn}_{tar}}$, it follows that all $\texttt{Cn}_{tar}$ are representations of $\texttt{Cn}_{jit}$, and the rest is trivially true by construction.

R-SWAP-INDIRECT

The only difference between c and c' is the set of statements. In c', a block statement $\texttt{stmt}_t$ is added. Hence, to show that c' is well-formed, we need to show that $\texttt{stmt}_t$ is well-formed under Γ, which reduces to showing $P, \Gamma \vdash \texttt{stmt}$ for all statements in $\texttt{stmt}_t$. All statements in $\texttt{stmt}_t$ are of the form $\texttt{x}$ **to** $\texttt{Cn}_{tar}$**;**, for which its is left to show that $\texttt{x} \in dom(\Gamma)$, and that $\texttt{Cn}$ is one of the representations of $\texttt{Cn}_{jit}$, with $P \vdash \Gamma(\texttt{x}) <: \texttt{Cn}_{jit}$. From WF-SWAP of the initial statement we know $\texttt{x} \in dom(\Gamma)$. From the existence of the transition path $\overline{\texttt{Cn}_t}$, it follows that all $\texttt{Cn}_t$ are representations of $\texttt{Cn}_{jit}$.

R-SWAP-DIRECT

We must only show that the new stack frame $\langle \overline{\texttt{stmt}'}, L', I' \rangle$ is well-formed. Because the transition function $\texttt{td}$ is well formed, the statements in its body $\overline{\texttt{stmt}'}$ are also well-formed. Further, L' is well-formed because: First, $dom(L')$ is finite by construction; Second, it is trivial to see that $L(x) <: \Gamma(x)$; ; and Third, $L(\textbf{source}) =: \texttt{Cn}_{src}$ and $L(\textbf{target}) =: \texttt{Cn}_{tar}$ are trivially true as well.

R-TFR-INDIRECT

The difference between c and c' is the set of statements and the local store. In c', a statement of the form $\texttt{x}_c$ **to** $\texttt{Cn}$**;** is added and the local store is extended with a new binding. Hence, to show that c' is well-formed, we need to show that statement is well-formed under some Γ' and that the new local store is well-formed under that same Γ'. Hence, what is left to show is that $\texttt{x}_c \in dom(\Gamma')$, that $\texttt{Cn}$ is one of the representations of $\texttt{Cn}_{jit}$, with $P \vdash \Gamma'(\texttt{x}_c) <: \texttt{Cn}_{jit}$, and that $P, \Gamma', H \vdash L(x_c) <: \Gamma'(x_c)$. From the existence of the transition path $\overline{\texttt{Cn}_t}$, it follows that all $\texttt{Cn}_t$ are representations of $\texttt{Cn}_{jit}$. The rest is trivial because we construct $\texttt{x}_c$ ourselves.

R-TFR-DIRECT

For this reduction we need to show That the new top stack frame is well-formed and that the updated heap H' is well-formed. Because, the statements $\overline{\texttt{stmt}}$, the local store L_c, and the invocation context I_c have not been changed since they were the top frame, they are still well-formed when they become the top frame again. That the calling context adheres to the required types is guaranteed by caller-has-expected-type. What is left to show is that H' is well-formed. This is trivial because the updated binding points to a well-formed object with the same static type as the original $\texttt{Cn}_{jit}$.

We showed for each of the 23 reduction rules that applying that rule to a well-formed configuration always yields a new well-formed configuration. Hence, we have proved *preservation*. □

We have proved both *progress* and *preservation* for our type system, hence, we have proved *soundness*. □

5.6 JIT$\Delta\sigma$ with Single Inheritance

In this chapter we presented JIT$\Delta\sigma$, a formal specification of JITds, based on Lightweight Java. JIT$\Delta\sigma$ extends Lightweight Java with support for representation changes and we show how we can guarantee that a well-formed program never results in a "method not found" exception and accessing a field never results in a "field does not exist" exception. However, in order to be able to focus on the peculiarities that that stem from JITds's representation changes, we do not model *single inheritance*, a key feature of object-oriented programming languages in general, and Java (derivates) specifically.

In this section we sketch the impact on JIT$\Delta\sigma$ if single inheritance were to be included. All subclass relations that are possible in JIT$\Delta\sigma$ are shown in black in figure 5.24, i. e., either class is a simple class in isolation (e. g., `C`), or a class is simple class that is used as an representation of a just-in-time class (e. g., `R1`, `R2`, and `R3`), or a class is a just-in-time class (e. g., `T`).

Now, if we want to model single inheritance, we first introduce the keyword `extends` as it is known from Java. Then in combination with `combines`, a class definition can either extend one other class, combine multiple *simple* classes, or neither. For completeness, we then have to redefine what simple classes are. First, simple classes as introduced earlier in this chapter (i. e., classes that neither combine nor extend another class) are still simple classes. Furthermore, we consider a class that extends another simple class to be a simple class as well (recursive). With this new definition in mind, figure 5.24 shows all possible variations of the subclass relation in an extension of JIT$\Delta\sigma$ where single inheritance is allowed, i. e., these new relations are coloured (■). There is one variation of the subclass relation that seems to be missing in figure 5.24, that is where a representation (directly or indirectly) combines multiple classes. This relation, however, is not allowed because only simple classes can be combined, and the new definition of simple class forbids this construction.

With these new subclass relations, we need to redefine the auxiliary functions for method look-up and field look-up. Concretely, the class hierarchy needs to be taken into account as well. Strniša et al. [76] show how this can be done. We now sketch for each

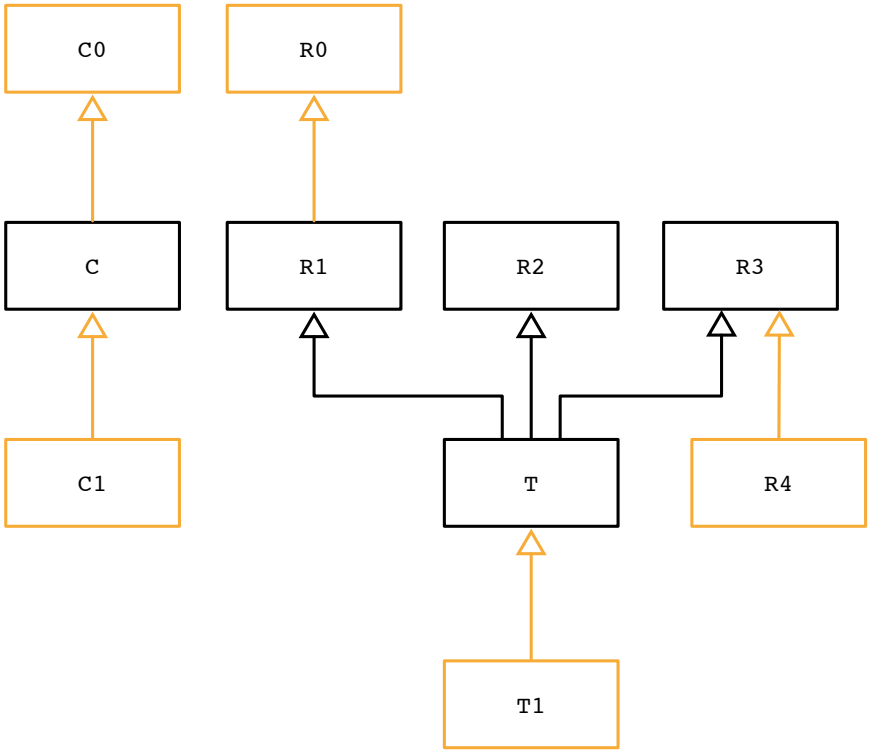


Figure 5.24: A class hierarchy with single inheritance.

of the new variants of the subclass relation how this influences the properties shown for JIT$\Delta\sigma$ without inheritance.

`C extends C0` and `C1 extends C` The three classes `C0`, `C`, and `C1` form a classic single inheritance class hierarchy as it is known in vanilla Java. Strniša et al. [76] have already shown that such hierarchies are type safe.

`R1 extends R0` When we consider `R0` and `R1` in isolation, we can rely on the work of Strniša et al. [76] to assume that this hierarchies is type safe. If we also consider `T` in the hierarchy, then we have a just-in-time class `T` that combines representations of which (at least one) has a parent class. The methods (and fields) in `R1` are then not only those actually defined in `R1`, but also those defined in `R0`, i. e., classic single inheritance. The target of a swap statement can only be `R1`, `R2`, or `R3`, as defined in this chapter and in chapter 4. Hence, as far as the behaviour of the just-in-time class `T` is concerned, the class `R1` could as well have been a class without the parent `R0`, but with all its functionality. For a type point of view, however, `T` is a subtype of `R0`. This is fine since `R1` is a subtype of `R0` as well (transitivity).

`R4 extends R3` When we consider `R0` and `R1` in isolation, we can rely on the work of Strniša et al. [76] to assume that this hierarchies is type safe. If we also consider `T` in the hierarchy, then we have to conclude that whether `R4` is a subclass of `R3` does not influence the behaviour or type of the just-in-time class `T`.

`T1 extends T` When we consider the hierarchy of `T1` and `T`, we rely on the work of Strniša et al. [76] to conjecture that this hierarchy is type-safe as well. First, as long as only behaviour implemented in `T1` is used, all is as if `T1` was a simple class. When behaviour that is not implemented in `T1` is used, then the behaviour of `T` is used. The behaviour of an instance of `T` is studied in depth in this chapter and is proved to be type-safe. In the absence of any counter argument, we conjecture that in general this hierarchy is type-safe.

Neither of the four cases of single inheritance reveals an argument that contradicts our conjecture that adding single inheritance to JIT$\Delta\sigma$ would entail a major improvement in completes or would entail a significant risk of rendering the insight gathered in this chapter invalid. Hence, we remain convinced of our choice to omit single inheritance in JIT$\Delta\sigma$, in favour of compactness.

5.7 Conclusion

This chapter adds two new insights. First, the formal specification of JITds reveals, besides specialised swaps, a second kind of implicit representation change in JITds's semantics, i. e., upon returning from a method invocation or a transition function. Second, we can guarantee that in a well-formed JITds program invoking a method never results in a "method not found" exception (cf. "message not understood" in SmallTalk) and accessing a field never results in a "field does not exist" exception. The only exceptions that can occur are **`NPE`** and **`USE`**. We briefly revisit these two key observations.

5.7.1 Implicitly Changing the Representation of the Caller

When introducing JITds in chapter 4, we explain how *specialised swaps* are *implicit representation changes* that occur when a method is invoked for which the current representation does not provide an implementation. The reduction rules R-MR-INDIRECT and R-TFR-INDIRECT reveal that there is a second kind of *implicit representation change* present in JITds.

A sequence of statements in JITds is known to be statically-typed in the context of a method body (i. e., well-formed).[5] To guarantee that the remainder of the statements are still well-formed after executing the first statement, we have to make sure that the pseudo-variable **`this`** still has the same representation (i. e., dynamic class). This cannot be guaranteed in the case that **`this`** refers to a just-in-time object whose representation has changed during the execution of a method invocation. Hence, we added the reduction R-MR-INDIRECT to detect and resolve this case.

[5] For simplicity, we ignore transition function bodies here, but the argumentation holds for those as well.

5.7.2 A Case in favour of Unsupported Swap Exceptions

By proving the soundness of the type system we defined for JIT$\Delta\sigma$, we can guarantee that no well-formed JIT$\Delta\sigma$ program will ever get stuck during the execution. However, it is possible that a well-formed program reduces to an exception configuration. The configuration with a **`NPE`** is known from Java and its derivatives. The configuration with an **`USE`** is unique to JIT$\Delta\sigma$.

Reaching a configuration with an **`USE`** means that the program has requested a representation change, but the needed transition functions are absent. If the well-formedness rule of a just-in-time class would require all possible transition functions to be defined (i. e., from all representations to all other representations) no **`USE`** could ever occur. This is true because all requested representation changes are statically guaranteed to occur between representations of the expected just-in-time class.

Section 4.6 explains that it is a well-considered design choice in JITds to *not require* all possible transition functions to be defined. One of the arguments is that some representation changes are unwanted (e. g., opening a locked file). Thus when an **`USE`** is encountered, this can mean one of two things: either the developer forgot to implement a transition function (developer's error) or the program requested a representation change that was deliberately unsupported. In the latter case, the developer can build exception mitigating code to resolve the issue.

Chapter 6

Designing Just-in-Time Data Structures: When to Change the Representation

In chapter 4 we introduced JITds, a language that allows developers to define data structures that can change their representation during the execution of a program. To initiate a representation change, JITds introduces the *swap statement*. However, in the resulting code, *application logic* and *representation change logic* are entangled. Both sections 2.1 and 2.2 argue in favour of separation of application logic and representation change incentive code. This implies the need for language features that allow a developer to disentangle representation change incentive code and application logic.

In this chapter we introduce and discuss *swap rules*. Swap rules are the constructs in JITds that allow the developer of a just-in-time data structure to express **when** a representation change is needed. In general, a swap rule is a member of a just-in-time data structure that encapsulates representation change incentive code. Such a swap rule makes quantified statements about which representation change is to be executed in which circumstances. Less formally, swap rules express, based on the *observed* usage of a just-in-time data structure, a *reaction* in the form of a transition from one representation to another.

Concretely, this chapter is organised as follows. First, we identify the need for three kinds of swap rules, which we introduce sections 6.1.1 to 6.1.3, respectively. Further, we show how to broaden the applicability of swap rules. In section 6.2, we show how to define *history based swap rules*, and in section 6.3, we show how to define *(machine) learned swap rules*. Finally, in section 6.4 we revisit the idea of swap rules as a kind of *domain specific aspects*.

6.1 Swap Rules

Swap rules are the constructs in JITds that allow developers to express representation change incentives without entangling them with the application's base logic. From chapter 3 we know that there are two kinds of representation change incentive. Either a representation change is imposed on a data object from the outside, or it is initiated by

the data object itself. Below, we revisit two examples from section 2.2 that illustrate the difference between those two variants – internal and external (see chapter 3) – of representation change incentive. We also discuss a third example that reveals the need for a hybrid from of representation change incentive.

A first matrix example. In section 2.2 we showed that a performance-aware programmer prefers to invoke the matrix multiplication method `mul(Matrix a, Matrix b)` with arguments of the `RowMajorMatrix` and `ColMajorMatrix` representation, respectively. Moreover, the experiments in section 2.2.3 reveal that paying the cost of changing the representation at runtime to match the expected representations can be beneficial for performance as well. Listing 2.17 on page 37, for instance, shows how this preference of `mul` can be expressed. Here, `mul` is the "user" of two matrix data objects, and thus the code in `mul` to express that some representation changes are needed is external representation change incentive code. In JITds, it is possible to express this kind of external representation representation change incentive by means of an *external swap rule*. These external swap rules are introduced in section 6.1.1.

A second matrix example. In section 2.2 we write that a performance-aware programmer prefers a `SparseMatrix` representation when the number of non-zero values is low. Further in section 2.2, we provide the example of a stencil computation where the developer is not necessarily aware that a matrix has become sparse. A representation change from a dense representation to `SparseMatrix` should thus be initiated by the matrix itself, i. e., by means of internal representation change incentive code. In JITds, it is possible to express this kind of internal representation representation change incentive by means of an *internal swap rule*. These internal swap rules are introduced in section 6.1.2.

A third matrix example. A diagonal matrix has only elements on its main diagonal. There exist representations for the matrix ADT that rely on this property to store the elements of such a matrix more efficiently. Appendix C, for instance, provides an implementation of `DiagonalMatrix`, which stores all elements in an array with the length of the main diagonal. This is sufficient since non-zero values only occur on the main diagonal in a diagonal matrix. Hence, when a user of such a matrix tries to set a non-zero value outside of the diagonal, such a representation usually throws an exception. Alternatively, the matrix could decide that it is more appropriate to transition to a representation that does support non-zero values outside the diagonal. In JITds, it is possible to express this kind of internal representation change incentive by means of an *interface swap rule*. This third kind of swap rule is introduced in section 6.1.3.

6.1.1 External Swap Rules

External Swap Rules are swap rules that express the need for a representation change that is initiated by the *user* of a just-in-time data structure. It is thus a piece of code that augments a computation with the preferred representations of the data objects this

computation uses. In object-oriented languages, methods form the most straightforward unit of computation. Therefore we restrict ourselves to *method invocations* as join points at which to introduce representation changes. The matrix multiplication method `mul`, for instance, is a computation for which we know the preferred representations of its arguments (see section 2.2).

Thus, an external swap rule needs to do two things. First, it has to define to which method the swap rule applies; and second, it has to express what representation changes have to take place when that method is invoked. The implied relation with aspect-oriented programming and static quantification is discussed in section 6.4.

Syntax: External Swap Rule

```
class T combines R0, ... , Rn {
  swaprule Cr C.mn(C0 a0, ... , Cn an) {
    ...
    proceed;
    ...
  }

  swaprule Cr C.mn(C0 a0, ... , Cn an) {
    ...
    Cr result = proceed;
    ...
  }
}
```

With `C` different from the defining just-in-time class `T` class or any of its representations $\texttt{R}_i$.

First, an external swap rule is a *member* of a just-in-time class (i. e., `T` in the syntax description). Apart from the keyword `swaprule`, an external swap rule looks a lot like a method definition. Hence, we use the same terminology and say that an external swap rules has a *signature* and a *body*.

The header of an external swap rule defines to which method the swap rule applies. Recall that a method is uniquely defined by its name, the types of its formal parameters, its modifiers, and its defining class. The header of an external swap rule is therefore an exact copy of the header of that method `md`[1], qualified with the class `C`, in which it was defined. Here `mn` from `C` which has the type $(\texttt{C}_0, \ldots, \texttt{C}_n) \rightarrow \texttt{C}_r$. Note that in order to define an *external* swap rule, the method can not be defined in the just-in-time class or any of any of its representations. In this case this means that `C` can not be `T` or any of $\texttt{R}_i$. Swap rules where `C` is `T`, or any of $\texttt{R}_i$, are discussed in section 6.1.3.

The body of an external swap rule consists of three parts: a set of statements, a `proceed` statement, and another set of statements. 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 captured method. It is also possible to access the

[1] The return type is not strictly necessary.

Listing 6.1: An external swap rule that matches the `mul` method of the class `MatrixUtils`, and which ensures that the arguments adhere to the optimal representations

```
1  swaprule static Matrix MatrixUtils.mul(Matrix a, Matrix b) {
2    if ((a.getRows()*a.getCols()) > LARGE)
3      a to RowMajorMatrix;
4    if ((b.getRows()*b.getCols()) > LARGE)
5      b to ColMajorMatrix;
6    proceed;
7    /* do nothing */
8  }
```

Listing 6.2: Swapping a swappable Data Structure

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

result value of the actual method invocation (cf. second case in the syntax box above), e. g., to perform a swap statement. However, it is not possible to change the actual return *value*.

Example. Listing 6.1 shows an external swap rule for the just-in-time class `Matrix` as presented in listing 4.5. This swap rule captures all the invocations of the static `mul` (matrix multiplication) method defined in `MatrixUtils`. Further, the swap rule specifies that, when the matrices are "large enough" (cf. the constant `LARGE`), their representations should be `RowMajorMatrix` and `ColMajorMatrix`, respectively. Consequently, an invocation of `checkCommutativity` (also in `MatrixUtils`, see listing 6.2), which calls the `mul` method twice, could potentially introduce four representation changes.

6.1.1.1 Receiver and `this`

An external swap rule is a *static* member of a just-in-time class. This means that it is associated with the just-in-time class itself rather than with a concrete just-in-time object. Hence, **`this`** in the body of an external swap rule does not refer to a just-in-time object. The alternative, **`this`** as keyword to denote the receiver of the instrumented method invocation, introduces ambiguity. We defer the detailed discussion to section 6.1.4. Here, we introduce the keyword `receiver` to denote the receiver of the method instrumented by an external swap rule.

Syntax: Accessing the receiver in an External Swap Rule

```
swaprule C_r T.methodName(C_0 a_0, ... , C_n a_n) {
  receiver to R_i;
  proceed;
}
```

With `T` a just-in-time class, and R_i one of its representations.

6.1.2 Internal Swap Rules

Internal Swap Rules are swap rules that express the need for a representation change initiated by a just-in-time data structure *itself*, based on its *current state*. For an instance of `RowMajorMatrix`, for instance, the *size* is an example of state, i. e., is the product of `rows` and `cols`. Note that state, in this context, is not limited to member fields. The *number of active threads*, for instance, is another example of state.

An internal swap rule is a piece of code that augments the state of a data object (in the broadest sense of the word, as described above) with the preferred representation for that state. For a `SparseMatrix`, for instance, it is better to swap to one of the dense representations when the ratio of non-zero values to the size of the matrix exceeds some constant `T`. The implied relation with aspect-oriented programming and dynamic quantification is discussed in section 6.4.

Syntax: Internal Swap Rule

```
class T combines R_0, ... , R_n {

  swaprule R_from {
    ...
    if ( ... ) {
      this to R_to;
    }
  }

}
```

Where both R_{from} and R_{to} are representations of the defining just-in-time class.

An internal swap rule starts with the keyword `swaprule`, just like an external swap rule. In an internal swap rule, however, the `swaprule` keyword is followed by the representation R_{from} to which this internal swap rule applies. We call this representation R_{from} the *source representation*. The body of an internal swap rule is a set of statements between curly braces. All statements in the body have access to the just-in-time object through the use of **`this`** as if the swap rule were an ordinary method of that object. Consequently, the statements (e. g., an `if`-statement) can be used to check for any condition. The intent of an internal swap rule is to invoke a representation change when a

certain condition is met. Hence, a typical internal swap rule has a body, similar to listing 6.3, with a condition (line 3) and a swap statement (line 4), which performs the actual representation change.

Listing 6.3: Internal swap rule to `SparseMatrix` based on estimated sparsity.

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

Example. Listing 6.3 shows an internal swap rule from the just-in-time class `Matrix` as presented in listing 4.10. This swap rule can only trigger when the matrix is in the `SparseMatrix` representation. The rule in listing 6.3 states that the data structure should swap to the `RowMajorMatrix` representation when the number of zeroes drops below a threshold `SPARSITY_FACTOR`.

6.1.2.1 Type Safety

The header of an internal swap rule denotes the representation to which that swap rule applies. Hence, in the body of an internal swap rule that applies to the representation `Cn`, one is guaranteed that **`this`** refers to a just-in-time object with `Cn` as its current representation. Moreover, in the body of an internal swap rule, **`this`** is treated as a simple object, of the current representation's type, instead of a just-in-time object. This has two advantages. Firstly, it provides access to the members (i. e., both fields and methods) of the current representation without ambiguity because the current representation is known. Secondly, it prohibits the use of methods that are a specialisation of another representation. The latter avoids specialised swaps during the execution of an internal swap rule, which would be unwanted behaviour.

Example. Recall from section 4.5.1, that the methods available in a just-in-time object can be partitioned into core methods and specialised methods, depending on whether they are provided by *all* representations (core) or by a subset of the representations (specialised). On line 2 of listing 6.3, the size of the matrix is computed by multiplying the number of rows and columns in the matrix. This value is assigned to the variable `size`. The number of rows and columns are retrieved though calls to `getRow` and `getCols`, respectively. Both operations are part of `Matrix`'s core. `getNonZeroCount()`, on the other hand, is not part of the core of `Matrix`. However, since it is a member of the set of specialised methods of `SparseMatrix` it is type safe to invoke this method. This is true because we know that the matrix is in the `SparseMatrix` representation at the moment the body of the swap rule is executed. A call to `getArray` (i. e., a specialised method of `ColMajorMatrix` and `RowMajorMatrix`) would not be allowed.

6.1.2.2 Performance Issues

An internal swap rule defines a set of conditions for which a developer identified that when these conditions hold, it is opportune to initiate a representation change. Conceptually, we expect the language runtime of JITds to continuously check whether the conditions hold. This approach, however, might not be optimal in practise[18]. For instance, our current implementation — as we discuss in chapter 7 — executes the bodies of all internal swap rules after each method invocation. Finding the optimal balance between responsiveness and performance, however, has not yet been investigated but is discussed as future work in chapter 10. Until then, the rule of thumb when designing internal swap rules is thus to keep them as efficient as possible, e. g., by relying on readily observable state only.

6.1.3 Interface Swap Rules

Interface Swap Rules are swap rules that initiate a representation change when a certain method of the just-in-time object is invoked — external incentive — but only when the just-in-time data structure has a certain state — internal incentive. Interface swap rules are thus a hybrid between internal and external swap rules. Again, the relation to aspect-oriented programming and static quantification is discussed in section 6.4.

Syntax: Interface Swap Rule

```
class T combines R0, ... , Rn {

  swaprule Cr R.methodName(C0 a0, ... , Cn an) {
    ...
    Cr result = proceed;
    ...
  }

  swaprule Cr methodName(C0 a0, ... , Cn an) {
    ...
    proceed;
    ...
  }

}
```

With R as one of the representations of the defining just-in-time class.

The syntax of an interface swap rule is roughly equivalent to that of an external swap rule, as it matches the signature of a method. The method name in an interface swap rule is either fully quantified with the name of a specific representation, or is not quantified at all. The semantics of an interface swap rule correspond to that of an external swap rule. The statements before `proceed` are executed before the actual method invocation and the statement after `proceed` are executed after the actual method invocation. In the body of an interface swap rule — as opposed to external swap rules and similar to internal swap rules — one can refer to **`this`**, which denotes the receiver of the method

Listing 6.4: The class `Matrix` combines four representations.

```
1  class Matrix
2  combines RowMajorMatrix, ColMajorMatrix, CRSMatrix , DiagonalMatrix {
3
4    DiagonalMatrix to Matrix {
5      target( source.getRows(), source.getCols() );
6      for (int i=0 ; i<source.getRows() ; i++) {
7        target.set(i, i, source.get(i, i));
8      }
9    }
10
11   Matrix to Matrix {
12     target( source.getRows(), source.getCols() );
13     for (int r=0 ; r<source.getRows() ; r++) {
14       for (int c=0 ; c<source.getCols() ; c++) {
15         target.set(r, c, source.get(r, c));
16       }
17     }
18   }
19
20   swaprule void DiagonalMatrix.set(int row, int col, double val) {
21     if ( (row!=col) && (val!=0) ) {
22       this to CRSMatrix;
23     }
24     proceed;
25     /* do nothing */
26   }
27
28 }
```

invocation which is a just-in-time object, to invoke methods on. Note that **`this`** can be only be used to access fields when the interface swap rule is quantified with a representation.

Example. Listing 6.4 shows an implementation of a just-in-time class `Matrix`, which combines four different matrix representations. The first two representations have been introduced in previous examples. The third and fourth representation, i. e., `CRSMatrix` (cf. compressed row storage (CRS)) and `DiagonalMatrix`, are discussed in detail in appendix C. For this example, it is sufficient to know that both `CRSMatrix` and `Diagonal-Matrix` are *sparse matrix* implementations and that `CRSMatrix` can be used to store any matrix, whereas `Diagonal-Matrix` is only suitable for matrices where the non-zero values occur only on the main diagonal.

Now consider the usage of such a just-in-time `Matrix` that has `DiagonalMatrix` as its initial representation. To simplify the example shown in listing 6.5, we assume our matrices to be square. The first three write operations (lines 3-5) are executed as expected, since they represent updates of the diagonal of `m`. The write operation on line 7, however, tries to set a non-zero value at position $(0,1)$ which is not part of the main diagonal. When executed by a `DiagonalMatrix` (see line 22 in listing C.2 in ap-

Listing 6.5: Setting values of a `Matrix` in the `DiagonalMatrix` representation.

```
1  Matrix m = new Matrix.DiagonalMatrix(3,3);
2
3  m.set(0, 0, 1);
4  m.set(1, 1, 2);
5  m.set(2, 2, 3);
6
7  m.set(0,1,6);
```

pendix C) this invocation of `set` throws an exception because it is not supported by the `DiagonalMatrix` representation.

A user of a general purpose `Matrix`, however, does not expect a valid call to `set` (i. e., row and col within bounds) to throw an exception. The interface swap rule on lines 20-27 in listing 6.4 prevents the unwanted behaviour described above. Concretely, this swap rule triggers when a `Matrix` that is in the `DiagonalMatrix` representation has to execute a non-zero `set` outside its main diagonal (line 21). This interface swap rule effectively avoids the runtime exception from being thrown, by changing the representation to `CRSMatrix`, which does allow non-zero values outside the main diagonal.

6.1.4 Scoping Rules for Swap Rules

In this chapter we introduce three kinds of swap rules, i. e., *external swap rules* in section 6.1.1, *internal swap rules* in section 6.1.2, and *interface swap rules* in section 6.1.3. We show how internal swap rules and interface swap rules have access to **`this`**, which denotes a just-in-time object, to access and modify instance fields. Here, we discuss what **`this`** denotes in the context of an external swap rule.

We can think of two sensible meanings for **`this`** in the context of an external swap rule, and both have their pros and cons. On the one hand **`this`** can denote the just-in-time class of which it is a member. On the other hand, **`this`** can denote the receiver of the method matched by the external swap rule.

Lexical uniformity is one argument in favour of **`this`** to refer to the just-in-time class in which it resides. An external swap rule is a member of a just-in-time class and hence denoting that just-in-time class by **`this`** conforms to the other usages of **`this`** in a just-in-time class, e. g., internal swap rules, interface swap rules, and bypass methods. On the other hand, an external swap rule is a *static* member of a just-in-time class, and thus does not have a just-in-time object associated with it at runtime. Thus, **`this`** would refer to the just-in-time class instead of a just-in-time object as in internal swap rules for instance. Hence, the lexical uniformity argument does not really apply. A second downside of referring to the just-in-time class by **`this`**, is that, to access the receiver of the invoked method, a new keyword has to be introduced (cf. `target` in AspectJ[81]).

For clarity and complete lexical uniformity, we do not allow **`this`** in the body of an external swap rules, i. e., not to access the receiver of the invoked method and not to access the class . As introduced in section 6.1.1.1, we opt to introduce the keyword `receiver` as the pseudo-variable which can be used to refer to the receiver of the in-

voked method. The just-in-time class can be accessed *by name* (cf. as in Java). All scoping rules to access values in swap rules are summarised in table 6.1.

	Method invocation		Fields and Methods	
Swap Rule	Receiver	Arguments	Just-in-time class/object	
External Swap Rule	`receiver`	by name	by name	class
Internal Swap Rule	n.a.	n.a.	**`this`**	instance
Interface Swap Rule	**`this`**	by name	**`this`**	instance

Table 6.1: Scoping Rules for Swap Rules.

6.2 History Based Representation Changes

In JITds, representation change incentives can be separated from application logic by implementing external swap rules, internal swap rules, or interface swap rules. The example swap rules all show representation change incentives based on readily observable state: the invocation of the `mul` method with "large" arguments in section 6.1.1, the number of non-zero values in section 6.1.2, and the invocation of `set` in section 6.1.3. Orthogonal to the choice of implementing external, internal, or interface swap rules, it can be interesting to base the decision of a representation change not only on readily observable state, but also to take the history — of execution or state — into account.

In this section we show how swap rules become more expressive when they take the past into account. We revisit the example of the matrix that can be either sparse or dense, which relates to using the `SparseMatrix` or the `RowMajorMatrix` representation, respectively. We show, by example, that for a matrix in a dense representation it is hard (i. e., expensive or inaccurate) to determine whether it has become a sparse matrix and thus should change its representation, when only readily observable state is taken into account. Conversely, when the past is taken into account, swap rules can become more expressive.

Too Expensive. The internal swap rule in listing 6.6 converts a `RowMajorMatrix` into a `SparseMatrix` when the number of non-zero values drops below a certain threshold (`nonZeroCounter < (size/T)`). Computing the actual sparsity for a densely stored matrix, however, is probably too expensive to check regularly.

Too Inaccurate. The alternative is a representation change based on a *heuristic* to compute sparsity. A straightforward heuristic is verifying if `set` is invoked with the argument `value` equal to 0. This can be implemented in JITds using an interface swap rule such as in listing 6.7. A representation change comes at a certain cost. Therefore, it is not always economical to change the representation eagerly. The heuristic used in listing 6.7 is not a very good heuristic. Swapping from `RowMajorMatrix` to `SparseMatrix` on the first call to `set` with a zero value is counterproductive if that "zero set" is a lone wolf.

Listing 6.6: Internal Swap Rule to convert a dense matrix into a `SparseMatrix`.

```
1  class Matrix combines RowMajorMatrix, SparseMatrix {
2
3    swaprule RowMajorMatrix {
4      int size = this.getRows() * this.getCols();
5      int nonZeroCounter = 0;
6      for (int r=0 ; r<this.getRows() ; r++) {
7        for (int c=0 ; c<this.getCols() ; c++) {
8          if ( this.get(r,c) != 0) nonZeroCounter++;
9        }
10     }
11
12     if ( nonZeroCounter < (size/T) ) {
13       this to SparseMatrix;
14     }
15     proceed;
16   }
17
18 }
```

Listing 6.7: Interface Swap Rule to convert a dense matrix into a `SparseMatrix`.

```
1  class Matrix combines RowMajorMatrix, SparseMatrix {
2
3    swaprule RowMajorMatrix.set(int row, int col, double value) {
4      if ( value == 0 ) {
5        this to SparseMatrix;
6      }
7    }
8
9  }
```

The use case presented above would benefit more from a swap rule that does not only take the readily observable state into account, but also the state observed in the past. Here, the past denotes “in previous executions” and “earlier in the execution”. To facilitate the bookkeeping of historical information, as introduced in section 4.4, swap rules have access to fields of a just-in-time class. Concretely, external swap rules have access to statically defined member fields in the just-in-time class and internal swap rules and interface swap rules have access to instance member fields of a just-in-time class. It is possible to use these fields to store information of previous swap rule invocations.

Example. Listing 6.8 shows a new version of the earlier interface swap rule, which now also takes the history of previous calls to `set` into account. Every invocation of the `set` method, either increments the `zeroSet` counter of the `nonZeroSet` counter. The new interface swap rule does not cause a representation change upon the first occurrence of a set of a zero value. Conversely, only when the number of sets of zero values greatly exceeds the number of sets of non-zero values (cf. quadratic, line 11), a representation

Listing 6.8: Interface Swap Rule to convert a dense matrix into a `SparseMatrix` (v2.0).

```
1  class Matrix combines RowMajorMatrix, SparseMatrix {
2
3    int zeroSet    = 0;
4    int nonZeroSet = 0;
5
6    swaprule RowMajorMatrix.set(int row, int col, double val) {
7      if ( val == 0 )    zeroSet++;
8      else            nonZeroSet++;
9
10     if ( ((zeroSet+nonZeroSet) > FREQUENT_SET) &&
11          (zeroSet > nonZeroSet*nonZeroSet) )
12       this to SparseMatrix;
13   }
14
15 }
```

change is issued. The swap rule defined in listing 6.8 is still an estimate and hence not perfect. Compared to the swap rule defined in listing 6.7, however, it is more accurate in predicting whether a matrix has become sparse enough to change the representation to `SparseMatrix`.

6.2.1 Invocation Counters

This last example (listing 6.8) shows how to make use of instance member fields to store the history of interface invocations, i. e., both quantitative (counting) and qualitative (with constraints). The need for similar information to decide whether or not to issue a representation change is also identified by Shacham et al. [69], i. e., *opCounts*, and by Xu [90], i. e., *swap conditions*. Moreover, in all our example programs (see chapter 8) interface swap rules are only used to perform such counts. To make counting the number of invocations of member methods easier, we introduce *invocation count expressions*.

Syntax: Invocation Counters

```
#methodName( T0 a0, ... , Tn an );

#R.methodName( T0 a0, ... , Tn an );

#methodName( T0 a0, ... , Tn an ) {
  ...
  count-if ( ... );
}

#counterName as methodName( T0 a0, ... , Tn an );
```

In its simplest form, an invocation counter is a hash-symbol (cf. “number of”) fol-

Listing 6.9: Internal swap rule to `SparseMatrix` based in estimated sparsity.

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

lowed by a method name and a list of formal parameters between braces. Optionally, the method name is quantified with a specific representation. Such an invocation counter captures exactly one method from the just-in-time data structure's interface. The value of an invocation counter is equal to the number of invocations of the captured method since the last representation change. The counters can be refined by adding a body. In this body the `count-if` statement determines whether or not the invocation counter is incremented, e. g., depending on the actual arguments. Just like a regular `if` statement, the `count-if` statement has a condition, i. e., the expression between parentheses. However, a `count-if` statement does not have a consequent, or an alternative. Only when the `count-if` statement is reached and its condition evaluates to `true`, is the counter incremented. Otherwise, when the condition evaluates to `false`, the counter is left unchanged. Optionally, using the `as` keyword, an invocation counter can be given a more revealing name.

Example. In listing 6.9, we reimplement the dense-to-sparse example. This time we define three invocation counters and an internal swap rule. The first invocation counter (line 3) `#set` counts all invocations of the `set` method. The other two invocation counters (lines 5 and 7) also count the number of invocations of `set`, but, as their bodies reveal, they only get incremented when `val` is zero or non-zero, respectively. Finally, the internal swap rule expresses the heuristic used in section 6.2: when the number of set of zero values greatly exceeds the number of set of non-zero values, it is beneficial to swap to the `SparseMatrix` representation.

Intermezzo: Sample-average Method with the Decreasing Epsilon Greedy Strategy

Q-learning is a reinforcement learning technique which tries to learn the best action for a given state [80]. A simpler version of this problem is the multi-armed bandit problem where only one state exists. In general, Q-learning constructs an action-value function that predicts the expected cost of taking a given action in a given state. One of the simplest methods for estimating the action-value function is to compute the average cost of each action. When this function is stable, the optimal behaviour is to perform the action with the lowest expected cost. To balance the trade-off between exploring all possible actions and exploiting the actions that are known to have low costs, a strategy such as the *Decreasing Epsilon Greedy Strategy* can be used. This strategy states that every time a decision has to be made, the best, and thus the cheapest, action is chosen with high probability (cf. greedy). To cope with variability in observed costs, a factor of randomness is introduced to ensure exploration of estimated suboptimal actions. With a probability of $|\epsilon|$, a random action is chosen (cf. epsilon). To stabilise the system the value of ϵ is decreased over time (cf. decreasing). The result is a selection strategy that exhibits a lot of explorative behaviour at first and high exploitative behaviour later.

Listing 6.10: A straightforward implementation of the Decreasing Epsilon Greedy Strategy in Java.

```
1  public class DecreasingEpsilonGreedyStrategy {
2
3    int actions;
4    double epsilon = 1.0;
5
6    int[]      cost, frequency;
7
8    public DecreasingEpsilonGreedyStrategy(int actions) {
9      this.actions = actions;
10
11     costs     = new int[actions];
12     frequency = new int[actions];
13   }
14
15   public long relativeCost(int action) {
16     return  costs[action]/frequency[action];
17   }
18
19   public int greedy() {
20     int bestAction = 0;
21
22     for (int action=1 ; action<actions ; action++) {
23       if ( relativeCost(action) < relativeCost(bestAction) )
24         bestAction = action;
25     }
26
27     return bestAction;
28   }
29
30   public int next() {
31     if ( rand()<epsilon ) {
32       epsilon = epsilon * COOL_DOWN;
33       return r.nextInt(actions);
34     } else {
35       return greedy();
36     }
37   }
38
39   public void update(int action, int cost) {
40     costs[action]+=cost;
41     frequency[action]++;
42   }
43
44 }
```

Listing 6.11: The representations of a just-in-time class can be seen as an enumeration type.

```
1  class T combines R0, ..., Rn {
2    public static enum Representation { R0, ..., Rn }
3  }
```

6.3 Learning Representation Changes.

> Those that fail to learn from history, are doomed to repeat it. — Winston Churchill

History-based decisions for representation changes naturally lead to *learning representations changes*. Since actual machine learning techniques fall outside the scope of this text, we only give a gentle introduction of how *computing the average cost* and *the Decreasing Epsilon Greedy Strategy* work in the intermezzo on page 136. For now, we assume the existence of `DecreasingEpsilonGreedyStrategy`, a class which provides a method `next` that is able to **learn** the best "action" to perform next. In the context of just-in-time data structures this translates to "learning which representation change(s) improve(s) performance given that the initial representation(s) is/are known".

6.3.1 First Class Representations

JITds introduces the built-in function `representation`, which retrieves the current representation of a just-in-time object. For a simple object this function is not specified. The number of representations is finite and statically known and can thus be seen as the value of the enumeration type `T.Representation` which implicitly exists for each just-in-time class (see listing 6.11). The type of this enumeration is written as `T.Representation`, i. e., `T` qualified with `Representation`. Hence, defining a new just-in-time class `T` results in the definition of two new types: the just-in-time class `T` itself and the representation type `T.Representation`.

Syntax: Representation (built-in function)

```
T.Representation r = representation( o );
```

In the next section, when introducing listing 6.13, it becomes clear that *representations are first class* in JITds. Firstly, we show that representations can be assigned to variables (lines 6-7 and 10-11). Secondly, we show that representations can be the arguments of a method invocation (lines 9 and 22). And thirdly, we show that they can be the result of a method invocation (line 10-11). Hence the representations of a just-in-time class are first class values in JITds. Moreover, these representation-values can be used in swap statements as well, e. g., lines 15 and 16.

Listing 6.12: `MatrixMultiplicationLearner`

```
1  public class MatrixMultiplicationLearner extends DecreasingEpsilonGreedyStrategy {
2
3    int f(Matrix.Representation left, Matrix.Representation right) {
4      int leftI  = (left==RowMajorMatrix)?0:1;
5      int rightI = (left==RowMajorMatrix)?0:1;
6      return (2*leftI) + rightI;
7    }
8
9    final static Pair<Matrix.Representation>[] pairs = new Pair(){
10     new Pair(RowMajorMatrix,RowMajorMatrix), /* 0 */
11     new Pair(RowMajorMatrix,ColMajorMatrix), /* 1 */
12     new Pair(ColMajorMatrix,RowMajorMatrix), /* 2 */
13     new Pair(ColMajorMatrix,ColMajorMatrix)  /* 3 */
14   };
15
16   Pair<Matrix.Representation> f(int i) {
17     return pairs[i];
18   }
19
20   public Pair<Matrix.Representation> next() {
21     return f( super.next() );
22   }
23
24   public void update( Matrix.Representation left,
25                       Matrix.Representation right,
26                       long executionTime) {
27     super.update( f(left,right), executionTime )
28   }
29
30   public MatrixMultiplicationLearner(){
31     super(4);
32   }
33
34
35 }
```

6.3.2 An External Swap Rule with Reinforcement Learning

Example. Recall the external swap rule shown in listing 6.1 which forces matrices used in the `mul` computation to adhere to the `RowMajorMatrix` and `ColMajorMatrix` representation for the left operand and the right operand of the multiplication, respectively. Note that the swap rule is based on expert knowledge acquired by rigorously studying the performance characteristics of the matrix multiplication algorithm (cf. chapter 2). If we did not have this knowledge, we could use machine learning to acquire such knowledge.

First, note that there are four possible representation states when `mul` gets called, i. e., all combinations of `RowMajorMatrix` and `ColMajorMatrix`. Consequently, there are also four possible representation states after any representation change, i. e., again the same combinations. Listing 6.12 shows an extension of `DecreasingEpsilon-`

Listing 6.13: Learned Reaction to the Occurrence of a call to `mul`

```
1  // Static field:
2  static MatrixMultiplicationLearner l = new MatrixMultiplicationLearner();
3
4  // Learning external swaprule:
5  swaprule Matrix MatrixUtils.mul(Matrix mA, Matrix mB) {
6    Pair<Matrix.Representations> action = l.next();
7    Matrix.Representations newA = action.left();
8    Matrix.Representations newB = action.right();
9
10   long begin = System.currentTime();
11
12   mA to newA;
13   mB to newB;
14
15   proceed;
16
17   long end = System.currentTime();
18
19   l.learn( newA , newB , (end-begin) );
20 }
```

`GreedyStrategy` that — instead of integers — uses the actual representations as states and actions. The conversion from `int` to a pair of matrix representations is implemented by `f` on lines 16, 17 and 18, the conversion of two matrix representation into an integer is implemented by `f` on lines 3–7.

Then, `MatrixMultiplicationLearner` can be used as follows: When `next` is called, it responds with a pair of representations, namely those it has learned that are the best choice for proceeding the multiplication with. For learning purposes, `update` should be called with the chosen representations (the action) and the measured execution time (a measurement of cost). The responsibility of `DecreasingEpsilonGreedyStrategy` is then to learn — and thus improve — the cost functions to give better suggestions later.

Listing 6.13 uses the `MatrixMultiplicationLearner`, as discussed above, in an external swap rule to learn the best representations when computing the product of two matrices. First, the best representations for the matrices `mA` and `mB` are obtained (lines 6–8, cf. actions). Then, the suggested representations are enforced on `mA` and `mB` (lines 12–13) and the multiplication is executed (line 15). Simultaneously, the time to perform the representation changes and the actual computation is measured using the system's clock (lines 13 and 20). Finally, the measured time is used to update the cost functions (line 19). Experiments show that this algorithm learns that **changing** to the combination `RowMajorMatrix` × `ColMajorMatrix` is the optimal choice, just like we established in section 2.2. Details on the performance benefits of using the reenforcement learning algorithm to determine the optimal combination of representations in for example are discussed in chapter 8.

6.4 Domain Specific Aspects

In JITds, swap rules allow us to disentangle application logic from representation change incentive code. Our two implementations of JITds, as we discuss in chapter 7, are responsible for weaving these swap rules into the application logic. This can effectively be realised through aspect-oriented programming as well [49]. AOP is a programming paradigm focussed on separating crosscutting concerns. In JITds, these crosscutting concerns are, on the one hand, the core application logic, and on the other hand, all code that is involved in changing the representation of data objects.

> *AOP is the desire to make programming statements of the form: In programs P, whenever condition C arises, perform action A.*
> ...
> *In an AOP system, we make quantified statements about which code is to execute in which circumstances. Over what can we quantify? Broadly, we can quantify over the static structure of the system and over its dynamic behavior.* — Filman and Friedman [32]

While our work does not focus on AOP as such, it is interesting to position our work in the context of AOP, for instance, by comparing it to the work of Filman and Friedman [32]. A short fragment of their paper is given above. Note that our swap rules match with what they call "quantified statements about which code is to execute in which circumstances". Further note that they identify two kinds of quantifications: quantifications over the static structure of a program, and over its dynamic behaviour. In retrospect, external swap rules and interface swap rules are static quantifications, i. e., the invocation of a method can be lexically determined. Internal swap rules, on the other hand, are dynamic quantifications, i. e., they trigger when a certain condition is met. Note that the invocation of an internal swap rule does not correspond to a lexical place in the code but rather to the dynamic behaviour of the system.

The chapter on related work, chapter 9, does not explicitly cover domain-specific AOP approaches in the context of performance, because these efforts tend to focus on the aspects related to AOP itself instead of on the aspect related to reducing performance. LoopsAj, for instance, introduces expressiveness dedicated to join points for loops [39], which in turn allows parallelisation of the code and improves performance. The focus of Harbulot and Gurd [39] is the study of language constructs to express finer grained join points [49] and not the parallelism.

6.5 Conclusion

JITds is a statically-typed, class-based, object-oriented language which allows objects to change their representation at runtime. We introduce the concept of a just-in-time class which *combines* multiple representations into one new class. Instances of such a class, just-in-time objects, can be instructed to change representations by invoking a swap statement, while transition functions ensure an orderly transitions between the representations.
To separate application logic from representation changing incentive code, JITds supports swap rules. The swap rules are members of a just-in-time class that express for which observed state a representation change is recommended. We identify and introduce three types of swap rules: *external swap rules*, *internal swap rules*, and *interface swap rules*. To facilitate the development of *history-based* representation change incentive code, we also introduce *invocation counters*.
Section 2.2 identifies the need for data structures that can change their representation at runtime. Chapters 4 and 6 introduce the language JITds in sufficient detail to support the claim that JITds facilitates the construction of such data structures. Moreover, chapter 6 shows how JITds facilitates the separation of application logic and representation related logic, which was put forth as a requirement in chapter 2.

Chapter 7

JITds: The Language Implementation

In chapters 4 and 6 we introduced JITds as a language. In this chapter we approach JITds from the language implementation angle and answer the question of how to execute a JITds program.

We describe two implementations of JITds: JITds-Java and JITds-C. The choice for *two* implementations is a pragmatic one. Both JITds-Java and JITds-C are *prototype implementations*. JITds-Java focusses on usability and JITds-C, on the other hand, focusses on completeness. In chapter 8, for instance, we make use of the JITds-Java compiler to generate executable Java programs out of JITds programs. This is interesting because we can make use of all language features and standard libraries of Java from within JITds, and because we can use plain Java programs as the base-line to compare against in terms of performance and code structure. Some legal JITds programs, however, cannot be compiled (correctly) by JITds-Java. To resolve this limitation of JITds-Java, we implemented JITds-C. The limitation of JITds-C is that we did not make the effort to implement the full base language which we got for free in JITds-Java. Table 7.1 summarises the implemented features and properties of each of the two implementations of JITds.

JITds-Java and JITds-C are implemented using the two most popular techniques to implement dynamic reclassification, i. e., *forwarding* and *inlining*, respectively. Before we discuss the two implementations in detail, we first give a high-level description of forwarding and inlining as they are known in the literature. Second, we also introduce an abstract JITds program that we use to explain how both compilers translate a given construct.

7.1 Dynamic Object Reclassification

The key language feature of JITds that has to be implemented is *dynamic object reclassification*. We know, e. g., from the work of Cohen and Gil [13], that there are two classic approaches to implement dynamic object reclassification: *inline reclassification* or *reclassification by forwarding*.

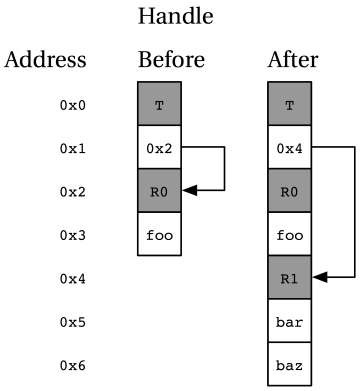
Reclassification by forwarding relies on a handle to point to the object with the current representation. When the object is reclassified, it suffices to update the handle to

	JIT ds	Java JIT ds	C JIT ds
Combining representations	✓	✓	✓
Transition functions	✓	✓	✓
Swap statements	✓	✓	✓
External swap rules	✓	✓	✓
Internal swap rules	✓	✓	✓
Interface swap rules	✓	✓	✓
Invocation counters	✓	✓	✓
By-pass methods	✓	✗	✗
Transient state	✓	✗	✗
Identity preservation	✓	✓	✓
Liskov substitutable	✓	✗	✓
Full fletched base-language	✓	✓	✗
Escaping pointers avoided	✓	✗	✓
First class representations	✓	✓	✗

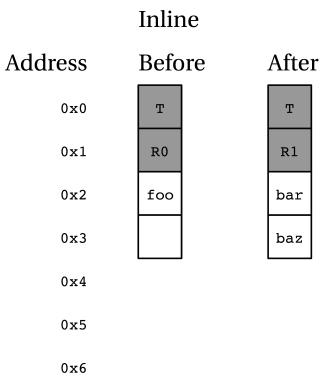
Table 7.1: A comparison of the different implementations of JITds in function of the available features.

point to a another object, i. e., the new representation (see figure 7.1a). This fresh object can reside anywhere in memory. This technique is used in our JITds to Java compiler and is discussed in section 7.3. An alternative implementation omits the explicit handle-object, but requires a full heap traversal upon reclassification to update all references to the old object to point to the new object. This avenue has not been pursued by our research because we expect it to be too costly because of the full heap traversal. Miranda and Béra [58] propose an implementation where the heap traversal is not longer necessary (cf. an efficient `become`). This technique could be investigated in future research when engineering more mature implementations of JITds.

Inline reclassification, on the other hand, does not require a handle but transparently stores an object at a fixed memory location. To support reclassification, however, enough memory has to be allocated to fit the largest possible representation. Figure 7.1b shows how an object of type `T`, or type `R1`, occupies four memory slots, while representation `R0` only requires three (i. e., the empty slot denotes that a slot is not used). Upon reclassification from `R0` to `R1`, the unused, but already allocated, memory location can be overwritten. Thus, when an object is reclassified its address is left unchanged. This technique is used in our JITds to C compiler and is discussed in section 7.4.



(a) Reclassification with handles updates a reference to a newly allocated memory location.



(b) Inline reclassification overwrites existing memory locations.

```
class T combines R0, R1 { ... }
class R0 { int foo;
... }
class R1 { int bar, baz;
... }
class R2 {
... }
```

(c) Definitions of a swappable type `T` and two representations `R0` and `R1`.

```
T o = new T.R0();
// address of o = 0x0
o to R1;
```

(d) Reclassification of an object in representation `R0` to representation `R1`.

Figure 7.1: Reclassification of an object of swappable type `T` from representation `R0` to `R1` and its effect on memory layout.

7.2 The JIT class T: an Abstract Example Program

We explain our two compilers with an illustrative *abstract program*, i. e., a JITds program that does not serve an actual purpose. The program was conceived such that it uses all of the JITds features simultaneously.

The program consists of five new classes: the just-in-time class `T` (see listing 7.3), its three representation classes `R0`, `R1`, and `R2` (see listing 7.1), and the simple class `C` (see listing 7.2), which operates on just-in-time objects.

The three representation classes are classes with one, two, and zero integer fields, respectively. They each implement the method `f`, and both `R0` and `R1` implement one specialised method (cf. section 4.5.1). Hence, these three classes are combinable into one just-in-time class: `T`. First, `T` describes several transitions between the three representations. The transition graph of this example is not complete, i. e., transitions from `R2` to the other two representations are not provided. Second, `T` introduces an invocation counter `#highG` that count all invocations of `g0` where the argument is larger than

Listing 7.1: Three simple classes: `R0`, `R1`, and `R2`.

```
class R0 {
  int foo;
  int f()  { return 0; }
  int g0(int i) { return 0; }
}

class R1 {
  int bar, baz;
  int f()  { return 1; }
  int g1(int i) { return 1; }
}

class R2 {
  int f() { return 2; }
}
```

Listing 7.2: Another simple class.

```
class C {
  int start(T a, T b) {
    int x = a.f();
    int y = b.f();
    return x+y;
  }

  int main(R1 t){
    return t.g1(1);
  }
}
```

`10000`. Third, `T` introduces an internal swap rule from `R0` to `R1` when `foo` happens to be even; and finally `T` introduces an external swap rule that triggers when `C`'s `start` method is invoked. This example is sufficiently complex to explain all of the features implemented in the two compilers. Figure 7.2 shows a UML class diagram of the example just-in-time class `T`.

7.3 JITds-Java

JITds-Java is an implementation of JITds that translates JITds code to plain Java and AspectJ code [48]. The result of the translation is code that can be run on any JVM. In this section we describe the translation from JITds to Java. As explained before, the translation introduces an extra level of indirection to cope with the varying implementations: a just-in-time object that receives a call to a method, forwards it along to the *current representation*. In general, this technique is called *forwarding*. In section 7.3.2 we discuss the relation between this implementation and some of the well-known design patterns

Listing 7.3: The class `T` combines three representations.

```
class T combines R0, R1, R2 {
  R0 to R1 {
    target.bar = source.foo/2;
    target.baz = source.foo/2;
  }

  R1 to R0 {
    target.foo  = source.bar;
    target.foo += source.baz;
  }

  R0 to R2 { }
  R1 to R2 { }

  #highG as g0(int i)  {
    count-if (i>10000);
  }

  swaprule R0 {
    if ( this.foo%2 == 0 && #highG>5 )
      this to R1;
  }

  swaprule int C.start(T a, T b) {
    a to R1;
    proceed;
    b to R0;
  }
}
```

[34], and in section 7.3.3 we describe how forwarding can be implemented in *dynamic languages*.

A JITds program is a set of class definitions. It is possible to partition these definitions into the set of simple class definitions and the set of just-in-time class definitions. In general, simple class definitions are plain Java class definitions and thus do not require translation.[1] The just-in-time class definitions, on the other hand, do require translation. We now describe how to translate a just-in-time class into a plain Java class.

The core idea is to generate a new class with the same name as the just-in-time class, which has a single field `instance` to which all calls can be forwarded, hence "implementation by forwarding". This is shown in listing 7.4. Concretely, `instance` will always be a reference to an instance of one of the representation classes of `T`.

As explained in chapter 4, the data interface of `T` is the union of all of its representations' methods. We reify this union of methods by means of a Java interface. Additionally, this interface introduces two auxiliary methods: one to initiate a representation change

[1] Swap statements that occur in simple classes are the exception, but these statement can be trivially compiled into a call to the `swap` method as introduced later.

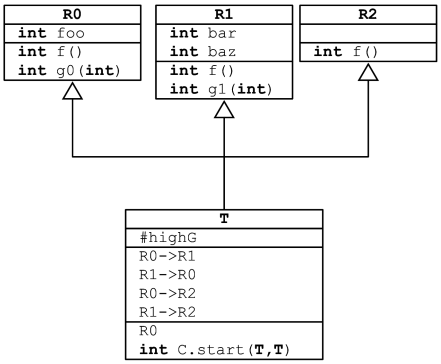


Figure 7.2: UML class diagram of `T` and its three representations.

(i. e., `swap`), and one to retrieve the representation (i. e., `getRepresentation`). The resulting interface `I` is shown in listing 7.5.

Because the original representation classes do not implement this interface, a new version of each representation is created that: First, extends the original representation and, second, implements the newly created interface. The definition of such a class is given in listing 7.6 and is defined as an *inner class* of `T` (this, however, is not shown in the listing). Further, all *transition functions* defined in `T` are added to the appropriate representation classes as *constructors*. For instance, the transition function from `R1` to `R0` is added as a constructor to `R0`. This constructor takes one argument: an object `source` of type `R1`, which corresponds to the keyword from JITds. In the body of the constructor, `target` is replaced by `this`. This straightforward translation results in valid Java code (see listing 7.6, lines 3–6).

The inverse operation, changing from `R0` to another representation, is enabled by the implementation of the `swap` method (lines 16–23). This method creates a new instance by passing `this` (an object with the current representation) to one of the *transition constructors* as introduced above, and assigns it to `T`'s `instance` field. The switch statement only enumerates existing transitions. The default case thus triggers when the intended representation change is not backed up by a transition function/constructor, and throws an `UnsupportedSwapException`.

We can now explain the implementation of `g1` in `___R0`. Although only `R1` implements the `g1` method, it is defined in `___R0` as well. The behaviour of the `g1` method in `___R0` is to change the representation of the just-in-time object from `R0` to `R1`. After the representation change, the `g1` method is invoked again. This time `R1`'s `g1` method can actually be executed.

Finally, the names of the representations are reified as an *enum definition* in the just-in-time class (see line 2 in listing 7.4), i. e., `T.Representation`.

The result of the translation steps just described to go from JITds to plain Java is shown in the class diagram in figure 7.3. These translation steps are sufficient to implement the features described in chapter 4. Invocation counters and swap rules (cf. chapter 6), however, require additional translation steps.

Listing 7.4: The new class `T` has an instance of type `I`. This allows for the encoding of the `combines` relation as a forwarding pattern.

```
1  class T {
2    enum Representation { R0, R1, R2 }
3
4    I instance = null;
5
6    int f()       { return instance.f(); }
7    int g0(int i) { return instance.g0(i); }
8    int g1(int i) { return instance.g1(i); }
9  }
```

Listing 7.5: The interface `I` contains the union of methods found in all representations of `T`. Further, `I` introduces two auxillary methods.

```
1  interface I {
2    int f();
3    int g0(int i);
4    int g1(int i);
5
6    void swap(Representation to);
7    Representation getRepresentation();
8  }
```

Invocation Counters are reified in the representation class (e. g., `__R0`) as an integer field.[2] That field is incremented in the body of the method to which the counter applies, possibly guarded by an if-statement. The invocation counter `highG` from `T` in listing 7.3 is thus translated into the code shown on line 4 of listing 7.10.

Internal Swap Rules An internal swap rule is straightforwardly translated to a new method in a representation class. The internal swap rule shown in listing 7.3, for instance, is added to the class `__R0` as the new method `swaprule`, shown in listing 7.7. The swap statement `this to R1` is translated to a call to `swap`, with the intended representation as argument (listing 7.7, line 3). Directly after the swap, `true` is returned. If the end of the body is reached without performing a representation change, then `false` is returned. In the current implementation of JITds-Java, internal swap rules are checked *before* the execution of every method. To this end, we introduce a call to `swaprule` at the start of every method and either re-invoke the intended method, or forward the method call to the super implementation (one example is shown in listing 7.8). Note that this is a naive implementation strategy because checking whether to swap representation classes on every method invocation can introduce a lot of overhead. In future work we want to investigate how techniques from the field of declarative programming, static analysis, and dynamic analysis can help to reduce the overhead of this approach [18].

[2] To avoid naming conflicts the counter's name is prefixed with two underscores (listing 7.10, line 1).

Listing 7.6: The class __R0 extends the origninal R0 but also implements the new interface I.

```
1  class __R0 extends R0 implements I {
2
3    public __R0(R1 source) {
4      this.foo  = source.bar;
5      this.foo += source.baz;
6    }
7
8    int f()  { return super.f(); }
9    int g0(int i) { return super.g0(i); }
10
11   int g1(int i) {
12     swap( Representation.R1 );
13     return instance.g1(i);
14   }
15
16   void swap(Representation to) {
17     if ( to == Representation.R0 ) return; /* no swap */
18     switch (to) {
19       case R1: T.this.instance = new __R1( this ); break;
20       case R2: T.this.instance = new __R2( this ); break;
21       default: throw new UnsupportedSwapException();
22     }
23   }
24
25   Representation getRepresentation() { return Representation.R0; }
26
27 }
```

Listing 7.7: An internal swap rule is translated into a regular method.

```
1  boolean swaprule() {
2    if ( this.foo%2 == 0  && __highG>10000 ) {
3      this.swap(Representation.R1);
4      return true;
5    }
6    return false;
7  }
```

Listing 7.8: To invoke a translated internal swap rule, the method `swaprule` is invoked before the actual forwarding.

```
1  int f()  {
2    if ( swaprule() ) return T.this.instance.f();
3    else              return super.f();
4  }
```

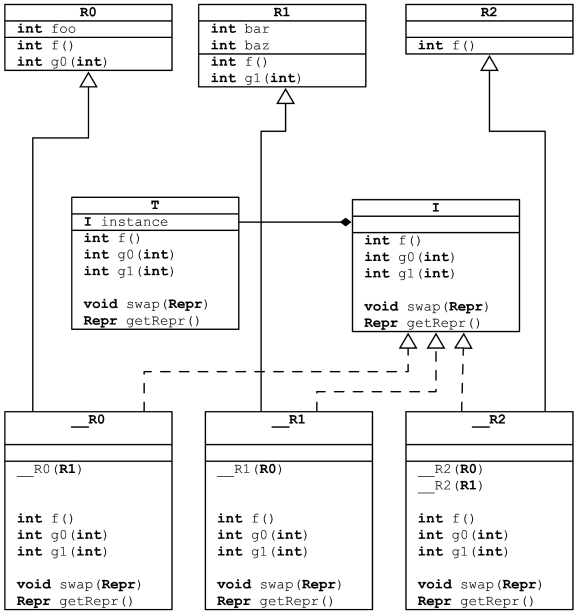


Figure 7.3: UML class diagram of `T` and its three representations after the translation to plain Java.

Listing 7.9: An external swap rule is translated into AspectJ: an *around advice* on a method execution pointcut.

```
1  static aspect ExternalSwaprules {
2    int around(T a, T b): execution( public int C.start(T, T) ) && args(a, b) {
3      a.swap( Representation.R1 );
4      int res = proceed(a, b);
5      b.swap( Representation.R0 );
6      return res;
7    }
8  }
```

External Swap Rules and Interface Swap Rules Translating an external swap rule, such as `C.start` in `T` from listing 7.3, requires extensive code weaving in the otherwise plain Java files. Potentially, this weaving has to be done in classes of existing libraries. Code weaving, however, is the main feature of aspect-oriented programming. Hence, our JITds-Java compiler translates external swap rules into AspectJ: an *around advice* on a *method execution pointcut* (see listing 7.9). Similarly, *interface swap rules* are translated to *around advice* on *method execution pointcut* of AspectJ.

Listing 7.10: An invocation counter is reified as an integer field that gets updated in the body of the correct method.

```
1  int __highG = 0;
2
3  int g0(int i)  {
4    if ( i>10000 )     __highG++;
5
6    if ( swaprule() ) return T.this.instance.g0(i);
7    else              return super.g0(i);
8  }
```

Swap Statements are straightforwardly transformed into a call to the `swap` method. Swap statements can also occur in simple classes that *use* just-in-time objects. Hence, simple class definitions that contain swap statements have to be processed as well. The JITds-Java compiler, however, currently does not perform this pass but it could be added as a trivial compilation step.

7.3.1 Technology used in the Implementation of JITds-Java

Concretely, the above translation is implemented using *Xtext* [28]. Xtext is a framework to prototype new programming languages. In Xtext ones defines a language by defining the grammar and a set of transformations in the form of a recursive AST pretty printer. Then Xtext generates a full infrastructure, including parser, linker, and compiler as well as editing support for Eclipse[3]

Above we discussed how each of the features of JITds are transpiled into plain Java. Here, we discuss one concrete Xtext grammar rule and transformation function, i. e., the grammar rule for an external swap rule and the function `generate` that transforms an *AST-node representing an external swap rule* into Java (and AspectJ). Listing 7.11 shows the syntax rules for an external swap rule in Xtext. In these syntax rules everything between quotes has to match literally, `*` refers to "any number of occurrences" of a term and `?` refers to "an optional occurrence" of a term (cf. regular expressions). In Xtext the terms of a syntax rule can be given an explicit name. For instance, the second rule `ExternalSwapruleBody` matches the occurrence of a left curly bracket, a number of `BlockStatements`, the word "proceed"; a number of `BlockStatements`, and a right curly bracket. When matched, the rule generates an `ExternalSwapruleBody` AST node with the aforementioned elements. The first set of statements and the second set of statements, however, can be accessed by name, using `pre` and `post`, respectively.

Listing 7.12 shows the function that generates Java/ApsectJ code for an `External-Swaprule` AST node. The result of this function is the output generated between the

[3] Alternatively, we could have used *MorphJ* [45], a programming language for meta programming. MorphJ allows the developer to specify a classes by iterating over members of other classes, i. e., as we do with our JITds-Java transpiler written in Xtext. MorphJ, however, does not provide parser, linker, nor IDE support. MorphJ is also *not* discussed as proper related work in chapter 9 because the focus of MorphJ lays on *meta-programming* and not on *just-in-time representation changes*.

Listing 7.11: The syntax rules for an external swap rule in Xtext.

```
1  ExternalSwaprule: 'swaprule' mod?='static'? returnType=JvmTypeReference name=QualifiedName formalParameters=FormalParameters body=ExternalSwapruleBody;
2  ExternalSwapruleBody: '{' pre+=BlockStatement* 'proceed';' post+=BlockStatement*'}';
```

Listing 7.12: The transformation function for an external swap rule in Xtext.

```
1  def generate(ExternalSwaprule sr) {
2    val returnType = pp(sr.returnType)
3    val typesAndNames = pp_paramterlist2paramterlist(sr.formalParameters.params)
4    val types = pp_paramterlist2paramtertypelist(sr.formalParameters.params)
5    val names = pp_paramterlist2arguments(sr.formalParameters.params)
6    val isVoid = sr.returnType.equals("void")
7  '''
8  «pp(sr.returnType)» around(«typesAndNames»): execution( public «IF sr.mod»static«ENDIF» «returnType» «sr.name»(«types») ) && args(«names») {
9    «FOR s : sr.body.pre»
10   «pp(s)»
11   «ENDFOR»
12   «IF !isVoid»«returnType» res= «ENDIF» proceed( «names» );
13   «FOR s : sr.body.post»
14   «pp(s)»
15   «ENDFOR»
16   «IF !isVoid»return res;«ENDIF»
17 }
18 '''
19 }
```

triple quotes (lines 7–18). Triple quotes are Xtext's variant of Lisp's `quasiquote`, and everything between them is interpreted literally. Fragments between « and » are actually evaluated and the result is interpreted literally, much like Lisp's `unquote`. Thus, `'''int i = «(2+3)»;'''`, for instance, creates a variable declaration statement `int i = 5;`. Further, Xtext provides built-in constructs such as `FORALL` (iteration) and `IF` (conditional) to be used within triple-quoted fragments. Lines 9–11, for instance, use the `FORALL` construct to iterate over all the statements in the `pre`-part of the swap rule's body. Each of these statements is pretty-printed (cf. call to `pp`). The function `generate` transforms an external swap rule into an *around advice* on *method execution pointcut* (line 8), a set of statements before the proceed statement (lines 9–11), the proceed statement itself (line 12), a set of statements after the proceed statement (lines 13-15), and a potential return statement (line 16).

All other constructs are transformed using similar transformation functions defined in Xtext. The complete implementation is available on our website (`http://soft.vub.ac.be/~madewael`).

7.3.2 Design Patterns

Multiple *design patterns* have been proposed by Gamma et al. [34] to decouple abstraction from implementation, so that they can vary independently: 1. The strategy pattern is used when behaviour needs to be decoupled; 2. The state pattern is used when state needs to be decoupled; and 3. The bridge pattern is used when both state and behaviour are expected to vary and thus need to be decoupled. A reader familiar with these patterns will recognise our implementation by forwarding as a special case of all three of these patterns.

7.3.3 Forwarding and Delegation in Dynamic Languages

Another technique to implement JITds is to translate representation changes as a *dynamic reparenting* operation, e. g., as is possible in Self [12]. Dynamic reparenting can be seen as a variation of our "implementation by forwarding". The parent is used to forward all method invocations to, that are not readily supported by the child to its "current parent". An implementation in a dynamic language, however, would also mean that the properties of type-safety, as discussed in chapter 5, can no longer be guaranteed. The discussion on which approach is better, statically or dynamically typed languages, belong to the folklore of programming language design and falls outside the scope of this work.

7.3.4 Limitation of Implementation by Forwarding

The current implementation of JITds-Java has two key limitations. First, we cannot support Liskov substitutability between a just-in-time class and its representations (cf. chapter 4). Concretely this means that it is not possible in the current implementation of JITds-Java to write `R0 r = new T.R1();` even though `T` is a subtype of `R0`. Second, as a result of the implementation by forwarding, the JITds-Java implementation suffers

from what we call the *problem of escaping pointers*. This problem occurs when the current representation (directly or indirectly) *leaks* a reference to itself. Then, it is possible to access an instance of a representation that conceptually might no longer exist, e. g., because the just-in-time object has changed its representation and the old representation is outdated. In section 7.4 we present our second implementation JITds-C, which is developed to overcome these limitations.

Example. A well-known example of *the problem of escaping pointers* is the iterator pattern used in many Java collections. Assume we have a just-in-time list (which we discuss more thoroughly in section 8.3.2). When an iterator is requested from the just-in-time list, that request is forwarded to the current representation, e. g., `ArrayList`. Hence, the resulting iterator, `it`, is an `ArrayListIterator`. When, during the use of the iterator, the just-in-time list changes its representation to `LinkedList`, `it` is no longer a legal entry point to the list because it still references the old representation. In section 8.3.2 we discuss how we build such a just-in-time list using JITds-Java and how we cope with this limitation.

7.4 JITds-C

JITds-C is our second implementation of JITds, which is developed with the sole purpose of showing that it is possible to overcome the limitations of the JITds-Java implementation. Thus, we want an implementation of JITds that supports Liskov substitutability and that solves the problem of the escaping pointer. Concretely, we want to be able to assign a just-in-time object of dynamic type `T` to a variable whose static type is a representation of `T` (cf. Liskov substitutability) and `this` should refers to a just-in-time object, as opposed to in JITds-Java where `this` points to the `instance` for which the just-in-time object acts as a proxy (cf. escaping pointer). As a result, we did not put in the engineering effort to support a full-fletched base-language as JITds-Java does. Concretely this means, for instance, that the only primitive type supported by JITds-C is integers (`int`) and arrays; and that simple classes can only define one constructor. This is a non-exhaustive list of limitations, however, we claim that these limitations only decreased the implementation complexity of JITds-C and the ease of programming in JITds, but that JITds-C is still able to support all of the JITds specific features. To this end, JITds-C is a straightforward translation of the formal semantics presented in chapter 5.

The base implementation strategy used in JITds-C to implement dynamic object reclassification is *inlining* (cf. figure 7.1b). This means that the new representation is stored at the same memory location as the old representation. We first discuss how objects are stored in general. An object combines both state and behaviour. The state of the object is unique per object. Because JITds is a *class-based* programming language, behaviour is shared for a class of objects. In JITds-C, an object can thus be represented by a collection of its state and a reference to its class's behaviour. The class's behaviour is then defined by implementations of all methods, which are referenced to through *method tables* (see section 7.4.1).

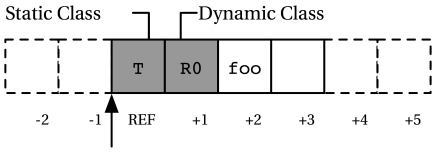


Figure 7.4: A just-in-time object with static class `T` and dynamic class `R0` as it is represented in our custom runtime engine.

Listing 7.13: The object head is a pre-allocated chunk of memory.

```
1  typedef void** REF;
2  REF MEM = malloc( MEMORY_SIZE*sizeof(REF) );
3
4  REF  MEMREF(REF base, int idx)          { return base[idx]; }
5  void MEMSET(REF base, int idx, REF val) { base[idx]  = val; }
```

Listing 7.14: The method `R0.f` is compiled into the C-fucntion `R0_f`.

```
1  typedef REF (*method_t)(REF,REF*);
2
3  REF R0_f(REF this, REF* args){
4    return i2r(0); // comes from return 0;
5  }
```

In chapter 5, we explain that in JITds the behaviour is defined by *two classes*: the *static class* and the *dynamic class*. Figure 7.4 shows an object with the just-in-time class `T` as the *static class* and the simple class `R0` as the *dynamic class*. `R1` is the representation that needs the most space (i. e., it has three fields). Furthermore, a just-in-time object needs enough space to store all the fields, here three, because the largest representation requires space to store three fields. The object heap is a pre-allocated chunk of plain memory (listing 7.13), where all objects are stored.

JITds methods are compiled into C-functions with two arguments: the `REF` to the receiver object, and an array `REF*` which leads to the arguments supplied to the method invocation. These functions have the type `method_t`. This is exemplified by listing 7.14.

In a class-based object-oriented programming language the behaviour of an object is defined by its class. As is common in implementations of these languages, the correct method is looked up at invocation time in *method tables*. This is also the case in JITds-C, however, because of representation changes things become trickier (chapter 5). We now go into the details of method tables, starting with the basic implementations used in many traditional language implementations, including JITds. We then build on this to explain the more complex cases in JITds.

Listing 7.15: Simple method table and `invokeD`.

```
1  method_t** simple_method_table = init_smt();
2
3  REF invokeD(int method_id, REF obj, REF* args){
4    int class_id = obj_dynamic_class(obj);
5    method_t method = simple_method_table[class_id][method_id];
6    return method(obj, args);
7  }
```

7.4.1 Method Tables

Just like in any other class-based object-oriented programming language, object behaviour is shared among a class of objects in JITds (cf. per group [52]). Therefore, behaviour is not stored per object but per class. In JITds-C, the methods that define the behaviour of a class of objects are stored in *method tables*. JITds-C relies on three different kinds of method tables: *simple method tables*, *just-in-time method tables*, and *just-in-time method conversion tables*.

Simple Method Tables. A simple class has a set of methods, some of them are inherited from ancestors, others are introduced by this class, and others override methods from ancestors. It is possible to assign to each method a unique index number (from 0 to x), such that methods with the same name as that of an ancestor class have the same number as that ancestor's method. The *simple method table*, shown in listing 7.15, maps from a (simple) class and a method identifier to a method reified as a C-function pointer (`method_t`) (line 5). The simplest invocation construct, `invokeD`, looks up the function pointer (lines 4 and 5) and calls it with the receiver and the provided arguments (line 6). The *simple method table* is a method table as it is traditionally found in implementations of class-based object-oriented programming languages.

Just-in-Time Method Tables. A simple method table, however, is not sufficient for a just-in-time class. Because the simple method tables of the various representations are created independently from each other, it is possible that the *same* method has a different numeric identifier in the method tables of the various representations (e. g., `f` in `R0` and `R1`); or *different* methods can have the same identifier in the various representations (e. g., `g0` in `R0` and `g1` `R1`). The solution is to have a unique numbering for all the methods of the just-in-time class, regardless of their numbering in the simple classes; and to build a second kind of method table that converts from the just-in-time class numbers to the numbers used by the original representation classes. The *just-in-time method table*, shown in listing 7.16, maps from a just-in-time class, a representation class, and a method identifier to a method reified as a C-function pointer (`method_t`). The invocation construct `invokeI` is used when it is statically known that the receiver is a just-in-time object. `invokeI` looks up the requested method and, when found, calls that function with the receiver and the provided arguments (line 10). The *just-in-time method table* is a method table as it is traditionally found in implementations of object-

Listing 7.16: JIT method table and `invokeI`.

```
1  method_t*** jit_method_table = init_jmt();
2
3  REF invokeI(int method_id, REF obj, REF* args){
4    int jit_id = obj_static_class(obj);
5    int rep_id = obj_dynamic_class(obj);
6    method_t method = jit_method_table[jit_id][rep_id][method_id];
7    if ( method==NULL ) {
8      return invokeT(method_id, obj, args);
9    } else {
10     return method(obj, args);
11   }
12 }
```

oriented programming languages with interfaces such as Java. For a discussion on how to efficiently implement such constructs we refer to [4].

Up until now, JITds-C proves to be a (naive) implementation that uses the classic implementation techniques of other object-oriented programming languages with *interfaces* (e. g., Java), i. e., our simple method tables are known as *static method tables* in Java and our just-in-time method tables are known as *virtual method tables* in Java. In JITds, however, it is possible that the requested method does not exist for the current representation. In this case, `invokeT` is called (line 8) which takes care of representation changes. We return to `invokeT` and representation changes later.

Simple invocation: revisited In the general case, `invokeD` is sufficient to look up and execute a method of a simple class. Because a just-in-time class is a subtype of all of its representations (e. g., `T <: R0`), and because of Liskov's substitution principle, it is possible that a just-in-time object is used where the expected static type is a simple class of one of the representations. For instance, this is the case in `C.main` in listing 7.2. In this case, `invokeD` only works if the current representation is a subtype of the statically expected type. To guard these cases, regular method invocations are not compiled into an `invokeD`, but into a different method, called `invokeS`, which also takes the expected static type as a parameter and performs some tests.

`invokeS` , shown in listing 7.17, is similar to `invokeD` but has an extra parameter, `expected_id`, which holds the expected dynamic class of the receiver. `invokeS` calls `invokeD` in either of two cases: if the receiver is a simple object (cf. the static class is the same as the dynamic class), or if the current representation of the receiver is a subtype of the expected class. When neither condition is met, `invokeI` is called to handle the method invocation. To use `invokeI`, however, `method_id` first has to be translated from the simple class numbering to the just-in-time class numbering of the method. These translations are reified in the final method table: `jit_convert_table`. The *just-in-time method conversion table*, also shown in listing 7.17, is a mapping from a just-in-time class, a representation class, and a method identifier to a new method identifier.

Listing 7.17: `invokeS` guards `invokeD` from calling the wrong method on a just-in-time obejct.

```
1  int*** jit_convert_table = init_jct();
2
3  REF invokeS(int method_id, int expected_id, REF obj, REF* args){
4    int jit_id = obj_static_class(obj);
5    int rep_id = obj_dynamic_class(obj);
6    if ( jit_id == rep_id ) {
7      return invokeD(method_id,obj, args);
8    } else if ( is_subtype(rep_id,expected_id) ) {
9      return invokeD(method_id,obj, args);
10   } else {
11     method_id = jit_convert_table[jit_id][rep_id][method_id];
12     return invokeI(method_id, obj, args);
13   }
14 }
```

Listing 7.18: transition functions

```
1  typedef void(*tf_t)(REF);
2  tf_t***     tf_table;
3
4  REF invokeT(int method_id, REF obj, REF* args){
5    int jit_id = obj_static_class(obj);
6    int rep_id = obj_dynamic_class(obj);
7    for(int to=0 ; to<number_of_classes_and_jit_class ; to++){
8      method_t method = jit_method_table[jit_id][to][method_id];
9      tf_t     tf     = tf_table[jit_id][rep_id][to];
10     if ( (method!=NULL) && (tf!=NULL) ) {
11       tf( obj );
12       return method(obj, args);
13     }
14   }
15   exit(-1);
16 }
```

Change the representation and re-invoke. When `invokeI` fails to find a method for the current representation, it calls `invokeT`. `invokeT` (see listing 7.18) iterates over the just-in-time method table and tries to find a representation for which the method does exist. If there exists a *transition function* from the current representation to the representation with the requested method, then the transition function and method are executed. Transition functions are also looked up in a table, the *transition function table* `tf_table`. A transition function table holds a mapping from a just-in-time class, a source representation, and a target representation to a transition function reified as a three-dimensional C-function pointer (`tf_t`).

Listing 7.19: The functions provided by *the runtime*.

```
1  REF alloc_instance(int field_count);
2
3  REF obj_read(REF obj, int field);
4  void obj_write(REF obj, int field, REF v);
5
6  int  obj_static_class(REF obj);
7  int  obj_dynamic_class(REF obj);
8
9  REF invokeD(int method_id, REF obj, REF* args);
10 REF invokeS(int method_id, int expected_id, REF obj, REF* args);
11 REF invokeT(int method_id, REF obj, REF* args);
12 REF invokeI(int method_id, REF obj, REF* args);
```

7.4.2 Technology used in the Implementation of JITds-C

As has become clear in the previous section, a significant part of the compiler's job is to generate the method tables and other tables. Besides the existence of three different kinds of method tables and the corresponding invocation constructs, the implementation of the JITds-C compiler is a fairly straightforward translation of the formal semantics described in chapter 5. In what follows, we briefly sketch its implementation.

A JITds program is compiled into a single C file that imports *the JITds runtime*. The JITds runtime is a custom-built set of functions that form the execution engine of the online execution of the program: creating objects, reading and writing fields, and invoking methods. An important part of the interface of the runtime is given in listing 7.19. Besides generating the various method tables, a second job of the compiler is to generate functions that call these runtime functions, e. g., the JITds statement `this.x = 5` has to be translated into something like `obj_write(this, f__T_x, (REF)5)` (with `f__T_x` the index of the field `x` in type `T`).

The compiler generates its code in four phases. The first phase of the compilation process starts with a single JITds file that is *parsed* and converted to *AST nodes*. That first file contains exactly one class definition, which is subsequently parsed into a *class definition node*. During this AST building process, each created *class definition node* is added to the *class table*. When an unknown class is referenced (i. e., it is not yet present in the class table), the file containing the definition of the new class is also scheduled for parsing and *AST building*. The first phase continues until no more classes are scheduled, i. e., when all classes used in the program have been parsed and converted into AST-nodes. The result is a class table, `CT`, that contains mappings between all class names and the corresponding class definitions as AST nodes.

In the second phase, the *compose phase*, the AST nodes from the first phase are updated such that all contextual information is directly accessible from each AST node. For instance, after the first phase, a class definition node knows its parent only by name. During the *compose phase*, this name is replaced by the corresponding class definition AST node as it can be found int the class table `CT`. Similarly, for field definitions, method definitions, ... the names of a type are replaced by an actual AST node from `CT`.

Listing 7.20: The method `R1.f` translated into C.

```
1  REF R1_f(REF this, REF* args){
2    return invokeS(m__C_sum,c__C,
3      obj_read(this,f__R1_c),
4      (REF[]){obj_read(this,f__R1_bar), obj_read(this,f__R1_baz)});
5  }
```

The third phase is the *type-check phase*, in which the AST nodes found in the class table are checked against the well-formedness rules from section 5.4.4. The program is either rejected, because there exist static type errors, or the program is accepted, in which case the AST nodes are typed (when applicable) . This means, for instance, that a method invocation node is tagged with of the receiver's type.

The fourth phase computes all indices are constructs all tables, i. e., each class is assigned a unique index number, all methods within a class are assigned a unique number, and methods are given a unique name, etc. Then our implementation pretty-prints the AST nodes of methods and transformation functions as C code. `R1.f`, for instance, is translated into the code shown in listing 7.20.

After these four phases, the resulting C file can be compiled into a binary file (i. e., using a standard C compiler) and run. To build the JITds-C compiler, we used ANTLR as the parser generator and Racket to convert AST nodes into C code.

7.4.3 Conclusion

JITds-C is a naive and straightforward implementation of a conventional statically-typed, class-based object-oriented programming language, based on two method tables, augmented with a *transition function table* and a *just-in-time method conversion table*. The JITds-C compiler is developed with the sole purpose of showing that is feasible to implement JITds, without the limitations of JITds-Java. Concretely, in JITds-C a just-in-time object can be used when statically one of its representation is expected (cf. Liskov substitutability) and JITds-C does not suffer from the problem of escaping pointers. We can guarantee this because JITds-C is a straightforward translation of the operational semantics presented in chapter 5, for which we proved that is works.

7.5 Summary

Introducing JITds-Java and JITds-C shows that it is possible to implement JITds. JITds-Java is a compiler that translates JITds code into plain Java and AspectJ code. The implementation technique that is used to implement the representation changes is *forwarding*. The JITds-Java compiler, however, suffers from two limitations: Liskov substitutability is not guaranteed and we have the problem of escaping pointers. We show that these issues can be resolved by implementing JITds-C, which compiles JITds code into C code. The limitations of JITds-Java are not present because JITds-C uses the inlining implementation technique. The custom-built runtime engine relies on method

tables to look-up the behaviour of object. Variations of the simple method tables and the just-in-time method tables can be found in implementations of traditional class-based programming languages as well. The JITds-C compiler has a *just-in-time method conversion table* that enables the dynamic look-up of a method in a just-in-time object when statically a simple object was expected. This method table is unique to the JITds-C implementation. A limitation of the JITds-C compiler is that it supports only a subset of JITds's base language, for instance, only integers are supported as a native type. The efficiency and usability of JITds, however, would benefit from a more mature implementation.

Chapter 8

Evaluation

The goal of this chapter is to evaluate the *language design* of the programming language JITds, against the requirements set forth in chapter 1: The language must allow to change the data representation of data objects at runtime (**R1**); the language must provide guarantees w.r.t. the type safety in the context of representation changes (**R2**); and the language must allow programmers to developers to disentangle general application logic from the crosscutting concern of representation changes. (**R3**).

Chapters 4, 5 and 7 sufficiently show that **R1** and **R2** are met. Swap rules, as described in chapter 6, are introduced to meet **R3**, as they allow programmers to disentangle application logic from representation change logic. While chapter 6 introduces these features conceptually, we did not yet show whether these features actually work towards **R3**. Thus, the focus of this chapter is to show that in a JITds program, application logic and representation logic are cleanly separated, and that application logic and representation logic only need to be intertwined if changing the representation is an actual part of the application logic, e. g., opening a file. At the same time we evaluate the performance gains of changing the representation of a data structure at runtime.

8.1 A set of synthetic benchmarks

We evaluate JITds using a set of five synthetic benchmark programs. The programs were designed to illustrate the different scenarios of applicability (see chapter 3) and to use the main features of JITds (see chapters 4 and 6). These example programs are discussed in section 8.3, where they are compared to plain Java programs in terms of efficiency and code structure.

8.1.1 Caveat

Showing a significant performance increase in real-world applications when using JITds would count as a strong evaluation. However, we choose to evaluate JITds in the context of a set of synthetic benchmarks. Here, we explain the pragmatic choice of evaluating

JITds in the context of a synthetic benchmark suite instead of in the context of real-world applications.

JITds is a programming language that is designed to enable *programmers* and *performance experts* to develop programs that rely on representation changes to improve the overall performance. A large part of the performance gain has to be attributed to *well-developed representation change incentive code*, i. e., the job of the performance expert. To evaluate JITds, however, we need to verify whether JITds is sufficiently expressive to develop programs in a variety of scenarios (see chapter 3) and whether application logic and representation change incentive code can be disentangled. Much of the related work (see chapter 9) focusses on designing representation change incentive code itself, and for building the solid models that lay at the base of good representation change incentive code, we refer to our related work chapter (i. e., chapter 9). Deciding when and why to change the representation (i. e., representation change incentive code), however, is not the focus of this chapter.

We choose to evaluate JITds using a set of five synthetic benchmark programs because it enables us to focus on *how to express* representation change incentive code in JITds, as opposed to focussing on the development of the actual representation change incentive code. More structured approaches to develop representation change incentive code are presented in [47, 61, 69]. As we will show in section 8.4, all main features of JITds were used in these five programs. Moreover, these same five programs cover all five scenarios, as sketched in chapter 3.

8.2 Experimental Setup

First, the code presented in this chapter is tweaked for *illustrative* purposes. To simplify the discussion or to convey a message more clearly, the code may have been simplified or adapted. For example, when comparing two similar code fragments, variables of one of the fragments may have been renamed to match those of the other fragment; edge cases which are handled by condition blocks may have been omitted; methods have been manually inlined to show multiple computations in one block. These adaptations should not have a significant impact on the presented results. Nevertheless, for reproducibility purposes, the actual code that was used is available on our website, as well as the source code of the JITds-java compiler used to generate the just-in-time classes presented in this chapter (`http://soft.vub.ac.be/~madewael`).

Second, all the just-in-time classes presented in this chapter are compiled with the JITds-Java compiler. As discussed in chapter 7, this compiler implementation suffers from the problem of *escaping pointers*. In the the *List Program* (section 8.3.2), we took this limitation into account while *designing* the application (see discussion section 8.3.2.2). In the *String Program* (section 8.3.5), this was not possible, hence we needed to *manually* add one line of code to the code generated by our JITds-Java compiler (see discussion section 8.3.5.2).

Third, since we use the JITds-Java compiler for this chapter, which compiles JITds code into plain Java, we need to take the peculiarities of performance benchmarking on the JVM into account [35, 9]. It is important to be aware of JVM warm-up time, the dif-

ference between interpreted mode and mixed mode, dead code elimination and other aggressive compiler optimisations, garbage collection effects, etc. Hence, to take these peculiarities into account, we repeat all benchmark programs 30 times and discard the first three runs, i. e., common practice to approximate a steady state execution. Concretely, we report on the arithmetic mean and standard deviation of the remaining 27 runs. The data files from which our graphs are generated are available on our website as well.

Finally, the benchmarks are executed on a HP Proliant HPDL585G7 with four AMD Opteron 6376 processors, forming an eight-node NUMA setup supporting 64 hardware threads, running the Ubuntu 15.04 (GNU/Linux 3.19.0-33-generic x86_64) operating system. The device has 128 GB of working memory (RAM) with 16MB of L3 cache per processor and a 1 TB hard disk. We used Java version 1.8 (update 66) on the Java HotSpot 64-Bit Server VM (build 25.66-b17, mixed mode). We ran the java command with the standard settings for this VM, i. e., standard garbage collector and standard heap sizes.

8.3 Example Programs

In this section we evaluate JITds using five different example programs. The *Matrix Program* (section 8.3.1), the *List Program* (section 8.3.2), the *File Program* (section 8.3.3), the *Sorting Program* (section 8.3.4), and the *String Program* (section 8.3.5). Each of these micro-benchmark programs are created based on the five scenarios sketched in section 3.2.7.2: Changing Data Access Pattern, Changing Interface Usage Pattern, Changing Functionality, Freezing and Thawing Objects, and Data Specialisation, respectively. Furthermore, in these five examples we are able to demonstrate practical use of the following language features of JITds: *combining representations*, *transition functions* and *named transition functions*, *swap statements*, *specialised swap*, *external swap rules*, *internal swap rules*, *interface swap rules*, *invocation counters*, and *first class representations*[1].

The following five sections present different example programs. In each section we following the same pattern: 1. We sketch the **context**; 2. We sketch the **implementation** of the representations and the just-in-time class; 3. We present the **experimental setup**; 4. We discuss the measured **results**[2]; 5. We discuss the **difference in code** between using JITds and not using JITds .

When comparing the code of a JITds program with a regular (Java) program, we colour the fragments that serve a different purpose with different colours, i. e., all choices of which representation to use is coloured ■, all transformation logic is coloured ■, all representation change incentive code is coloured ■, and all application logic is coloured ■(see figure 8.1).

[1] These are *all* the features presented in chapters 4 and 6, except for *transient state* and *by-pass methods*.

[2] This section is omitted in the file example (section 8.3.3, because the file program does not aim to improve performance.)

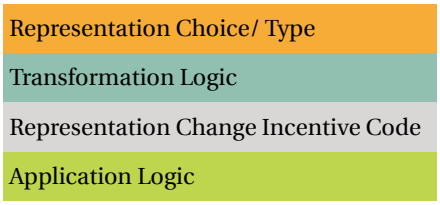


Figure 8.1: The colour scheme to differentiate between the purposes of code fragments.

8.3.1 The Matrix Program

We call the first example program "The Matrix Program". The Matrix Program raises a 512 by 512 matrix to the 16th power. This program relies on an external swap rule to react to change in data access pattern with a representation change.

8.3.1.1 Context: Changing Data Access Pattern

For the same amount of work, different data access patterns can result in different execution times. In the presence of caches, for instance, a program that exhibits good data locality is expected to be more efficient than a program that exhibits bad data locality. Changing the layout of the data can, for the *same computation*, increase or reduce data locality. As already shown in section 2.2, the data access pattern for the matrix multiplication `mul(mA, mB)` prefers `mA` to be stored in row-major order and `mB` to be stored in col-major order for better performance. This first example program is based on these insights.

8.3.1.2 Implementation

In this first example program, we evaluate a just-in-time class `Matrix` that changes its representation between `RowMajorMatrix` and `ColMajorMatrix`. We refrain from presenting the details of the implementation of both representations here, as they are discussed extensively in earlier chapters (see section 2.2). We do present the auxiliary class `Utils` (see listing 8.1), which implements a multiplication method (`mul`, lines 3–15) and a power method (`pow`, lines 17–23). Other static auxiliary methods (e. g., `makeIdentityMatrix`, used on line 18) are omitted from this code snippet.

For this experiment we define a just-in-time class `Matrix` (see listing 8.2) that combines `RowMajorMatrix` and `ColMajorMatrix`, together with a definition for the two possible *transition functions* (lines 4–11 and lines 14–22, respectively). Furthermore, listing 8.2 shows two external swap rules for the `mul` method with the subscripts `1` and `2`, respectively. The subscripts are introduced to differentiate between the two alternative swap rules: in the actual experiment *only one of the swap rules is used.*

From section 2.2, we know that the matrix multiplication (as defined in listing 8.1) is best executed with the matrix `mA` in `RowMajorMatrix` representation and the matrix `mB` in `ColMajorMatrix` representation. The *external swap rule* on lines 24–28 (i. e., the one with subscript `1`) hardcodes these insights. The *external swap rule* on lines 33–

Listing 8.1: Raising a matrix `mA` to the n^{th} power.

```
1  class Utils {
2
3    public static Matrix mul(Matrix mA, Matrix mB) {
4      Matrix mC = new Matrix(mA.getRows(), mB.getCols());
5      for(int r = 0 ; r<mC.getRows() ;  r++ ){
6        for(int c = 0 ; c<mC.getCols(); c++ ){
7          double d = 0;
8          for(int k = 0 ; k<mA.getCols(); k++ ){
9            d += (mA.get(r,k) * mB.get(k,c));
10         }
11         mC.set(r,c,d);
12       }
13     }
14     return mC;
15   }
16
17   public static Matrix pow(Matrix mA, int n) {
18     Matrix mC = makeIdentityMatrix( mA.getCols() );
19     for (int i=0 ; i<n ; i++) {
20       mC = mul(mA, mC);
21     }
22     return mC;
23   }
24
25   ...
26
27 }
```

41 (i. e., the one with subscript `2`) uses machine learning to learn the best representation for both matrix `mA` and `mB`: measuring the execution time of each multiplication and learn which combination of representations can be multiplied the fastest.

To learn the best combination of matrix representations we use *the Decreasing Epsilon Greedy Strategy* as it was introduced on page 136. We use the measured the execution time — i. e., of performing a multiplication (lines 36–38) — as *cost function* and use `l`, an instance of `DecreasingEpsilonGreedyStrategy`, to learn how to minimise that cost. This boils down to learning which of four combinations of matrix representations takes the shortest time. The four combinations of matrix representations can be encoded by the integers 0,1,2, and 3; and we can use modulo (line 34) and integer division (line 35) to decode such an integer. This integer is called `action` in listing 8.2. `l` is an instance of `DecreasingEpsilonGreedyStrategy` that is able to learn the best our of four choices (see line 30 and 31). On line 34, the current best choice is retrieved from `l` and stored into `action` (`action` $\in \{0,1,2,3\}$). Furthermore, `action` is used on lines 34 and 35 to change the representations of `mA` and `mB`, respectively. Then, on line 39, we show how `l` is updated with the information about the cost of multiplying the two matrices (cf. `(end-begin)` with the respective representations (cf. `action`).

Listing 8.2: The just-in-time class `Matrix`.

```
1  class Matrix
2  combines RowMajorMatrix, ColMajorMatrix  {
3
4    RowMajorMatrix to ColMajorMatrix {
5      target( source.getRows(), source.getCols() , source.data);
6      target.data = new double[source.getRows()* source.getCols()];
7        for(int r = 0 ; r<source.getRows() ;  r++ ) {
8          for(int c = 0 ; c<source.getCols() ; c++ ) {
9            target.set(r,c,source.get(r,c));
10         }
11       }
12   }
13
14   ColMajorMatrix to RowMajorMatrix {
15     target( source.getRows(), source.getCols() , source.data);
16     target.data = new double[source.getRows()* source.getCols()];
17       for(int r = 0 ; r<source.getRows() ;  r++ ) {
18         for(int c = 0 ; c<source.getCols() ; c++ ) {
19           target.set(r,c,source.get(r,c));
20         }
21       }
22   }
23
24   swaprule static Matrix Utils.mul1(Matrix a, Matrix b) {
25     a to RowMajor;
26     b to ColMajor;
27     proceed;
28   }
29
30   static DecreasingEpsilonGreedyStrategy l
31               = new DecreasingEpsilonGreedyStrategy(4);
32
33   swaprule static Matrix Utils.mul2(Matrix a, Matrix b) {
34     int action = l.next();
35     a to (Matrix.Representation.get( action%2 ));
36     b to (Matrix.Representation.get( action/2 ));
37     long begin = System.currentTimeMillis();
38     proceed;
39     long end = System.currentTimeMillis();
40     l.update( action, (end-begin));
41   }
42
43 }
```

8.3.1.3 Experiment

The matrix micro-benchmark program implements a power function `pow(Matrix m, int n)`, which raises the matrix `m` to the n^{th} power (see listing 8.1). To do so, `pow` iteratively calls `mul`. We measure the execution time of raising a 512×512 matrix to the 16^{th} power. The size of the matrix is chosen such that it does not fit in the cache, and the exponent has been chosen to obtain a longer running computation.

In our experiment we compare five variations of calling `pow`:

JIT Matrix (no swap rules) An implementation that uses the just-in-time class `Matrix` and the method `mul` as described above, *without* any of the discussed external swap rules.

JIT Matrix (fixed swap rule) An implementation that uses the just-in-time class `Matrix` and the method `mul` as described above, with the external swap rule that always uses the representations `RowMajorMatrix` and `ColMajorMatrix` for its first and second argument, respectively (cf. $\texttt{mul}_1$ in listing 8.2).

JIT Matrix (learning swap rule) An implementation that uses the just-in-time class `Matrix` and the method `mul` as described above, with the external swap rule that uses the decreasing epsilon greedy strategy (cf. $\texttt{mul}_2$ in listing 8.2).

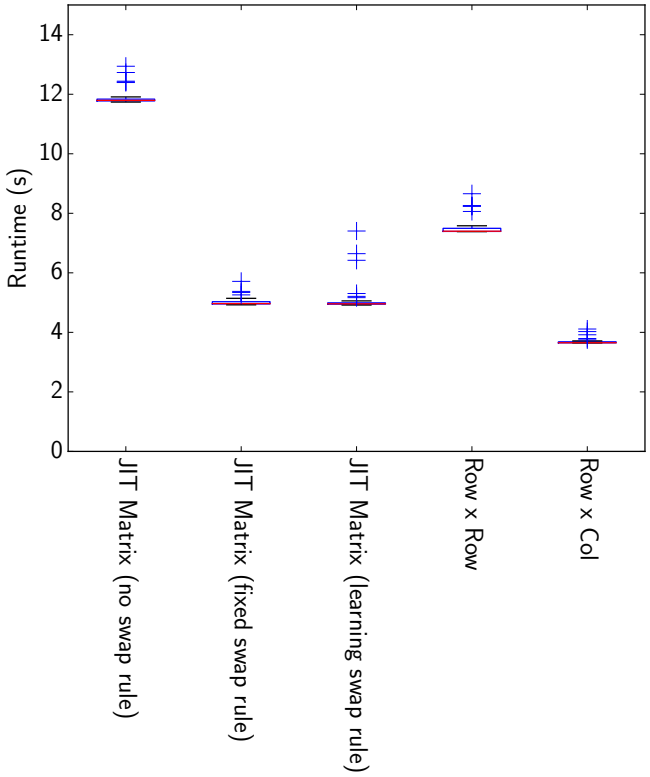
Row × Row A plain Java implementation, that uses a variation of the method `mul` from listing 8.1, adapted to statically accept two matrices of type `RowMajorMatrix`. This version is effectively the same as "JIT Matrix (no swap rules)" but without the overhead introduced by JITds-Java, i. e., a base line to compare against.

Row × Col A plain Java implementation, that uses a variation of the method `mul` from listing 8.1, adapted to statically accept a matrix of type `RowMajorMatrix` and a matrix of type `ColMajorMatrix` for its first and second argument, respectively. In this variation, the implementation of `pow` is adapted such that the type requirements are met, i. e., by means of manually introduced *local representation changes* (see rightmost version in figure 8.7a op page 192). This version is effectively the same as "JIT Matrix (fixed swap rule)" but without the overhead introduced by JITds-Java.

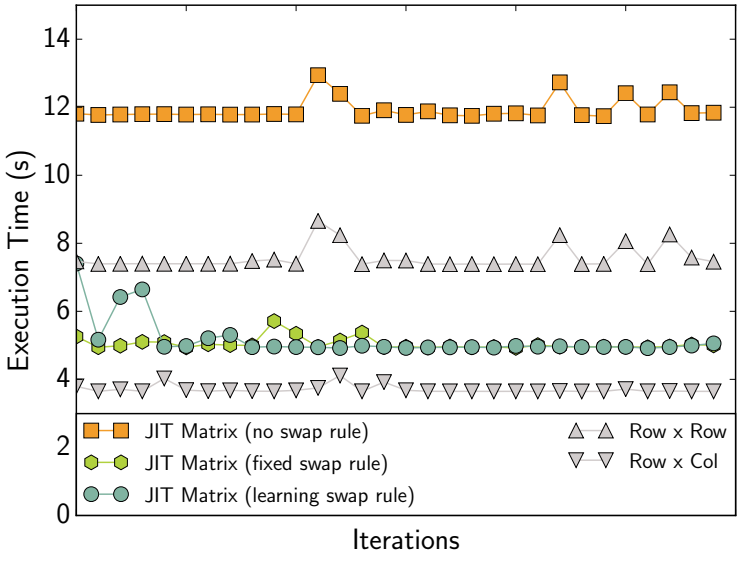
8.3.1.4 Results

The results of the multiplication experiment, repeated 30 times, are shown in figure 8.2. We first consider the summary of these results in the form of the box and whiskers plot in figure 8.2a (i. e., the lower to upper quartile of the measurements, the median, the range of the data, and the outliers).

A first observation is that for all our measurements, the boxes are tight and the whiskers are close to the boxes. This means that our experiments are relatively stable. This property seems to hold for all our micro-benchmarks; hence, for the remainder of this chapter we will no longer use box and whiskers plots, but rather report on the mean of the measured data points and plot the standard deviation as error bars. The other observations concern the matrix example itself.



(a) The execution times of 30 runs represented as a box and whiskers plot.



(b) The evolution of execution times over 30 consecutive runs.

Figure 8.2: Raising a 512x512 matrix to the 16th power.

First, the assumption that the data access pattern is best when the left matrix is in the `RowMajorMatrix` representation and the right matrix is in the `ColMajorMatrix` representation, and that paying the cost of a representation change is beneficial for performance, is confirmed again: "JIT Matrix (fixed swap rule)" is faster than "JIT Matrix (no swap rules)"; and "Row × Col" is faster than "Row × Row".

Second, both "JIT Matrix (no swap rules)" and "Row × Row" perform the exact same computation and have the exact same data access pattern. Hence, the difference in the measured execution time is the result of the extra layer of indirection introduced by the JITds-Java compiler (cf. implementation by forwarding, chapter 7). Similarly, we can explain the difference in execution time between "JIT Matrix (fixed swap rule)" and "Row × Col" by attributing the delta to the extra level of indirection.

Third,the difference in execution time between "JIT Matrix (fixed swap rule)" and "JIT Matrix (learning swap rule)" is almost negligible. The biggest difference between these two variants is that the "JIT Matrix (learning swap rule)" variant has more and larger outliers. Figure 8.2b plots the same data as figure 8.2a but reveals the variation in execution time over the different runs, i. e., the data points on the left represent the first run, the data points on the right represent the last run. We expect that all lines are more or less horizontal. Small variations can be explained by fluctuations in the virtual machine (e. g., garbage collection, JIT compilation). However, the fluctuations as seen in data gathered from "JIT Matrix (learned swap rule)", however, can not be explained as such. "JIT Matrix (learning swap rule)" sees higher execution times in the first few iterations because it takes time to learn the most efficient data representations. After a few repetitions, the execution time of "JIT Matrix (learning swap rule)" is comparable to that of "JIT Matrix (fixed swap rule)", as discussed above.

8.3.1.5 Code Comparison

Figure 8.7a shows three versions of a program that implement the same functionality: a type `Matrix`, for which there exist two representations `RowMajorMatrix` and `ColMajorMatrix`, and for which the multiplication and the power methods are defined. Moreover, the programs are written such that the multiplication always uses the best combination of representations, i. e., `RowMajorMatrix` times `ColMajorMatrix`.

The leftmost version uses JITds and implements "JIT Matrix (fixed swap rule)". The middle version is a plain Java version where the method `mul` is responsible for checking and (potentially) changing the representation of its arguments (local representation change). The rightmost version is again a plain Java version, where `mul` is statically defined to operate on the combination `RowMajorMatrix` times `ColMajorMatrix` only , and leaves the responsibility for changing the representation to the client code, which is `pow` (cf. the "Row × Col" variant) here.

Figure 8.7a reveals that using JITds allows the developer to define the application logic (■), while being oblivious of the data representation (■). Furthermore, when using JITds, all transformation logic (■) and all representation change incentive code (■) can be grouped together.

Conversely, in the version in the middle, the representation change incentive code (■) is entangled with the application logic and the transformation logic (■) is decoupled

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

Table 8.1: Intuitive performance characteristics of `List`-representations in Java.

from the representations. This is also true in the rightmost version, where information about the data representation (■) is scattered across the application code (■).

8.3.1.6 Conclusion

The conclusion of this experiment is that external swap rules can be used to improve the performance of a program by reacting to a change in the data access pattern of the computation. Moreover, the use of external swap rules allows developers to disentangle application logic (multiplication) from representation change incentive code (data access patterns) and transformation logic.

8.3.2 The List Program

We call the second example program "The List Program". The List Program populates a list by inserting elements close to the front and then estimates the average element based on a random sample. This program uses invocation counters and internal swap rules to react to changes in the *interface usage pattern* with a representation change.

8.3.2.1 Context: Changing Interface Usage Pattern

Consider a list abstraction and its two traditional implementations: the *pointer-based* list implementation (e. g., `LinkedList` in Java) and the *array-based* list implementation (e. g., `ArrayList` in Java). Table 8.1 summarises the intuitive performance characterisation of both. From this table we can conclude that for an algorithm where the number of random accesses dominates the computation, `ArrayList` is the optimal choice and in an algorithm where the number of updates dominates the computation, `LinkedList` is the optimal choice. When the (data interface) usage pattern of the data structure changes during the execution of a program,e. g., from an access-dominated computation to an update-dominated computation, then a representation change could improve the performance of that program. This second example program is based on this assumption.

8.3.2.2 Implementation

In this second example program we evaluate a just-in-time class `JL` (cf. Jit List) that toggles between a representation based on `ArrayList` and a representation based on `LinkedList`.

Listing 8.3: A reduced data interface for lists.

```
1  public interface IL {
2
3    public int size();
4
5    public Event get(int index);
6    public void insert(Event e);
7
8    public void initIterator();
9    public boolean hasNext();
10   public Event next();
11
12 }
```

Listing 8.4: The class `Event` represents a mouse event: a timestamp and coordinates.

```
public class Event {

  public final long t;
  public final int x, y;

  public Event(long t, int x , int y){
    this.t = t;
    this.x = x;
    this.y = y;
  }

}
```

Listing 8.5: The representation `AL`.

```
1  public class AL implements IL {
2
3    List<Event> lst;
4    ListIterator<Event> it;
5
6    public AL(){
7      lst = new ArrayList<Event>();
8      it  = lst.listIterator();
9    }
10
11   ...
12
13 }
```

Listing 8.6: The representation `LL`.

```
public class LL implements IL {

  List<Event> lst;
  ListIterator<Event> it;

  public LL(){
    lst = new LinkedList<Event>();
    it  = lst.listIterator();
  }

  ...

}
```

Concretely we define two new classes `AL` (based on `ArrayList`) and `LL` (based on `LinkedList`). The implementations of both `AL` and `LL` are straightforward and are omitted from listings 8.5 and 8.6. Both `AL` and `LL` implement the same interface `IL`. `IL` provides, for illustrative reasons, a smaller interface than provided by Java's `List` (see listing 8.3). `IL` is specialised to accept elements of the type `Event` (see listing 8.4) and incorporates (only) *one* dedicated iterator per list instance (cf. `it` in listings 8.5 and 8.6 and the operations on lines 8–10 in listing 8.3). We explain this design decision in section 8.3.2.2. Finally, the operation `insert` is used to add a new element to the list, just before the current position of the iterator. All this is implemented in listings 8.5 and 8.6, by delegating the operations to either `lst` (line 3) or `it` (line 4).

Secondly, we define the just-in-time class `JL` (see listing 8.7) which combines the two representations, `AL` and `LL`. `JL` defines both possible transition functions (lines 4–7 and 9–12), which are straightforward: all elements from the `source` list are copied to a new list in `target` and the iterator is (re)initialised.[3]

[3] To be more precise, the implementation should change the position of the iterator also.

Listing 8.7: The just-in-time class JL.

```
1  class JL implements IL
2  combines AL, LL {
3
4    AL to LL {
5      target.lst = new LinkedList<Event>( source.lst );
6      target.initIterator();
7    }
8
9    LL to AL {
10     target.lst = new ArrayList<Event>( source.lst );
11     target.initIterator();
12   }
13
14   #write as insert(Entry e);
15   #read  as get(int i);
16
17   swaprule AL {
18     int count = #write + #read;
19     if( (count>100) && (( #write / (double)(#read)) > 0.75) ) {
20       this to LL;
21     }
22   }
23
24   swaprule LL {
25     int count = #write + #read;
26     if( (count>100) && ((#read / (double)(#write)) > 0.50) ) {
27       this to AL;
28     }
29   }
30
31 }
```

Further, JL introduces two *invocation counters*. On line 14 #write is introduced to count the number of inserts. On line 15 #read is introduced to count the number of get operations. The counters are used by the two *internal swap rules*. The first internal swap rule (lines 17–22) states that LL is the preferred representation whenever the ratio of writes to reads exceeds 75%. The second internal swap rule (lines 24–29) states that AL is the preferred representation whenever the ratio the ratio of reads to writes exceeds 50%.

Avoiding the Problem of Escaping Pointers In this chapter, we use the compiler that translates JITds code into plain Java. We use this implementation of JITds because it gives us the full power of Java, i. e., we can reuse all built-in types and standard libraries. Recall from chapter 7 that the Java implementation suffers from the problem of escaping pointers (which was solved in the JITds-C implementation). In short, this problem arises when a representation (directly or indirectly) returns a reference to itself. When the representation of the just-in-time object changes, then conceptually, the reference to the

old representation has become stale, while it is still reachable. One concrete example in which the problem of escaping pointers occurs is in the iterator pattern, because an iterator is tightly coupled with the representation of the data it is iterating over, i. e., it contains a pointer to the actual list *implementation*. In this example, we *avoid* the problem of escaping pointers by making the iterator an integral part of the representation, and thus it can not escape. When the representation changes, then the transition function updates the iterator as well.

8.3.2.3 Experiment

Listing 8.8 shows a micro-benchmark program that detects and processes mouse events. Mouse events are represented by the class `Event`, which is a simple data class containing an x-coordinate, a y-coordinate, and a timestamp. The detection of mouse events is simulated by the method `generateEvent`, which creates an `Event` with a random x-coordinate and a random y-coordinate. Macroscopically, the timestamps of the generated `Events` seem to be increasing. To simulate that events can be detected out of order, we introduce a random factor (see line 13) that results in small local variations.

The actual program consists of two phases. In the first phase, implemented by `build`, the program "detects events" and inserts them in a list, such that the events are stored in order of decreasing timestamps, i. e., the iterator is advanced until it points to an element whose timestamp is larger than the timestamp of the received event (cf. insertion sort). In the second phase, implemented by `sample`, an estimate of the average mouse position is computed based on a random sample set of 15% of the list. The first phase is designed to be update-dominated, and the second phase is access-dominated.

In our experiment we compare three variations:

AL A plain Java version of the program that uses an instance of `AL` to store the events.

LL A plain Java version of the program that uses an instance of `LL` to store the events.

JL A JITds version of the program that uses an instance of `JL` to store the events, and which dynamically switches between its representations.

8.3.2.4 Results

For each of the variations we let the number of events in the list vary from 1000 to 25000 (in steps of 1000). The results of running all variations 30 times are presented in figures 8.3 and 8.4. All data points are the median of these 30 runs. The first two graphs, figures 8.3a and 8.3b, show the execution time of the first and the second phase, respectively. These results confirm our assumptions, listed in table 8.1, about the performance characteristics of `AL` and `LL`: `LL` is the most efficient representation in the first phase, whereas `AL` excels in the second phase. The JITds variant's execution time for either phase is comparable to the fastest variant in plain Java for that phase. The difference in execution time between the fastest Java implementation and the JITds variant can be attributed to the overhead of counting operations and the level of indirection from the implementation by forwarding.

Listing 8.8: The micro-benchmark program to test JL.

```
1  public class ListExperiment {
2
3    static long timestamp = 10050;
4    static final Random r = new Random();
5
6    public static void run(IL lst, int n){
7      lst = build(lst, n);
8      System.out.println( sample(lst) );
9    }
10
11   public static Event generateEvent(){
12     timestamp++;
13     return new Event(timestamp+(r.nextInt(100)-50),
14       R.nextInt(10),
15       R.nextInt(10)
16     );
17   }
18
19   public static IL build(IL lst, int size){
20     for (int i=0; i<size ; i++){
21       Event next = generateEvent();
22       lst.initIterator();
23       while (lst.hasNext() && lst.next().timestamp>next.timestamp );
24       lst.insert(next);
25     }
26     return lst;
27   }
28
29   public static double[] sample(IL lst){
30     int samples = lst.size()/15;
31     long sumX = 0, sumY = 0;
32     for (int i = 0; i<samples ; i++){
33       Event e = lst.get(r.nextInt(lst.size()));
34       sumX+= e.x;
35       sumY+= e.y;
36     }
37     return new double[]{sumX/(double)samples,sumY/(double)samples};
38   }
39
40 }
```

Figure 8.4 shows the execution times of the complete program, i. e., the aggregate of both phases. As expected, `JL` has the lowest execution time because `JL` combines the best qualities of both representations by changing its representation when the equations in the swap rules hold. Note that the boundary of the phases is **not** expressed as a part of the application logic, but rather as a property of the data structure.

8.3.2.5 Code Comparison

Figure 8.7c shows three versions of a program that first populates a list with events (inserts) and then randomly samples that list (random access).

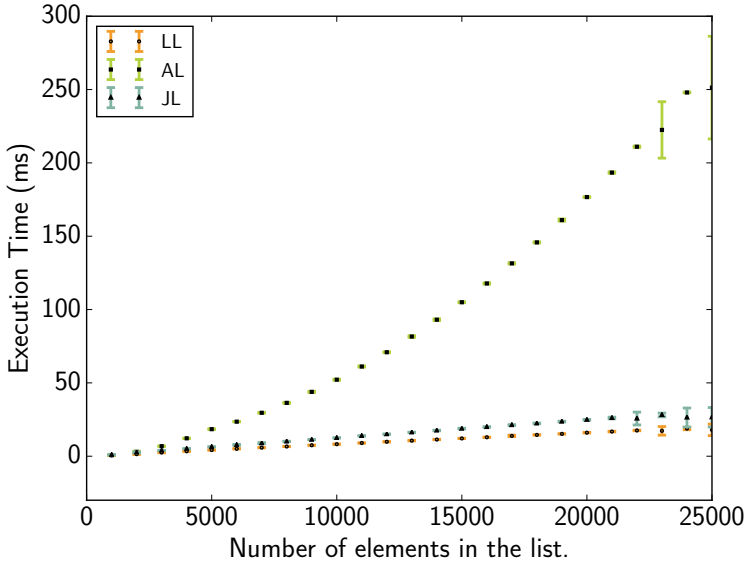
The first (from left to right) version uses JITds, which automatically changes the representation from `AL` to `LL` (or vice versa), depending on the detected usage pattern (cf. internal swap rules). The second version basically implements the same as the first version in plain Java and dynamically changes the representation, but without the expressive benefit of a just-in-time class to hold all the logic related to transforming and swapping. Concretely, the second version manually maintains two counters (i. e., `cR` for reads and `cW` for writes), and changes the representation based on them. In the thirds version the preferred representations are hardcoded, by means of a local representation change, before each phase (see in the `run` method in figure 8.7c). Concretely, version three uses `LL` in `build` and `AL` in `sample`.

The first and second versions implement the exact same functionality, and we compare them first. In the version that uses JITds, application logic ( ) is completely separated from any choice of representation, i. e., transformation logic ( ) and representation change incentive code ( ) are combined into one just-in-time class definition ( ). This is not possible in plain Java, hence we need to clutter the code with guards that checks whether or not to change the representation.

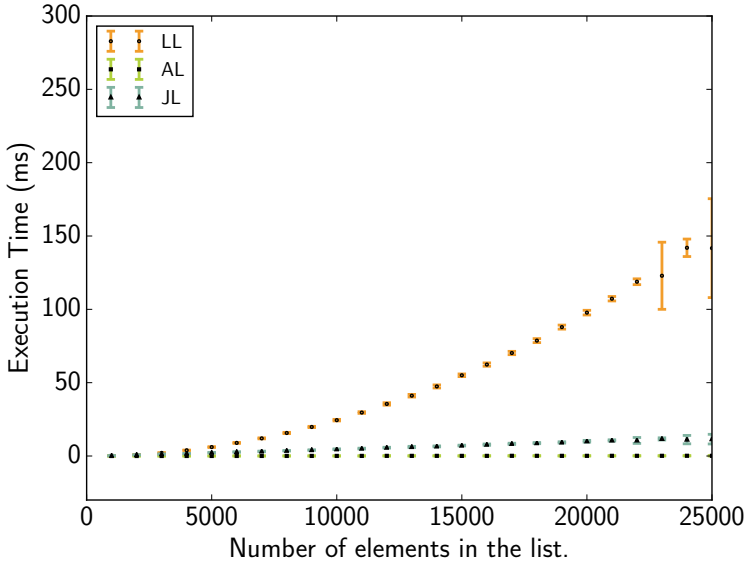
The last version is arguably cleaner than the second version and clearly shorter than the first two versions. There is, however, a fundamental difference between the first two versions and the last version in terms of the computation they perform. The first two versions detect the phase shift automatically based on the ratio of counters (invocation counters in the first version and simple integers in the second version). In the last version, the phase shift is considered to be known a-priori and the (local) representation changes are hardcoded before the invocations of the methods `build` and `sample`. This is only possible when the boundary of the phases is known a-priori and lexically determinable (e. g., before the calls to `build` and `sample`). Thus, the first two versions need bookkeeping (cf. the counter) which is not needed in the last version, hence the difference in length.

8.3.2.6 Conclusion

JITds facilitates the implementation of `JL`, a just-in-time class that exploits the strong points of both `ArrayList` and `LinkedList`, while it avoids the weak points of both representations. We observe that the iterator pattern is not well suited to be used in JITds. Finally, this example program shows how *invocation counters* and *internal swap rules* combined can detect a change in a data interface usage pattern without knowing



(a) Execution time for building a list of events.



(b) Execution time for randomly sampling 15% of the events.

Figure 8.3: Building or sampling a list of events.

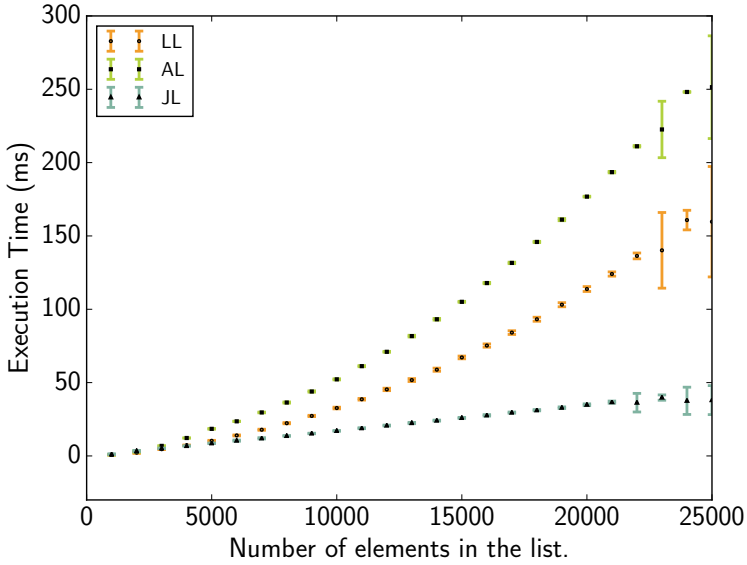


Figure 8.4: Execution time for building and sampling a list of events.

lexically where the phase shift occurs (cf. dynamic quantification in AOP). The implementation in JITds allows the developer to clearly separate the application logic from the choice and change of representations.

8.3.3 The File Program

We call the third example program "The File Program". The File Program models the abstract data type file, that can be in one of three states. Depending on the current state, a file can perform different tasks, e. g., a file can only be written to when it is open. This program uses specialised swaps and named transition functions to realise this behaviour without the need for encoding and checking the state explicitly.

8.3.3.1 Context: Changing Functionality

The main motivation for developing JITds is to support non-functional representation changes. However, it is also possible to implement *functional representation changes* in JITds. The most common scenario for functional representation changes is to model *objects with modes*, object evolution [13], or objects with protocols [78]. In these scenarios the developer wants to change the behaviour of an object depending on what *state* that object is in. This has led to the extension of the object-oriented programming paradigm called typestate-oriented programming [3]. The motivating example for typestate-oriented programming is a file that may be open, closed, or locked. An open

Listing 8.9: A closed file.

```
class ClosedFile  {

  String path;


  void setPath(String path) {
    this.path = path;
  }

}
```

Listing 8.10: An open file.

```
class OpenFile {

  String path;
  FilePointer filePtr;

  void write(String str){
    filePtr.write(str);
  }

}
```

Listing 8.11: A locked file.

```
class LockedFile {

  String path;

}
```

file may be written to, a closed file can change its path, and a locked file has a fixed path and can not be written to.

8.3.3.2 Implementation

In this program we conceptually model a `File` with three *states* (i. e., open, closed and locked as in [78]) and leave the actual I/O handling to the underlying Java libraries. To clearly separate the actual I/O functionality from the conceptual part, we implemented the class `FilePointer` which wraps the functionality of `FileWriter` (a basic file I/O class from Java). A `FilePointer` opens a connection with an actual file when created (constructor) and closes this connection when the method `close` is called. Text can be written to the file between construction and closing by invoking the method `write`.

The just-in-time class `File` combines three representations, i. e., one representation per state: `ClosedFile` (see listing 8.9), `OpenFile` (see listing 8.10), and `LockedFile` (see listing 8.11). In the JITds program, however, these three classes are not supposed to be used as applicationlevel types (cf. implementation-only classes [15]). One should always use an instance of the just-in-time class `File` (see listing 8.12). A just-in-time `File` is initially in the closed state, because `ClosedFile` is the first in the list of representations. Only certain transition functions are defined in `File`. `File` defines the transitions from closed to open (copy the path and create an open connection to a file), from closed to locked, and from open to closed (which copies the path and closes the open connection). This design implies that locked files can never be opened again.

All the methods inherited by `File` from `ClosedFile`, `OpenFile`, and `LockedFile`, are *specialised methods*. Thus any method invocation potentially causes a *specialised swap*.

8.3.3.3 Experiment and Results

For this example we do not present a performance evaluation. Instead, our goal is to show how a typical typestate-oriented program can be expressed in JITds. The code fragment in listing 8.13 shows how to create a `File f` (line 1) at location `tmp/data.txt` (line 2), to which the word “test” is written (line 3), and which is finally closed (line 4) and locked (line 5). A file does not need to be opened explicitly in order to be able to write to it, because write is a specialised method of the `OpenFile` class. Therefore, a call to `write` initiates a representation change (cf. specialised swap). Once the file is locked,

Listing 8.12: The just-in-time class `File`.

```
1  class File
2  combines ClosedFile, OpenFile, LockedFile {
3
4    ClosedFile to OpenFile as open {
5      target.path = source.path;
6      target.filePtr = new FilePointer(source.path);
7    }
8
9    ClosedFile to LockedFile as lock {
10     target.path = source.path;
11   }
12
13   OpenFile to ClosedFile as close {
14     target.path = source.path;
15     source.filePtr.close();
16   }
17
18 }
```

Listing 8.13: A fragment of Java code to write to a `File`.

```
1  File f = new File();
2  f.setPath("tmp/data.txt");
3  f.write("test");
4  f.close();
5  f.lock();
```

Listing 8.14: An alternative fragment of Java code to write to a `File`.

```
1  File f = new File();
2  f.setPath("tmp/data.txt");
3  f.open();
4  f.write("test");
5  f.close();
```

this transition is no longer possible and writing to it results in an `UnsupportedSwapException`. In the alternative implementation (see listing 8.14), the file is explicitly opened (line 3).

8.3.3.4 Code Comparison

In figure 8.8b, we compare the JITds code (left) and a plain Java implementation (right). A qualitative comparison between the JITds program and other typestate-oriented programs (i. e., in the language Plaid [78]), is given in section 9.1.5.

The biggest difference between between the two versions of the code in figure 8.8b is that in the plain Java version, all invariants that have to do with the current state, and thus the current representation, have to be manually encoded (■ and ■), checked (■), and maintained (■ and ■). Conversely, in the JITds version these invariants are automatically maintained by the just-in-time class (■) and all transformation logic (■) is grouped.

For instance, the invariants in the plain Java version are that `filePtr` in a closed file is `null`, that `filePtr` in an open file is not `null`, and that only locked files have `true` assigned to the field `locked` (cf. flag field). Behaviourally, only closed files can change

their path, only open files can be written to, and locked files are locked forever. The first set of invariants have to be manually maintained in the Java version, and the second set of invariants have to be manually checked in the Java version.

In JITds, on the other hand, the current state is reified by the current representation, hence there is no need to check any of the invariants. A closed file in the JITds program, for instance, does not even have a file pointer and a file that is able to execute the `write` method is known to be in the open state. These invariants are maintained automatically by JITds. Finally, the `LockedFile` representation does not need to explicitly encode that it is locked (cf. the flag field in the Java version), because the state is reified by the current representation.

8.3.3.5 Conclusion

Without much effort it is possible to use JITds as a typestate-oriented programming language. In other words, JITds does not only support *non-functional representation changes*, but also *functional representation changes*. How JITds relates to other typestate-oriented programming languages is discussed in chapter 9. Compared to plain object-oriented languages, the advantage of using typestate-oriented programming is that the behavioural aspects of the current state do not need to be encoded via design patterns (e. g., state [34]), if-tests on flags (e. g., `locked`), or any other indirect implementation strategy [3].

8.3.4 The Sorting Program

We call the fourth example program "The Sorting Program". The Sorting Program fills an array with random data, sorts the array, shuffles some elements, and re-sorts the array. This program uses an interface swap rule to decide which algorithm to use for sorting. The decision is made based on a characteristic of the data, here the *sortedness* of the array.

8.3.4.1 Context: Data Specialisation

Often, multiple algorithms exist to accomplish the same task. Their performance changes based on the *characteristics of the input data*. Similarly, different storage strategies for the same data type may exist, e. g., recall the `SparseMatrix` representation from section 2.2, which only pays off when the matrix is actually sparse, where "being sparse" is a *characteristic of the input data*. *Data specialisation* is the scenario where a dedicated implementation exists for data with a specific property.

It is known that the time complexity of insertion sort is $O(n^2)$ and that the time complexity of quick-sort is $O(n \log n)$. Hence, the best algorithm is quick-sort. However, this is the *asymptotic* and *average case* performance. In the context of sorting, characteristics of the data are, for instance, input size, *sortedness*, distribution, and value range. Depending on the characteristics of the data, asymptotic average case performance estimates can be completely wrong, and relying on the average case can be a significant

Listing 8.15: The just-in-time class `Sorter` combines three sorting algorithms.

```
1  class Sorter
2  combines InsertionSorter, MergeSorter, QuickSorter {
3
4    QuickSorter to InsertionSorter { target.data = source.data; }
5    QuickSorter to MergeSorter     { target.data = source.data; }
6
7    InsertionSorter to QuickSorter { target.data = source.data; }
8    InsertionSorter to MergeSorter { target.data = source.data; }
9
10   MergeSorter to QuickSorter     { target.data = source.data; }
11   MergeSorter to InsertionSorter { target.data = source.data; }
12
13   swaprule boolean Sorter.sort(int lo, int hi) {
14     boolean large  = (hi-lo) > 5000;
15     double[] data  = receiver.getData();
16
17     if ( large  && Utils.sortedness(data, lo, hi)>0.90 ) {
18       receiver to InsertionSorter;
19     } else {
20       receiver to QuickSorter;
21     }
22     proceed;
23   }
24 }
```

overestimate or underestimate. This example is based on these insights and is adapted from the motivating example from [6].

8.3.4.2 Implementation

In this example program we evaluate a data structure that maintains an array of numbers (`double`) that can be sorted by calling the method `sort(int lo, int hi)`. We implement three different representations of this data structure, i. e., `InsertionSorter`, `MergeSorter`, and `QuickSorter`, that use insertion sort, merge sort, and quick-sort to sort the array. We refrain from providing the concrete implementations because these are well-known algorithms.

The just-in-time class `Sorter` (see listing 8.15) combines the three sorting classes. This implies that changing the representation of a `Sorter` object effectively changes the algorithm that is used to sort the data. `Sorter` defines all six transition functions.[4] `Sorter` also defines one *interface swap rule*, which intercepts all calls to `sort` and makes the choice of which representation to use. The choice shown in listing 8.15 is to use insertion sort for *small* and *nearly-sorted* data sets, and to use quick-sort otherwise.

[4] These six transition functions are very similar and could be replaced by one *generic transition function*: `Sorter to Sorter target.data = source.data;`. The current version of the JITds-Java compiler, however, does not implement the syntactic sugar construct of generic transition functions, as described in chapter 4

Listing 8.16: Filling an array with random numbers.

```
1  void fill(double[] data) {
2    Random rnd = new Random();
3    int size = data.length;
4    for (int i=0; i<size ; i++){
5      double d = rnd.nextDouble();
6      data[i] = d*100;
7    }
8
9  }
```

Listing 8.17: Swapping three elements in a (sorted) array.

```
  void shuffle(double[] data) {
    Random rnd = new Random();
    int size = data.length;
    for (int i=0; i<3 ; i++) {
      int a = rnd.nextInt(size);
      int b = rnd.nextInt(size);
      swap(data, a, b);
    }
  }
```

8.3.4.3 Experiment

In this experiment we compare the time it takes to sort using the different classes (i. e., `InsertionSorter`, `MergeSorter`, `QuickSorter`, and `Sorter`(jit)). We do this once for a *random sequence* and once for a *nearly-sorted sequence*. To obtain a measurable effect we need sufficiently large sequences. Hence, we let the size of the sequences vary from 1000 to 20000 elements in steps of 1000. As shown in listing 8.16, the random sequence is built using Java's random number generator and creates elements in the range of 0 to 100 in double float precision. Listing 8.17 shows how we swap a few elements in a sorted sequence to obtain a nearly-sorted sequence.

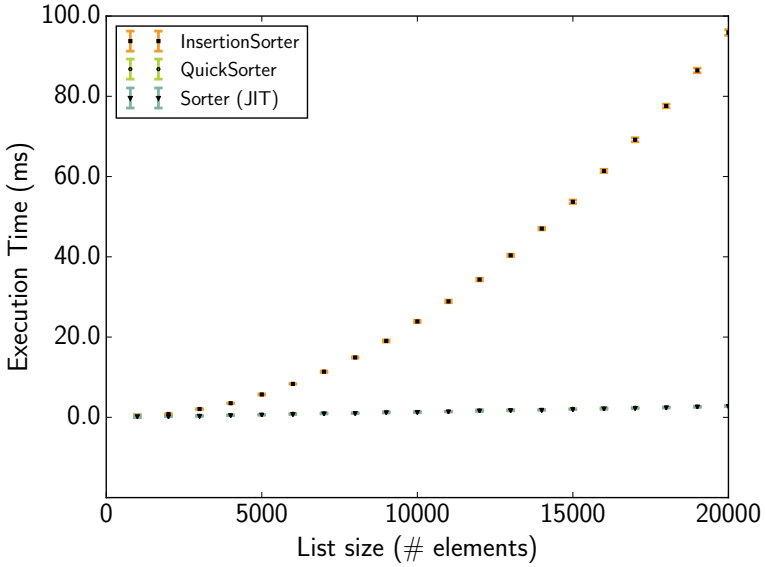
8.3.4.4 Results

The results of this experiment are shown in figure 8.5: figure 8.5a shows the time needed to sort a random sequence and figure 8.5b shows the time needed to sort a nearly-sorted sequence. All data points are the arrhythmic mean of execution times, gathered over 30 runs. We first focus on the two plain Java implementations that use `InsertionSorter` and `QuickSorter`. We see that, for the random sequence, `QuickSorter` performs better than `InsertionSorter`, however, when the sequence is nearly-sorted, the inverse is true. This aligns with the known performance characteristics of both sorting algorithms. If we look at the just-in-time class `Sorter`, we see that it behaves similarly to the best strategy for either case, i. e., `QuickSorter` for random and `InsertionSorter` for nearly-sorted sequences, as excepted[5].

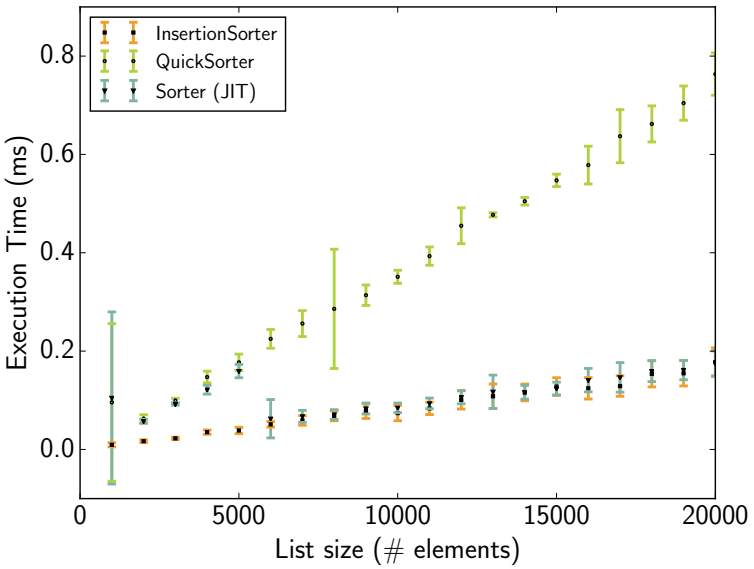
8.3.4.5 Code Comparison

Figure 8.7b shows three versions of the program that populates, sorts, shuffles, and resorts an array of doubles. The leftmost version relies on JITds to automatically change the sorting strategy based on an interface swap rule. The middle version implements the exact same functionality in Java. This version changes the sorting algorithm by implementing the strategy pattern [34]. The rightmost version hardcodes what sorting algo-

[5] The data points gathered for `MergeSorter` are of no specific interest to our conclusions and are thus omitted from the graphs.



(a) Time needed to sort a random sequence.



(b) Time needed to sort a nearly-sorted sequence.

Figure 8.5: Sorting a sequence using different algorithms.

rithm to use (■), first sorting a random sequence with quick-sort and then using insertion sort to sort a nearly-sorted sequence. The difference with the first two versions is that the choice of algorithm is made without checking the characteristics of the data to be sorted.

We conclude that JITds allows a clean separation of concerns when implementing a scenario where the representation is changed based on data characteristics. Furthermore, in this example where only computation is changed, JITds and the strategy pattern yield comparable code (w.r.t. code cluttering). The last version yields shorter code, but is less generic (hardcoded choice) and more cluttered.

8.3.4.6 Conclusion

JITds allows developers to clearly separate application logic from representation change logic when implementing a scenario where the representation change is based on the characteristics of the input data (i. e., data specialisation). This way, an ad-hoc implementation of the strategy pattern can be avoided while improved performance can be obtained at the same time. Both JITds and the strategy pattern result in code that has a similar structure, and is not entangled.

8.3.5 The String Program

We call the fifth example program "The String Program". The String Program creates a long piece of text and then computes the set of prefixes. This program combines a mutable and an immutable representation for text into a single just-in-time class and uses named transition functions to freeze or thaw the data structure.

8.3.5.1 Context: Freezing and Thawing

A common pattern in data structure design is that some data structure implementations perform best when building the data structure (mutable) and that some data structure implementations perform best when the data structure is being queried (immutable). Examples of these different designs in Java are `StringBuilder` and `String`, which are both implementations of `CharSequence`, or text.

`String` is an immutable data structure and all updates to its content result in an update to a *copy of the data*. `StringBuilder` is a mutable data structure and all updates are made *directly on the data*.

Hence, it is best practice in Java to use `StringBuilder` when creating a new, evolving piece of text; and to switch to `String` once the content is complete. This pattern is known as *freezing and thawing* data structures.

8.3.5.2 Implementation

In this example program we implement a just-in-time class `JitString` (see listing 8.18) that incorporates both representations: the immutable `String` (frozen) and the mutable `StringBuilder` (thawed). The data interfaces of `String` and `StringBuilder` are not perfectly aligned. In Java, a programmer can concatenate `String` instances using the dedicated string concatenation operator `+` (or with the method `concat`). To

Listing 8.18: The just-in-time class `JitString`.

```
1  class JitString implements StringInterface
2  combines ThawedString, FrozenString {
3
4    ThawedString to FrozenString as freeze {
5      target(source.value, 0, source.count);
6    }
7
8    FrozenString to ThawedString as thaw {
9      target();
10     target.append(source);
11   }
12
13 }
```

concatenate instances of `StringBuilder` the method `append` has to be used. Because we need uniformity between the data interfaces in our example, we reimplement `String` and `StringBuilder` to both implement a `concat` method and a `subSequence` method with the same signatures. To this end, we introduce two representations, `FrozenString` and `ThawedString`. `FrozenString`, on the one hand, is a `String` derivative that copies itself upon concatenation and that provides an immutable view of its content when `subSequence` is called. `ThawedString`, on the other hand, is a `StringBuilder` derivative that updates its own content upon concatenation and that returns an immutable copy of a part of its data when `subSequence` is called.

Listing 8.18 shows the implementation of `JitString` that provides both the transition function to freeze and to thaw the text. Note that both transition functions are *named transition functions*, i. e., `freeze` and `thaw`, respectively. Even though it would be possible, `JitString` does not implement any swap rules to detect a change in interface usage pattern and leaves the responsibility for changing the representation to the user of the `JitString`. The List Program (see section 8.3.5) describes the scenario where such a pattern is detected an reacted upon.

Avoiding the Problem of Escaping Pointers The current implementation of the JITds-Java compiler suffers from the problem of escaping pointers. Besides the list iterator example, this string program is a second example where the problem needs to be addressed. Here, the problem of escaping pointers occurs when a representation returns `this`, which is the case for `concat` of `ThawedString`.

Listing 8.19: Code generated by the JITds-Java compiler for the `concat`-method for instances of `JitList` in the `ThawedString` representation.

```
1  public StringInterface concat(StringInterface s) {
2    if ( invokeInnerSwaprules() ) {
3      return instance.concat(s);
4    } else {
5
6      return super.concat(s);
```

```
7     }
8   }
```

Listing 8.20: Manual intervention in the code generated by the JITds-Java compiler.

```
1   public StringInterface concat(StringInterface s) {
2     if ( invokeInnerSwaprules() ) {
3       return instance.concat(s);
4     } else {
5       super.concat(s);
6       return JitString.this; //manual
7     }
8   }
```

In this example, we *resolve* the problem of escaping pointers by making a small manual intervention to the code generated by the JITds-java compiler. Listing 8.19 shows the original code generated by the JITds-Java compiler for the `concat` method for instances of `JitList` in the `ThawedString` representation. The problem originates at line 6, where `super.concat(s)` returns `this`. Because of the implementation by forwarding, `this` actually denotes the instance currently being forwarded to, whereas semantically a JITds program is expected to return the complete just-in-time object. Listing 8.20 shows that, with one extra line of manually added code, the problem (in this case) is resolved. Thus, line 5 takes care of forwarding the call to the current implementation, which is known to return `this`. We then manually return `JitString.this` on line 6. When using the JITds-C compiler the problem of escaping pointers does not occur, but then we would introduce a complete new evaluation set-up for a single example. We choose to stick to the manual intervention for the sake of uniformity.

8.3.5.3 Experiment

In this example, the program has two phases. In the first phase, i. e., the method `build` (see listing 8.24), a text is created by concatenating characters. In the second phase, i. e., the method `prefixes` (see listing 8.25), an array is created that contains all prefixes of that text. This program is constructed such that the first phase is a building phase, and the second phase is a querying phase.

In our experiment we compare three variations:

`FrozenString` A plain Java implementation that uses a `FrozenString` instance in both phases (listing 8.21).

`ThawedString` A plain Java implementation that uses a `ThawedString` instance in both phases (listing 8.22).

`JitString` A JITds implementation that uses a `JitString` instance in both phases, with a *swap statement* between the two phases (cf. line 3 in listing 8.23).

To obtain a measurable effect we need sufficiently large sequences. Hence, for each of the three versions we build a string with a size that varies between 1,000 and 50,000 (in steps of 1,000). All data points are the median of 30 runs.

Listing 8.21: Example: `FrozenString`

```
1  run(FrozenString str, int n)
2    str = build(str , n);
3
4    return prefixes(str);
5  }
```

Listing 8.22: Example: `ThawedString`

```
run(ThawedString str, int n)
  str = build(str , n);

  return prefixes(str);
}
```

Listing 8.23: Example: `JitString`

```
run(JitString str, int n)
  str = build(str , n);
  str.freeze();
  return prefixes(str);
}
```

Listing 8.24: Building a string.

```
1  public static StringInterface build(StringInterface str, int n){
2    for (int i=0 ; i<n ; i++){
3      str = str.concat(Integer.toString(i%10));
4    }
5    return str;
6  }
```

Listing 8.25: Gathering all prefixes of `str` in `data`.

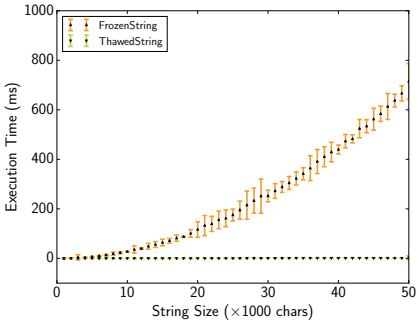
```
1  public static StringInterface[] prefixes(StringInterface str){
2    int count = str.length();
3    StringInterface[] data = new StringInterface[count];
4
5    for (int size = 1 ; size<count ; size++){
6      data[size-1] = str.subSequence(0, size);
7    }
8
9    return data;
10 }
```

8.3.5.4 Results

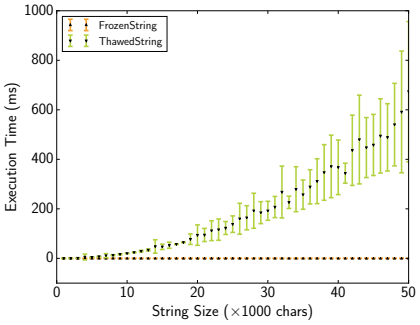
As expected, `ThawedString` is the ideal representation for constructing new strings (figure 8.6a) and `FrozenString` is the ideal representation for querying the string (figure 8.6b). Initially, a `JitString` is in the thawed representation. Because we instruct the `JitString` to `freeze` between the two phases, it can be seen in figure 8.6c that `JitString` outperforms both `FrozenString` and `ThawedString` in the complete program, since it relies on the "best" representation for each phase.

8.3.5.5 Code Comparison

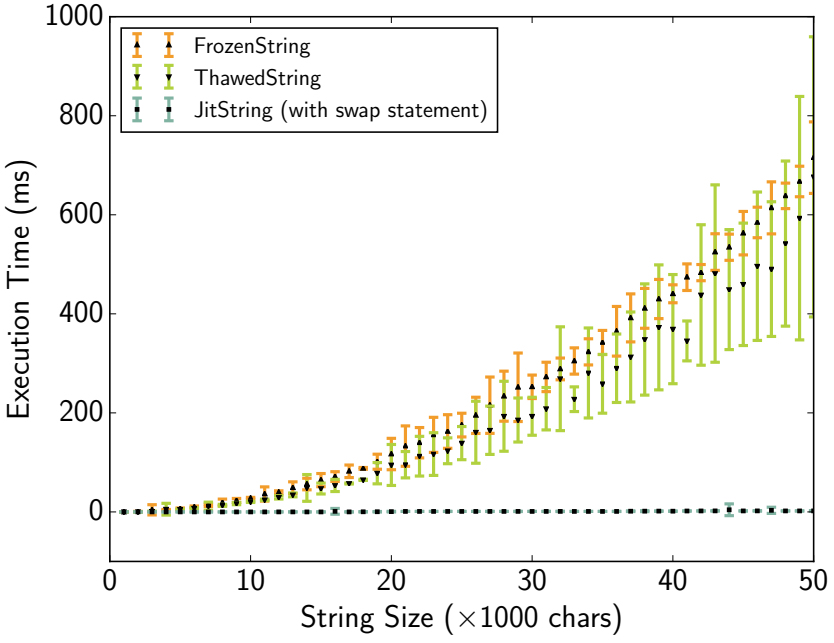
A shown in figure 8.8a, both in the classic (Java) version as well as in the JITds version, the instruction to freeze the data structure is part of the application code. This is to be expected as this is what the freeze-and-thaw pattern looks like: an update phase, a freeze statement, and then a query phase; or vice versa.



(a) Building a string.



(b) Computing the prefixes of a string.



(c) Building and computing the prefixes of a string.

Figure 8.6: Building or computing the prefixes of a string.

JITds features	Matrix	List	File	String	Sorting
Combining representations	✓	✓	✓	✓	✓
Transition functions	✓	✓	✓	✓	✓
Named transition functions	✗	✗	✓	✓	✗
Swap statements	✓	✗	✗	✓	✓
Specialised swap	✗	✗	✓	✗	✗
External swap rules	✓	✗	✗	✗	✗
Internal swap rules	✗	✓	✗	✗	✗
Interface swap rules	✗	✗	✗	✗	✓
Invocation counters	✗	✓	✗	✗	✗
First class representations	✓	✗	✗	✗	✗
Compiler limitations					
Escaping pointers		⚠		⚠	

Table 8.2: A comparison of the different implementations of JITds in function of the available features.

8.3.5.6 Conclusion

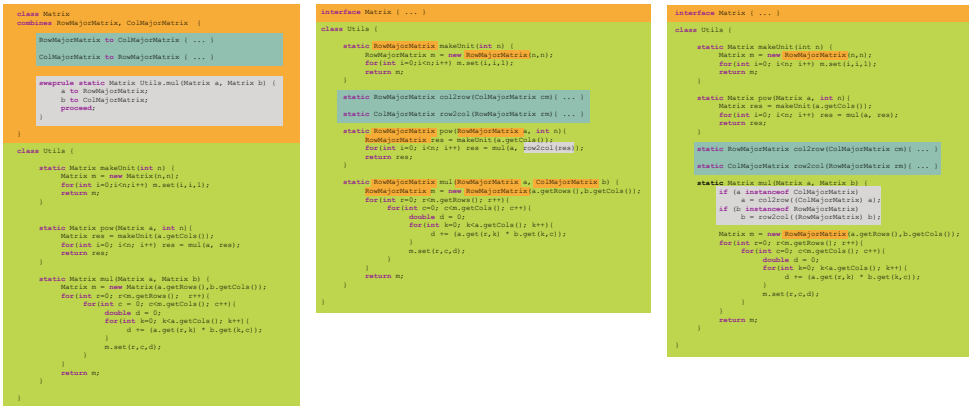
Using JITds to implement the unification of a frozen and thawed version of a data type yields the expected performance gains. The code of the JITds version and the classic (manual) version are similar (up to the definition of the just-in-time class).

8.4 Conclusions

Table 8.2 gives an overview of the JITds features used in each of the five example programs. For instance, the file example (see section 8.3.3) is the only example program that relies on *specialised swaps*. In a set of five programs, we were able to use most of the JITds features. Only *transient state* and *by-pass methods* are not used in this chapter, because these two features have not been implemented in either of our two compilers (see chapter 7). Furthermore, in two of the five programs we needed to manually address the problem of escaping pointers, a limitation of the current version of the JITds-Java compiler.

In four of the five examples (i. e., all except the file program which is not about gaining performance) just-in-time data structures improved performance as expected. The focus of this evaluation is not to evaluate the actual performance gains, but rather that JITds is sufficiently expressible to be applied in the scenarios described in chapter 3.

We also evaluated how, in JITds code, application logic and representation change logic are intertwined. When the user *explicitly* transitions between representations, then the application and representation logic are intertwined, e. g., in the the freeze-and-thaw pattern. This is to be expected, because there the representation change incentive code is arguably an inherent part of the application logic. When evaluating the entanglement of application logic with data representation logic in all other JITds programs, we see an apparent pattern emerging: the just-in-time class combines multiple representations



(a) Three versions of matrix-multiply-power.

(b) Three versions of the sorting program.

(c) Three versions of list-build-query program.

```
class JitString implements StringInterface
combines ThawedString, FrozenString {

    ThawedString to FrozenString as freeze {
        target(source.value, 0, source.count);
    }

    FrozenString to ThawedString as thaw {
        target();
        target.append(source);
    }

}
```

```
class Utils {
    static StringInterface[] run(JitString str, int n)
        str.thaw();
        str = build(str , n);
        str.freeze();
        return prefixes(str);
    }

    static StringInterface build(StringInterface str, int n){
        for(int i=0 ; i<n ; i++){
            str = str.concat(Integer.toString(i%10));
        }
        return str;
    }

    static StringInterface[] prefixes(StringInterface str){
        int count = str.length();
        StringInterface[] data = new StringInterface[count];

        for(int size=1; size<count; size++){
            data[size-1] = str.subSequence(0, size);
        }

        return data;
    }
}
```

```
class Utils {
    static StringInterface[] run(StringInterface str, int n)
        str = new ThawedString(str);
        str = build(str , n);
        str = new FrozenString(str);
        return prefixes(str);
    }

    static StringInterface build(StringInterface str, int n){
        for(int i=0; i<n; i++){
            str = str.concat(Integer.toString(i%10));
        }
        return str;
    }

    static StringInterface[] prefixes(StringInterface str){
        int count = str.length();
        StringInterface[] data = new StringInterface[count];

        for(int size=1; size<count; size++){
            data[size-1] = str.subSequence(0, size);
        }

        return data;
    }
}
```

(a) Two versions of the string building/querying program.

```
class ClosedFile  {
    String path;

    void setPath(String path) {
        this.path = path;
    }
}

class OpenFile {
    String path;
    FilePointer filePtr;

    void write(String str){
        this.filePtr.write(str);
    }
}

class LockedFile {
    String path;
}

class File
combines ClosedFile, OpenFile, LockedFile {

    ClosedFile to OpenFile as open {
        target.path = source.path;
        target.filePtr = new FilePointer(source.path);
    }

    ClosedFile to LockedFile as lock {
        target.path = source.path;
    }

    OpenFile to ClosedFile as close {
        target.path = source.path;
        source.filePtr.close();
    }

}

void run(){
    File f = new File();
    f.setPath("tmp/data.txt");
    f.open();
    f.write("test");
    f.close();
}
```

```
class File {

    String      path    = null;
    FilePointer filePtr = null;
    boolean     locked  = false;

    void setPath(String path) {
        if (filePtr == null) {
            this.path = path;
        }
    }

    void write(String str){
        if (filePtr != null) {
            this.filePtr.write(str);
        }
    }

    void close(){
        if (filePtr != null) {
            this.filePtr.close();
            this.filePtr = null;
        }
    }

    void open() {
        if (!locked && path != null) {
            this.filePtr = new FilePointer(path);
        }
    }

    void lock() {
        if (filePtr != null) close();
        this.locked = true;
    }

}


void run(){
    File f = new File();
    f.setPath("tmp/data.txt");
    f.open();
    f.write("test");
    f.close();
}
```

(b) Two versions of a program that creates a file and writes to it.

(■) and contains a block of transformation logic (■) and a block of representation change incentive code (■). Furthermore, the application code (■) is not entangled with anything else.

Chapter 9

Related Work

In chapter 3 we explore the design space for languages with support for representation changes. In chapters 4 and 6 we introduce JITds and show how JITds relates to the taxonomy of chapter 3. In this chapter we discuss the work related to JITds in considerable detail, i. e., we discuss each effort in relation to JITds, and we discuss each effort in relation to the taxonomy of chapter 3, when applicable.

The efforts discussed in this chapter can be categorised in roughly three partitions. First, in section 9.1, we discuss a set of programming languages and programming paradigms which provide developers with programming features to express representation changes of data objects at runtime. Second, in section 9.2, we discuss a set execution environments, and frameworks that help the developer to choose the right data representation for "collections" (e. g., list, set, array, ...). Finally, in section 9.3, we discuss work related to implicit changes of the computation, as opposed to changing the data representation, at runtime.

9.1 Languages and Paradigms

We now discuss a set of programming languages and programming paradigms where changing the representation of an object is a core feature. This set of programming languages was compiled during the literature study we performed when distilling the taxonomy from chapter 3. We first discuss *Late Data Layout* because it is the most related to JITds in the sense that Late Data Layout also focusses on improving the performance. The other languages are presented in order of genericness, i. e., SmallTalk, the most generic language, is presented first.

9.1.1 Late Data Layout

Late Data Layout (LDL) and Automated Ad hoc Data Representation Transformations (ADRT) are two techniques developed by Ureche et al. [85, 84]. LDL and ADRT focus on statically reducing the number of "expensive" heap allocated objects in favour of "cheap" stack values, in order to minimise execution time. The running example for explaining

ADRT is the Gaussian integer, i. e., a complex number $a + bi$ with $a, b \in \mathbb{N}$. From a software engineering perspective the ideal representation of a Gaussian integer is a *pair* of integers which allows easy access to both the real and the imaginary parts. From a performance point of view, however, the use of heap-allocated structures, such as pairs, introduces unnecessary overhead. Therefore, a performance-conscious programmer might want to encode the *high-level* integer pair into a single *long* value, and thus avoid allocating objects in the heap. Ureche et al. [85, 84] argue that manually minimising the execution time by translating the code from a high-level representation to a low-level representation is a tedious and error-prone activity. Their technique *LDL* combines type propagation with type inference to find those lexical places in a program where coercion between high-level and low-level data representations are needed [85]. Their technique *ADRT* makes such representation transformations expressible by the developer [84]. In short, LDL and ADRT leverage the type system in order to infer where *coercions* between levels of representations need to be inserted.

Both the work of Ureche et al. [85, 84] and our work on JITds incorporate the idea of changing the representation of data objects at runtime, with the key goal to improve performance. In general, the behaviour of LDL is to encode high-level representations (heap-allocated) as low-level representations (stack-based), and let this stack-based value be propagated thought the system. When a method is invoked on a stack-based value, however, it is converted back to its high-level representations, i. e., as a heap allocated object. This is much like the *specialised swap* from JITds, where a data object is coerced into a representation that does support the requested operation.

However, constantly changing back and forth between different representations can ruin potential performance gains. To avoid the overhead of constantly converting back and forth, Ureche et al. [84] propose the use of *bypass methods*. Bypass methods take the receiver object in its encoded format , i. e., stack-based value, as first parameter and provide a behaviourally-equivalent implementation for the original method. A simple example could be a `toString` method: `bypass_toString(x)` instead of `convert(x).toString()`. The bypass methods in JITds (section 4.5.1.3) are inspired by and directly based on the bypass methods in [84].

The techniques presented by Ureche et al. [85, 84] also differ from JITds in some aspects. A first difference is the focus of LDL on the impact on performance of stack-base values versus heap-allocated objects. A just-in-time object, however, is always heap-allocated. Hence, the source of performance improvement when using LDL is the reduction of overhead (allocation, dereferencing) which also translates to faster execution. The performance gains when using JITds, on the other hand, are expected to primarily originate from improvements on the algorithmic level.

A second difference between LDL and JITds is the relation between representations and their associated operations (cf. behaviour and data interface). In LDL, one of the representations, i. e., the heap-allocated value, is associated with a set of operations, whereas the other representation, i. e., the encoded stack value, is not. This implies a fundamental difference in treatment of both representations. In JITds, all representations are equivalent and all representations have a proper set of operations (cf. data interface).

As stated before, in ADRT a value can be stored in one of *two* representations: as a

stack value or as a heap-allocated value. The transformation logic which describes how to transition between the two representations has to be expressed inside of a *transformation description object* (see listing 9.2). This makes ADRT a nice example of an approach that ensures an orderly transformation with external transformation logic. The obligatory components of a *transformation description object* are the `toRepr` and the `toHigh` methods, which respectively describe how to transform from a high-level to a low-level representation and the other way around. Hence, the transition graph that can be derived from a *transformation description object* is always a complete (directed) graph with exactly two vertices. This is the third difference with JITds where transition graphs can be more general. JITds allows for arbitrary complex transitions between an arbitrary number of representations.

ADRT offers programmers a construct to express where representation changes are wanted in the program. This construct, which is called a *transformation scope*, can contain anything ranging from a single expression to an entire class definition. As a result, the transitions introduced by ADRT are limited to statically inferable, i. e., lexical, places only. Thus, the fourth difference between ADRT and JITds is that ADRT allows the developer to express external representation change incentive logic only. JITds, on the other hand, provides constructs to express both internal and external representation change incentives. It might seem like a limitation of ADRT to only support external representation change incentive logic. However, the absence of internal representation logic allows for the *automation* of inserting the statically inferred coercions. Moreover, because ADRT relies on the techniques from LDL, these statically introduced coercions can be shown to be optimal. JITds does not give such guarantees.

9.1.1.1 Example.

A Gaussian integer is a complex number $a + bi$ with $a, b \in \mathbb{N}$. In Scala, the base language of ADRT, such a Gaussian integer (cf. `GaussianInt`) can be represented by a pair of `Int`s. This is shown on line 2 of listing 9.1. Further, the class `GaussianInt` provides two methods, i. e., `%` and `norm` (lines 3 and 4, respectively). The method `gcd` of the object `GCD` can be used to compute the greatest common devisor of two such Gaussian integers using a variation of Euclid's algorithm (see lines 8–11).

In Scala, two `Int`s consume as much space as one `Long`. Hence, a `GaussianInt` could also be represented as a `Long`. Conceptually,however, the pair representation is preferred. From a performance point of view, however, the `Long` representation is preferred because a long can be stack allocated as opposed to pairs that are always heap allocated. A programmer who is optimising for performance, might identify the need to alternate between these two representations.

To describe the relation between the high-level representation and the low-level representation, the LDL developer can define a *transformation description object*. Listing 9.2 shows the signatures of such a transformation description object, with the compulsory coercion methods `toRepr` and `toHigh`. Further, the transformation description object contains a bypass constructor (line 7) and two bypass methods (lines 10 and 11).

Further, a programmer who is optimising for performance might identify the need to use the `Long` representation in the context of the `gcd` method because there, using the

Listing 9.1: `GCD` describes how to compute the greatest common devisor of two Gaussian integers.(code adapted from [84])

```
1  object GCD {
2    implicit class GaussianInt(pair: (Int, Int)) {
3      def %(that: (Int, Int)): (Int, Int) = ...
4      def norm = ...
5    }
6
7    adrt(IntPairAsGaussianInt) {
8      def gcd(n1: (Int, Int), n2: (Int, Int)): (Int, Int) = {
9        val remainder = n1 % n2
10       if (remainder.norm == 0) n2 else gcd(n2, remainder)
11     }
12   }
13 }
```

Listing 9.2: Definition of a transformation description object between a pair of `Int`s and single `Long`. (code adapted from [84])

```
1  object IntPairAsGaussianInt extends Transformation {
2    // coercions:
3    def toRepr(pair: (Int, Int)): @high Long = ...
4    def toHigh(l: @high Long): (Int, Int) = ...
5
6    // constructor:
7    def ctor_Tuple2(_1: Int, _2: Int): @high Long = ...
8
9    // interface:
10   def implicit_GaussianInt_%(n1: @high Long, n2: @high Long): @high Long = ...
11   def implicit_GaussianInt_norm(n: @high Long): Int = ...
12 }
```

heap-allocated pair representation implies too much overhead. Using ADRT's scopes, the developer can statically mark the `gcd` method as a method that prefers the `Long` representation. This is shown on lines 7–12 in listing 9.1. A scope takes a transformation description object as an argument (line 7) and defines the scope, here lines 8–11, in which the transformation is to be applied.

The following box summarises LDL and ADRT in terms the taxonomy presented chapter 3.

LDL and ADRT in Seven Questions and Answers:

Q1: Who is responsible for data representation changes?

In LDL and ADRT it is the developer who is responsible for defining the representations between which a change is possible and how the representation change is to be carried out. The environment, on the other hand, is responsible for initiating the representation changes when needed and/or possible.

Q2: How is a data representation change realised?
The developer has to define a *transformation object* which specifies the back and forth transitions between two representations. The transformation object ensures an *orderly* representation change, *external* to both representations.

Q3: When is a data representation change executed?
The environment is responsible for initiating the representation changes when needed and/or possible. The *lexical places* where an actual representation change has to occur are determined by the compiler, i. e., *statically* and *external* to the data structure.

Q4: Which data representation changes are possible?
A *transformation object* specifies the back and forth transitions between two representations, i. e., transition graphs in LDL and ADRT are always isomorphic with K_2.

Q5: How long does a data representation change take?
Representation changes in LDL and ADRT are always instant.

Q6: What is changed after a data representation change?
Representation changes in LDL and ADRT *preserve identity*, but change *state*. Arguably also *behaviour* is changed, because a representation change, changes the way in which the object is accessed. The observable behaviour, however, is supposed to remain fixed.

Q7: Why are data representation changes introduced?
In LDL and ADRT, representation changes are introduced to reduce the overhead of heap-allocated objects. The reason is clearly to improve performance.

9.1.2 Object Replacement (in Smalltalk)

Smalltalk is a dynamically-typed class-based object-oriented language. While Smalltalk was not designed to support representation changes as such, it does provide a powerful primitive method, `become`, that can be used to implement representation changes. In Smalltalk, `become:` is a message that is understood by any object `a` which expects a single argument, `b` and which effectively replaces the object denoted by `a` by the object `b`.

> *Of course, you shouldn't use* `become:` *casually. It comes at a cost, which may be prohibitive in many implementations. In early Smalltalks,* `become:` *was cheap, because all objects were referenced indirectly by means of an object table. In the absence of an object table,* `become:` *traverses the heap in a manner similar to a garbage collector. The more memory you have, the more expensive* `become:` *becomes.*
>
> ...
>
> *You can even conceive of type-safe* `become:` *. Two way* `become:` *is only type safe if the type of A is identical to that of B, but one way* `become:` *only requires that the new object be a subtype of the old one.*

Listing 9.3: The representation of `m` changes from `RowMajorMatrix` to `ColMajorMatrix` by a call to `become:` in Smalltalk.

```
1  m := RowMajorMatrix new.
2  m become: (ColMajorMatrix new: m).
```

> — Gilad Bracha
> http://gbracha.blogspot.be/2009/07/miracle-of-become.html

We can look at `become` as a primitive to implement genuine data representation changes. But, as illustrated by the quote of Gilad Bracha, `become:` in contemporary Smalltalk implementations is both expensive and not guaranteed to be type safe. More recently, 2015, Miranda and Béra [58] propose the use of a read barrier to support a more efficient `become`. To implement type safe representation changes in Smalltalk, the developer must guarantee that the old object and the new object have the exact same type.

Listing 9.3 shows a small Smalltalk program where the representation of `m` changes from `RowMajorMatrix` to `ColMajorMatrix` by a call to `become:`. If both `RowMajorMatrix` and `ColMajorMatrix` implement the exact same set of messages and instance variables, then `RowMajorMatrix` and `ColMajorMatrix` can be considered the yield instance of the same type and then the representation change on line 2 is type safe.

The following box summarises Smalltalk in terms the taxonomy presented chapter 3.

Smalltalk in Seven Questions and Answers:

Q1: Who is responsible for data representation changes?
In Smalltalk the developer is responsible for all potential representation changes.

Q2: How is a data representation change realised?
Smalltalk's `become` allows one object to replace another, which effectively implements a representation change. Such a representation change, however, is *not orderly*.

Q3: When is a data representation change executed?
Everywhere a developer introduces a `become`, a representation change is executed. Since `become` can only be inserted in lexical places, Smalltalk supports only *static* representation change incentive code.

Q4: Which data representation changes are possible?
In Smalltalk any object can become any object. Hence, the transition graph in Smalltalk is isomorphic with K_n (with n the number of classes in the system).

Q5: How long does a data representation change take?
`become` changes the representation of an object *instantaneously*.

Q6: What is changed after a data representation change?
`become` replaces one object by another: the *identity is preserved*, anything else can have changed, i. e., *both state and behaviour*.

Q7: Why are data representation changes introduced?
In Smalltalk the developer is responsible for everything. Hence, the developer can use `become`, which is general, for any reason.

9.1.3 Object Evolution

Changing the type of an object at runtime is called *dynamic object reclassification* (cf. chapter 4). Cohen and Gil [13] make the case for a restricted form of dynamic object reclassification called *Object Evolution*. Whereas dynamic object reclassification allows an object of any type to change to any other type, Object Evolution only allows an object to gain *properties*. (i. e., methods and/or fields) but never to lose any property. This restricted form of reclassification is called *monotonic reclassification* and Cohen and Gil [13] proposed three concrete implementation techniques to realise monotonic reclassification, i. e., M-Evolution, S-Evolution, and I-Evolution.

Explicit support for orderly representation changes is the key resemblance between Object Evolution and JITds. Object Evolution introduces *evolvers*, which describe how new fields and new method have to be initialised upon a representation change. Just like to JITds's *transition functions*, *evolvers* look a lot like parameterless constructors, responsible not for the orderly construction of an object but responsible for an orderly representation change of an object. A difference between evolvers and *transition functions* is that evolvers describe how to initialise *new* fields, whereas transition functions describe the full transition between two sets of fields, cf. the pseudo variables `source` and `target`.

The key difference between Object Evolution and JITds is the kind of dynamic object reclassification that is allowed. Object Evolution is restricted to monotonic reclassification. Not considering fields, this means an object can only acquire new methods and never lose the ability to invoke a method. In a strict perspective, JITds provides an even more restricted kind of reclassification, called *homomorphic reclassification*, where the set of messages understood is invariant.

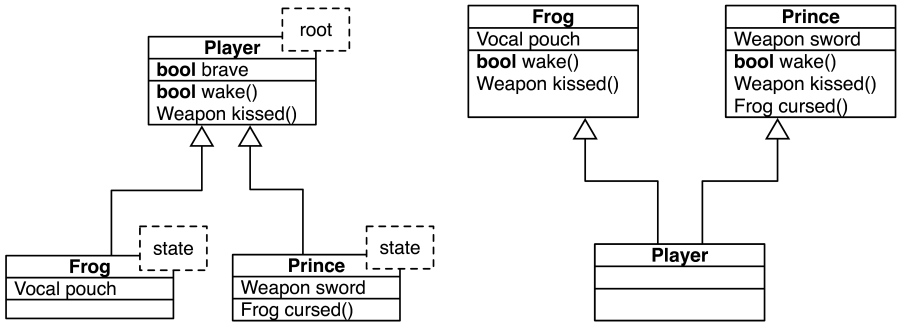
The following box summarises Object Evolution in terms the taxonomy presented chapter 3.

Object Evolution in Seven Questions and Answers:

Q1: Who is responsible for data representation changes?
In Object Evolution the developer is responsible for all potential representation changes.

Q2: How is a data representation change realised?
A transformation between two representations is defined by an *evolver*. The evolver is a member of a class, hence, transformation logic in Object Evolution is *internal* and *orderly*.

Q3: When is a data representation change executed?
Everywhere a developer introduces the →-operator, a representation change is exe-



(a) In Fickle$_{II}$ state classes are subclasses of a root class.

(b) In JITds any set of classes can be combined using a form of multiple-inheritance.

Figure 9.1: Fickle$_{II}$ and JITds differ in how reclassifiable classes are combined.

cuted. Since, → can only be inserted in lexical places, Object Evolution supports only *static* representation change incentive code.

Q4: Which data representation changes are possible?
Representation changes in Object Evolution are unidirectional (i. e., one can not transition back), and only down the ladder of class hierarchy (i. e., an object can evolve into a subtype but not into a supertype).

Q5: How long does a data representation change take?
The →-operator changes the representation of an object *instantaneously*.

Q6: What is changed after a data representation change?
The →-operator changes *both state and behaviour*, while the *identity is preserved*.

Q7: Why are data representation changes introduced?
Object Evolution is intended to be used in applications where dynamic object reclassification is wanted. Hence, Object Evolution is *generally* applicable.

9.1.4 Dynamic Reclassification (in Fickle$_{II}$)

Fickle$_{II}$ is, just like JITds, an imperative, class-based, statically-typed object-oriented language [27]. Fickle$_{II}$ introduces two new kinds of classes, i. e., *root classes* and *state classes*. All subclasses of root classes are state classes, and conversely, state classes have a root class up their hierarchy chain (see figure 9.1a). Fickle$_{II}$ allows dynamic reclassification of objects with the root class as type to any state class with the same root class. Reclassification is obtained though a new language construct, the reclassification primitive (⇓). Fickle$_{II}$ is proven to be type safe and sound with respect to the operational semantics.

The dynamic reclassification proposed by Fickle$_{II}$ is very related to our approach: objects of certain types can transition between different classes. Both approaches in-

troduce a reclassification primitive, i. e., $mate \Downarrow Frog$ and `mate to Frog` reclassify an object *mate* to the representation *Frog* in Fickle and JITds, respectively. We identify the following differences between Fickle$_{II}$ and JITds: 1. Fickle$_{II}$'s state classes (cf. our representation classes) have to be explicitly annotated as such and they have to be — direct or indirect — subclasses of a root class.[1] A just-in-time class, on the other hand, is a subtype of a set of representation classes. 2. To ensure soundness, Fickle$_{II}$'s type system does not allow field members to have the type of a state class. Because representation classes are regular classes in our approach, this limitation does not apply to our language (figure 9.1). 3. In Fickle$_{II}$, when an object reclassifies from class `A` to class `B`, all fields present in both classes are kept as is, fields not present in `B` are dropped, and fields only present in `B` are added and initialised with their default value. There is no support for transition functions (or a similar construct) to express orderly transitions between two states. 4. All reclassifications in a Fickle$_{II}$ program are (syntactically) explicit, i. e., by use of the $\Downarrow$ primitive. A just-in-time object can change its representation both explicitly (using the `to` statement), but also implicitly either because of a specialised swap, or because an internal or external swap rule triggered a representation change.

The following box summarises Fickle in terms the taxonomy presented chapter 3.

Fickle in Seven Questions and Answers:

Q1: Who is responsible for data representation changes?
In Fickle the developer is responsible for all potential representation changes.

Q2: How is a data representation change realised?
Drossopoulou et al. [27] do not specify how to express transformation logic.

Q3: When is a data representation change executed?
Everywhere a developer introduces the $\Downarrow$-operator, a representation change is executed. Since, $\Downarrow$ can only be inserted in lexical places, Fickle supports only *static* representation change incentive code.

Q4: Which data representation changes are possible?
Representation changes in Fickle are possible between all *state classes* that share a common *root class*.

Q5: How long does a data representation change take?
The $\Downarrow$-operator changes the representation of an object *instantaneously*.

Q6: What is changed after a data representation change?
The $\Downarrow$-operator changes *both state and behaviour*, while the *identity is preserved*.

Q7: Why are data representation changes introduced?
Fickle is a language that natively supports dynamic object reclassification. Hence, Fickle is *generally* applicable if one wants to introduce representation changes.

[1] In Fickle$_3$ [17], the successor of Fickle$_{II}$, the explicit annotation of `state` and `root` classes is not longer needed, because Fickle$_3$ has a more refined type and effect system than Fickle$_{II}$. However, the language design aspects are better described for Fickle$_{II}$ than for Fickle$_3$. Hence, we base our comparison on [27] instead of [17].

9.1.5 Typestate-Oriented Programming (in Plaid)

Typestate-oriented programming is an extension of the object-oriented programming paradigm where objects can be modelled in terms of *changing state* [3]. Plaid is a typestate-oriented programming language providing native support for encoding state machines directly in their programs [78]. Such state machines are reifications of assumed *protocols* that can often be found in software libraries or frameworks.

A simple example is an object representing a file which can be opened or closed (cf. section 8.3.3). Here, the *protocol* is that only open files can be read from, and only closed files can be opened. This example is implemented in listings 9.4 to 9.6 in Java, Plaid, and JITds respectively.

Java The protocol of how a `File` is to be used is cryptically added as a comment on line 3 in listing 9.4 but can not be enforced. For instance, when the file is still closed, reading from it will result in a `NullPointerException` (cf. unitised `filePtr`, on line 5 in listing 9.4) unless the programmer explicitly adds checks.

Plaid The protocol, and the corresponding state machine, of how a `File` is to be used that can be expressed using states in Plaid (see listing 9.5).[2] Changing the state of an object has to be programmed explicitly using the state change operator `<-`, for which only `this` can be the receiver. As a result, explicit checks to ensure correct the usage of the file protocol are not needed in Plaid. Hence, calling `readClosedFile` (see listing 9.7) with an open file causes a runtime exception.

JITds Finally, in JITds a `File` is a combination of `ClosedFile` and `OpenFile`. The methods `open` and `close` can be expressed as *named transition functions*: the logic to control representation changes is centralised in the just-in-time class `File`. When `readClosedFile` and `readFileAndClose` (listings 9.7 and 9.8, respectively) are executed in JITds, no runtime exception occurs. Calling `open` (cf. `readClosedFile`) is idempotent for open files; and a closed file will automagically be opened in `readFileAndClose`, because a call to `read` implicitly invokes a *specialised transition* from `ClosedFile` to `OpenFile`.

Both Plaid and JITds are class-based object-oriented languages with native support for representation changes, including an operator to instruct such a representation change.

A first difference is that Plaid, as opposed to JITds, is dynamically (or gradually) typed. A more important difference is Plaid's focus on *functional* and *behavioural* representation changes, i. e., object protocols. A representation change has its semantic origin in the application's state-based model, i. e., a file transitioning from open to closed implies a different state and therefore different behaviour is expected. This explains certain design choices such as the flow-sensitive, permission-based type system to track the current representation of the objects in the flow of a program. Such a type system makes less sense in the context of JITds, whose focus is on *non-functional* representation changes.

[2] Plaid, as described by Sunshine et al. [78], is a dynamically-typed language. For the purpose of comparison with Java and JITds we adapted the example code to match Java, i. e., we added static types.

Listing 9.4: A file in Java.

```
1  class File {
2    String path;
3
4    // null when closed.
5    OSFile filePtr = null;
6
7    String read() { ... }
8    void close() {
9      filePtr = null;
10   }
11   void open() {
12     filePtr = fopen( path );
13   }
14 }
```

Listing 9.5: A file in Plaid.

```
1  state File {
2    String path;
3  }
4
5  state OpenFile case of File {
6    OSFile filePtr;
7
8    String read() { ... }
9
10   void close() {
11     this <- ClosedFile;
12   }
13 }
14
15 state ClosedFile case of File {
16   void open() {
17     this <- OpenFile;
18   }
19 }
```

Listing 9.6: A file in JITds.

```
1  class SimpleFile { String path; }
2
3  class OpenFile extends SimpleFile {
4    OSFile filePtr;
5    String read() { ... }
6  }
7
8  class ClosedFile extends SimpleFile { }
9
10 class File combines ClosedFile, OpenFile {
11   ClosedFile to OpenFile as open {
12     target.path    = source.path;
13     target.filePtr = fopen( source.path );
14   }
15
16   OpenFile to ClosedFile as close {
17     target.path    = source.path;
18   }
19 }
```

Figure 9.2: Implementing a file and its usage protocol in Java, Plaid, and JITds.

Listing 9.7: Procedure that opens, reads, and closes a file.

```
1  String readClosedFile(File f) {
2    f.open();
3    String str = f.read();
4    f.close();
5    return str;
6  }
```

Listing 9.8: Procedure that tries to read the contents of a file and then closes it.

```
1  String readFileAndClose(File f) {
2
3    String str = f.read();
4    f.close();
5    return str;
6  }
```

A final difference is that Plaid does not provide dedicated language construct to initiate representation changes (cf. swap rules).

The following box summarises Plaid in terms the taxonomy presented chapter 3.

Plaid in Seven Questions and Answers:

Q1: Who is responsible for data representation changes?
In Plaid the developer is responsible for all potential representation changes.

Q2: How is a data representation change realised?
In plaid it is possible to provide a specialised implementation of the <−-operator, which is the operator to introduce a representation change. Hence, it is possible to define *orderly* representation changes in Plaid.

Q3: When is a data representation change executed?
Everywhere a developer introduces the <−-operator, a representation change is executed. Since, <− can only be inserted in lexical places, Plaid supports only *static* representation change incentive code.

Q4: Which data representation changes are possible?
Representation changes in Plaid are possible between all *state classes* that are *cases of* the same *state class*.

Q5: How long does a data representation change take?
The <−-operator changes the representation of an object *instantaneously*.

Q6: What is changed after a data representation change?
The <−-operator changes *both state and behaviour*, while the *identity is preserved*.

Q7: Why are data representation changes introduced?
Plaid is a language that natively supports dynamic object reclassification. Hence, Plaid is *generally* applicable if one wants to introduce representation changes.

9.1.6 Gilgul

Costanza [15] developed the programming language Gilgul to tackle the consistency problems that can arise when a programmer manually deals with dynamic object replacement: 1. all references to the object must be known in order to perform the re-

placement and 2. even if all references are known, updating them one by one leaves the heap in an inconsistent state during the updates.

Identity through indirection solves these issues: a *reference* points to an entry which holds a *referent* which points to an *object*. This extra level of indirection allows Gilgul to introduce the *referent assignment* operator #=. At the same time, this extra level of indirection adds a new dimension to the type of objects and these so called referents. For instance, referents can be *fixed*, which means that they may be copied and used but not replaced; alternatively, referents may be *bound*, which means that they cannot be replaced or copied.[3]

Just like JITds, Gilgul is based on Java. In the development of Gilgul, special care is taken to be backwards compatible with the type system of the base language. This implies that an object can only be replaced by another object, if that object implements at least the same set of methods and holds at least the same set of fields (cf. additive replacement). In class-based programming languages, additive replacement is only possible between instances of the same class or subclasses. Ensuring type safety in the context of subtractive replacement, i. e., replacement of an objects by an object that is not not an instance of a subclass, is more complex. To this end Gilgul extends Java, and its type system, with the concept of *implementation-only classes*: classes which provide a concrete implementation, but that can not be used as a type. Consequently, instances of these classes can always be replaced by instances of any class of the given hierarchy. Hence, just like Plaid and Fickle$_{II}$ (and to a lesser extend Object Evolution) the type of the representation changeable object is a super-type of all the possible representations. In JITds, this is the other way around: the type of a just-in-type object is a subtype of any of its representations.

The following box summarises Gilgul in terms the taxonomy presented chapter 3.

Gilgul in Seven Questions and Answers:

Q1: Who is responsible for data representation changes?
In Gilgul the developer is responsible for all potential representation changes.

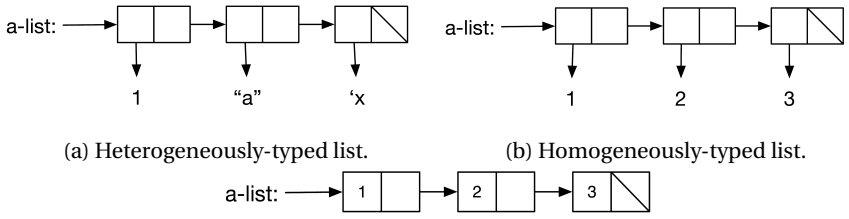
Q2: How is a data representation change realised?
Gilgul's #=-operator allows one object to replace another, which effectively implements a representation change. Such a representation change, however, is *not orderly*.

Q3: When is a data representation change executed?
Everywhere a developer introduces the #=-operator, a representation change is executed. Since, #= can only be inserted in lexical places, Gilgul supports only *static* representation change incentive code.

Q4: Which data representation changes are possible?
Representation changes from supertype to subtype are always possible in Gilgul. Other representation changes are only possible between all *implementation-only classes* that share a common ancestor.

Q5: How long does a data representation change take?
The #=-operator changes the representation of an object *instantaneously*.

[3] Bound referents is the classic approach of dealing with objects.



(a) Heterogeneously-typed list.

(b) Homogeneously-typed list.

(c) Homogeneously-typed list with inlining storage strategy.

Figure 9.3: A three element list stored in three different ways.

Q6: What is changed after a data representation change?
The #=-operator changes *both state and behaviour*, while the *identity is preserved*.

Q7: Why are data representation changes introduced?
Gilgul is a language that natively supports dynamic object reclassification. Hence, Gilgul is *generally* applicable if one wants to introduce representation changes.

9.1.7 Summary

Besides JITds, LDL and ADRT, there exist a fair amount of other programming languages with support for online data representation changes, i. e., SmallTalk, Fickle$_{II}$, Plaid, and Gilgul. JITds, LDL, and ADRT, focus on improving the performance of a program (i. e., a non-functional feature). These other languages, however, introduce representation changes to implement functional features (cf. typestate-oriented programming).

9.2 Frameworks and Environments for Changing Collections

Collections are arguably the most commonly used category of data structures. Virtually all applications use some sort of collection to manipulate data. Replacing the implementation of a collection by a more efficient implementation can increase the performance of an application significantly [10, 47, 69, 90]. It is thus not surprising that a lot of research projects aim to ease, or completely take over, the job of the software developer in choosing the best representation for his collections. In this section we discuss those efforts that we consider to be the closed related to JITds. Some of these efforts, however, only focus on representation *selection* as opposed to representation changes. These efforts are not discussed in function of the taxonomy of representation changes.

9.2.1 Storage Strategies

Collections in dynamically-typed languages can hold elements of effectively any type, i. e., collections are heterogeneously-typed (see figure 9.3a). However, in practice many

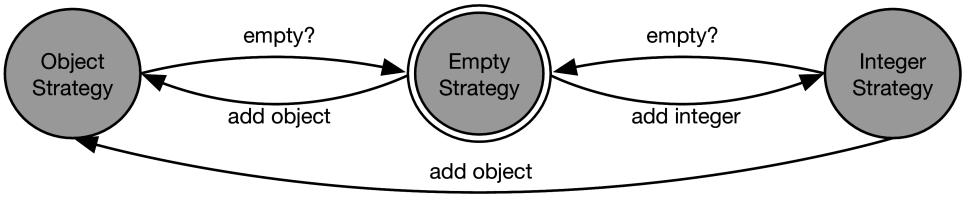


Figure 9.4: Finite state machine where the states reveal which storage strategy is used upon insertion or deletion of en element. (Image based on [10])

collections contain elements of only one type (e. g., a list whose elements are all numbers, see figure 9.3b). Such collections are homogeneously typed. Storage Strategies [10] is a technique for optimising the representation of homogeneously typed collections in VMs for dynamically-typed languages. The optimisation relies on avoiding unnecessary boxing of native values (e. g., as in figure 9.3c). Concretely, Storage Strategies have been implemented as an optimisation for the PyPy interpreter.

Storage Strategies are best explained using a finite state machine (see figure 9.4). Depending on the state, a different *storage strategy* is used. Any collection is initially empty, and therefore start by using the *empty strategy*. Upon the insertion of the first element, the collection transitions to a new state and starts using a different storage strategy, which depends on the dynamic type of the new element. When the element is an integer, the *integer strategy* is chosen. While a collection is using the *integer strategy*, elements are added to the collection without being boxed. The collection keeps using this strategy as long as all elements added to the collection are integers. As soon as a non-integer is added to the collection (see figure 9.4, long arrow from right to left), the collection starts using the *object strategy* which stores all elements as references. In the object strategy, native values have to boxed Hence, when transitioning from the integer to the object strategy, it is necessary to first box all integers already present in the collection. When the last element of a collection is removed, regardless of the current strategy, it transitions back to using the *empty strategy*. The *integer strategy* is an example of an *data specialisation* (cf. section 3.2.7.2). In the implementation of [10] there is an optimised strategy for each native data type (and each collection type, e. g., *string dict strategy* for a $dict$ containing only $strings$). The actual finite state machine is thus larger than the one shown in figure 9.4.

Storage Strategies has many similarities with our work on JITds. First, changing the storage strategy of a collection changes the behaviour of that collection, much like changing the representation of a just-in-time object causes different method implementations to be invoked. Second, changing the storage strategy of a collection changes the (type of) data structure that is used to back the collection (i. e., list of integers versus list of references). Similarly, changing the representation of a just-in-time object can change the type (and number) of fields. In JITds, it is the responsibility of the developer to keep the object consistent, this is realised by implementing a transition function. In the implementation of [10], a similar piece of code is found to ensure the transition of a collection with a specialised strategy (unboxed) to a collection with the *object strategy*

(i. e., $switch\_to\_object\_strategies$).

Unlike JITds, Storage Strategies was developed with a single environment and scenario in mind. Storage Strategies has to be discussed in the context of dynamically typed languages where a collection can store values of any type. The concrete scenario in which Storage Strategies prove their worth, is to optimise the storage scheme of homogeneously typed collections. This scenario translates to a rather regular (and fixed) *transition graph*: the *empty strategy* is the start state — because all collections are initially empty — with transitions defined from the *empty strategy* to any other strategy (bi-directional) and from any strategy to the *object strategy* (uni-directional). In a sense, with JITds we generalise the techniques introduced by Bolz et al. [10]. One of our implementations (i. e., JITds-Java) also uses a variant of the strategy pattern (i. e., bridge pattern) to implement dynamic reclassification.

The following box summarises Storage Strategies in terms the taxonomy presented chapter 3.

Storage Strategies in Seven Questions and Answers:

Q1: Who is responsible for data representation changes?
Storage Strategies is an implementation optimisation of the PyPy interpreter. All representation changes are defined and initiated by the *environment*. This implies that the developer is not necessarily aware of representation changes.

Q2: How is a data representation change realised?
The implementation of the PyPy interpreter realises the representation changes, i. e., storage strategies.

Q3: When is a data representation change executed?
The implementation of the PyPy interpreter decides when the representation changes, i. e., storage strategies.

Q4: Which data representation changes are possible?
The implementation of the PyPy interpreter foresees a set of possible representation changes.

Q5: How long does a data representation change take?
A representation change in Storage Strategies is *instantaneous*.

Q6: What is changed after a data representation change?
Both *identity* and *behaviour* are preserved in Storage Strategies, only the *state* is changed.

Q7: Why are data representation changes introduced?
Storage Strategies focusses on reducing the memory overhead of homogeneous collections in dynamic languages.

9.2.2 Brainy

Brainy is a program analysis tool that determines the best data structure for a given program, a given input, and a given architecture [47]. Brainy makes its decision about which data representation is the best for a given data interface based on a *machine-learned*

model. Hence, the biggest contribution of Jung et al. [47] is the techniques presented to construct such models. In Brainy, this is realised in two phases. In a first phase, Brainy randomly *generates programs* and *measures the execution times* for each viable alternative data representation. Phase one produces for each data representation a set of programs for which that data representation is the best option. Then, in phase two, a *training set* is created for each data representation. This training set contains all the *software and hardware features* that can be collected during the executions of the programs selected in phase one. Then, techniques from machine learning are applied to these training sets, which result in a model that explains which data representation works best with a given program, input, and hardware.

While Brainy is a fully automated tool, the models on which the decision making is based, are created a-priori. Jung et al. [47] focus their evaluation of Brainy on C/C++ and STL in the context of `vector`, `list`, `set`, and `map`. However, the techniques presented in [47] are said to be flexible enough to be generally applicable. In order to construct the models for e. g., `vector`, Jung et al. [47] had to explicitly identify `list`, `deque`, `set` (map), `avl_set` (avl_map), and `hash_set` (hash_map) as viable data representations for the `vector` data structure. Thus, the developer that wishes to use Brainy in the context of his own data structures has to create his own models that reason about the intended data structures and has to identify the set of data representations that are viable substitutes. This resembles JITds in the sense that the developer needs to explicitly list the set of representations of a just-in-time class.

The motivation for the work on the Brainy project is similar to what we present in the first part of our motivation chapter (see section 2.1). Jung et al. [47] present a convincing argument that data representation selection is important and that data representation selection is hard to get right. For instance, they refer to the work on Perflint [54], which shows that even expert developers make suboptimal decisions when it comes to data representation selection. Jung et al. [47] partially attribute this bad practice to the limitations of asymptotic complexity measures. We support this claim in section 2.1. Further, Jung et al. [47] identify *branch misprediction* as one of the hardware features that plays an important role in the performance of programs. Thus, it is important to consider the effects of branch prediction when deciding upon which data structure (and algorithm) to use in a program. Compared to two very related projects (Chameleon[69] and CoCo[90] which we discuss next), Brainy is the only project that explicitly takes the underlying hardware into account.

Brainy in Seven Questions and Answers:
Brainy is a tool (*environment*) that performs *data structure selection*. Brainy decides which representation is the best based on a *machine-learned model*. Because Brainy only supports data structure *selection* (as opposed to changes), it is not classifiable according to the seven dimensions of representation *changes*.

Listing 9.9: Rule language to guide collection selection (adapted from [69]).

```
rule       : from cond to
from       : Collection | ArrayList | LinkedList | ...
to         : ArrayList | LinkedList | ...
cond       : comparison | cond && cond | ...
comparison : expr < const | expr == const | ...
expr       : opCount | heapdata | expr + expr | expr / expr | ...
opCount    : #get(int) | #set(int,int) | ...
```

9.2.3 Chameleon

Many programs written in high-level languages make extensive use of collections. Chameleon is a tool that helps developers to pick the best implementation from a set of available collections [69]. First, Chameleon collects detailed information on the run-time usage of collections and report these back to the developer. Then, this information is fed into a rule engine which suggests the most appropriate choice of data structure. All this is done on a *per-allocation context* basis, i. e., the combination of the lexical place in the source code where a collection is allocated (`new Object()`) as well as the call stack at the moment of allocation.

Chameleon uses a rule engine to make its decisions. A subset of the language to express these rules is shown in listing 9.9. First, note the resemblance between internal swap rules and these rules: there is a "from" representation and a "to" representation, as well as a "condition that must hold" (cfr. the body of an internal swap rule). Second, note the resemblance between invocation counters and `opCount`: it is a counter based on the name and usage of the operation.

Further comparison of Chameleon with JITds shows fewer resemblances. The major difference is that Chameleon targets data structure *selection* and does not consider online representation changes. Shacham et al. [69] did some preliminary effort to fully automate their approach, i. e., to run Chameleon during the execution of a program. However, even with this online approach the selected implementation remains fixed for an already allocated object for the duration of its lifetime, i. e., representation selection instead of representation change. While the techniques explored by Shacham et al. [69] are not necessarily limited to collections, the whole approach of Chameleon is highly biased towards this frequently-used class of data structures. In Chameleon, *rules* have access to more information than swap rules in JITds, e. g., `heapdata` (see `expr` in listing 9.9) determines the amount of used memory versus the amount of needed memory of a collection object. This is information collected at run-time (during each garbage collection) and shows that Chameleon is tightly coupled with its host-language's execution environment, here an adapted version of IBM's J9 production JVM. Both of our implementations of JITds show much less coupled to a specific execution environment (i. e., we can target the Java runtime or our custom-built C runtime).

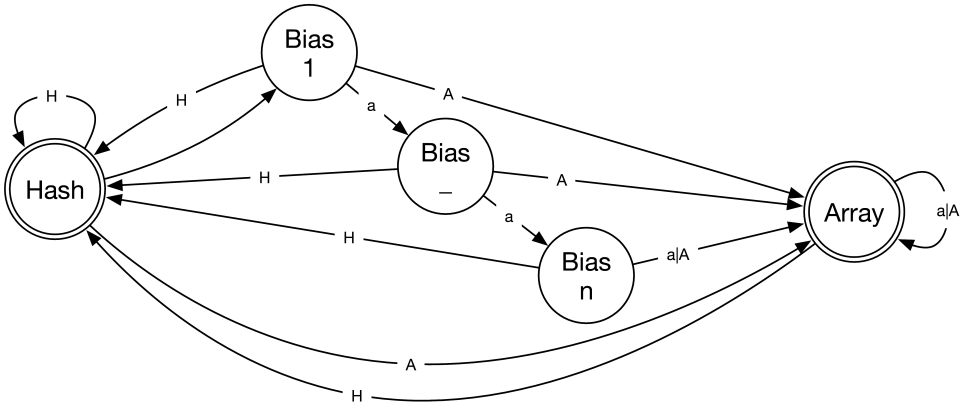


Figure 9.5: Finite state machine where the states encode the sequence of operations. (Image based on [61])

Chameleon in Seven Questions and Answers:
Chameleon is a tool (*environment*) that helps *developers* with *data structure selection*. Chameleon suggests a representation based on a rule engine and detailed information collected during an execution of the program. Because Chameleon only supports data structure *selection* (as opposed to changes), it is not classifiable according to the seven dimensions of representation *changes*.

9.2.4 Dynamically Transforming Data Structures

Österlund and Löwe [61] introduce *transformation data structures*. A transformation data structure is similar to a just-in-time data structure in the sense that both constructs aim to combine multiple representations into monolithic entity of which the representation can change during the execution of a program. Moreover, also the reason for changing the representation is the same, i. e., to adapt to a changing context in which the data structure is used in order to improve the efficiency of a program.

The implementation of transformation data structures is based on forwarding (cf. JITds-Java) and relies on the implementation of a `changeTo` and a `constructThis` method for each representation. These methods are to be provided by the developer of the transformation data structure, who is expected to be a library designer. This approach of defining transformation logic is an extreme case of *general transition functions* in JITds. On the one hand, this avoids the problem of an explosion in the number of transition functions, i. e., n instead of $n^2(-n)$, on the other hand, this approach requires that all representations can be defined in terms of read and update operations on the old representation (cf. section 4.6).

Österlund and Löwe [61] identify that a finite state machine can be used to encode sequences of invoked operations (i. e., data interface usage pattern, section 3.2.7.2), and

that depending on the data interface usage pattern using a different representation can yield a more efficient execution. Figure 9.5 show such a finite state machine that describes when to change the representation of a list between an array-based implementation and a hash-based implementation. The labels on the edges represent *operations*: `H` is an operation where the hash-based implementation is an order of magnitude more efficient than the array-based implementation; `A` is an operation where the array-based implementation is an order of magnitude more efficient than the hash-based implementation; and `a` is an operation where the array-based implementation is significantly more efficient than the hash-based implementation (but not an order of magnitude). When an operation is performed, the appropriate edge is taken, when a state is reached with double contours, then a representation change is performed. The core contribution in [61], is the process of creating such a finite state machine. This process uses offline learning based on profiled sample executions of example programs.

Österlund and Löwe [61] identify a set of pitfalls when using transformation data structures: e. g., issues that arise from multi-threading, issues that arise when using the iterator pattern, and issues that arise when the internal representation is used as monitor object. These issues are all instances of what we called the "the problem of escaping pointers" in chapter 7, which we identified to be a direct result of using the strategy pattern.

The following box summarises Dynamically Transforming Data Structures in terms the taxonomy presented chapter 3.

Dynamically Transforming Data Structures in Seven Questions and Answers:

Q1: Who is responsible for data representation changes?
To develop a *transformation data structure*, the *developer* needs to supply both *transformation logic*. The core contribution of Österlund and Löwe [61] is the model to train *representation change incentive*. The targeted audience for building transformation data structures are library designers.

Q2: How is a data representation change realised?
The methods `changeTo` and `constructThis` have to implemented for all possible representation of the transformation data structure.

Q3: When is a data representation change executed?
Österlund and Löwe [61] describe a process of how to build a finite state machine that records *operation sequences*. The resulting finite state machine then suggests when a representation change is advisable.

Q4: Which data representation changes are possible?
In a *transformation data structure* all alternative representations are equivalent, i. e., they share a set of operations (i. e., data interface) and a set of algorithms (i. e., a sort of abstract computation). The developer also defines all possible representation changes by implementing a lot of `changeTo` and `constructThis` methods. Hence, the transition graph of a transformation data structure is complete.

Q5: How long does a data representation change take?
To change the representation of a transformation data structure changes the appropriate `changeTo` and `constructThis` are invoked. When `constructThis` fin-

ished, the representation is complete. The user of a transformation data structure can not take control `changeTo` and `constructThis` are being executed, hence, transformation data structure have *instant* representation changes.

Q6: What is changed after a data representation change?
The *identity* of a transformation data structure is preserved when its representation changes, The *state* is changed and the and *behaviour* can change as well, i. e., when the new representation relies on different algorithms.

Q7: Why are data representation changes introduced?
[61] introduce transformation data structure to exploit the possibly changing context in which a data structure is used. They bring up number of available cores, size and type of input, contention, and memory usage as examples of *context attributes*.

9.2.5 CoCo

CoCo, *container combinations*, is similar to Chameleon, a technique to improve the performance of Java programs by selecting the best collection for a given application [90]. The biggest difference with Chameleon, however, is that CoCo intervenes at run-time, i. e., CoCo dynamically replaces an old collection implementation with a better one. In this context, CoCo focusses on two concerns: how to ensure a *safe* replacement and how the ensure *low overhead*.

The implementation of a container combination requires both manual and automated efforts. Consider, for instance, `ListCombo` with two possible representations `ArrayList` and `LinkedList`. The classes and interfaces that are needed to implement `ListCombo` are shown in figure 9.6. On the one hand, the CoCo static compiler generates the data interface for that container, called `OptimizableList` in figure 9.6, and a skeleton for the class `ListCombo`, which can change the current active list implementation. The developer,on the other hand, has to create an extension of each of the existing collections's implementations, here `ArrayList` and `LinkedList`, has to provide *abstraction-concretisation* functionality, and the *expert domain* knowledge about when to swap the active representation. The latter entails implementing the `swap` method in the generated skeleton class `ListCombo`.

Just like in JITds, a combo container in CoCo has an *active* representation. However, CoCo maintains all *inactive* containers as well (cf. gradual representation change). The rationale behind this approach is that "the natural idea of moving all elements from one container to another may result in significant overhead" (paraphrased from Xu [90]). In order to avoid moving elements back and forth, CoCo introduces *abstract elements*, as opposed to the concrete elements commonly present in a container. An abstract element in CoCo is a reference to another container combined with a range in that container. In figure 9.7 two lists $\mathtt{L}_0$ and $\mathtt{L}_1$ are shown that together hold the concrete elements O_0, O_1, O_2, O_3, O_4. $\mathtt{L}_0$ holds the concrete objects O_2 and O_3, while $\mathtt{L}_1$ holds the other concrete objects. Because both lists represent the same data, $\mathtt{L}_0$ holds an abstract version of O_2 and O_3, i. e., *those elements of* $\mathtt{L}_0$, *from position 2 until 3* ($< L_0, 2, 3 >$). Similarly, $\mathtt{L}_1$ holds abstract versions of the other elements, which refer to the concrete elements in

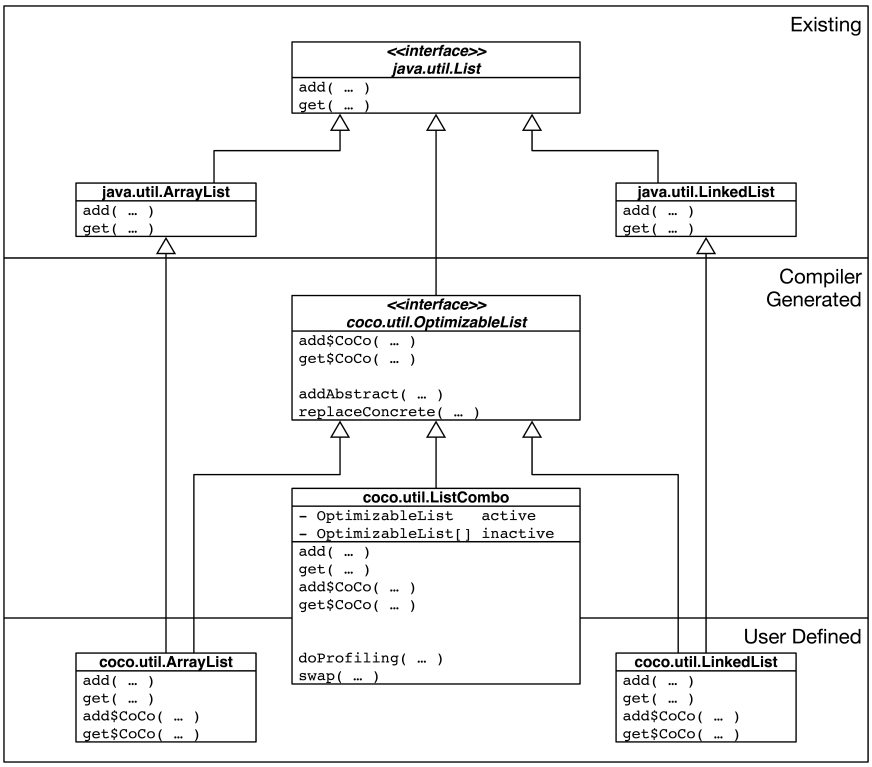


Figure 9.6: Overview of the classes and interfaces needed to create a container combination, here a `ListCombo`.

L_0. The abstract elements in the active container are lazily concretised, whenever they are accessed. For instance, calling L_0`.get(2)`, which should return O_2, replaces the abstract version of O_2 in L_0 by the concrete O_2 from L_1.

Further, CoCo proves that this technique of maintaining abstract and concrete values in multiple containers is sound in the context of a change in container representation. Xu [90] say that a *container replacement* is sound if any element added to a combo container can always be successfully retrieved from that container, even after a container replacement. How this influences operations other then adding and retrieving, e. g., obtaining the size of the container, remains unclear. Hence, the *sound replacement* of collections only applies to adding and retrieving elements.

The biggest differences between JITds and CoCo are that JITds proves soundness of any representation change expressible in JITds and that representation changes in JITds are not limited to containers only. As described in [90], Brainy only supports representation changes between containers that adhere to the same interface. While Xu [90] foresaw the ability to support representation changes between multiple (specialised) data interfaces as future work in 2013, this has not been investigated as such. JITds, on the

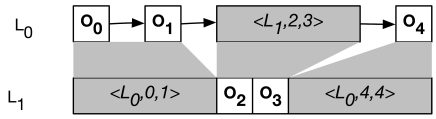


Figure 9.7: Two lists, $\mathtt{L}_0$ and $\mathtt{L}_1$, that contain both abstract elements (italic) and concrete elements (bold). Moreover, the abstract element in the one list are concrete in the other and vice versa.

other hand, supports representation changes between multiple (specialised) data interfaces by means of *specialised swaps*. A final difference is that JITds does not keep abstract objects around, and does not maintain multiple concurrent representations of the same data structure.

The following box summarises CoCo in terms the taxonomy presented chapter 3.

CoCo in Seven Questions and Answers:

Q1: Who is responsible for data representation changes?
To develop a *Combo Container*, the *developer* needs to supply both *transformation logic* as well as *representation change incentive*. The CoCo compiler (*environment*) compiles these into plain Java.

Q2: How is a data representation change realised?
A container is a collection of elements. Instead of moving all elements from one container to another upon a a representation change, CoCo only moves elements to the current container on-demand, i. e., lazily. This is possible because the inactive containers hold *abstract elements* which can be *concretised* on-demand. The developer is responsible for defining *transformation logic* in the form of *abstraction/concretisation*-procedures per collection and per interface method.

Q3: When is a data representation change executed?
The developer of a combo container is responsible for defining *swap conditions*. When a swap condition holds, a representation change is executed. The swap conditions can trigger at virtually any point during the execution of a program. Hence, swap conditions are *internal* and *dynamic* representation change incentives.

Q4: Which data representation changes are possible?
A developer combines a set of collection implementations into one combo container, cf. *the vertices in the transition graph*. These collections have to share an *interface*, cf. *the relation between vertices in the transition graph*. The developer also defines the possible representation changes by means of swap conditions, cf. *the edges in the transition graph*. Hence, the transition graph in CoCo is unique per combo collection and is fully defined by the developer.

Q5: How long does a data representation change take?
When a combo collection changes its representation, the currently active container becomes inactive and a previously inactive container becomes the active one. The inactive containers, however, are also maintained. Abstract elements are only *gradually* moved into the active container, i. e., on-demand.

Q6: What is changed after a data representation change?
Both *identity* and *behaviour* are preserved when a combo container changes its representation, and only the *state* is changed.

Q7: Why are data representation changes introduced?
CoCo focusses on reducing memory bloat and exploiting algorithmic advantages in collections in general, and Java collections in specific. The techniques presented in [90] are only applicable in the context of *collections of elements*.

9.2.6 Summary

Storage Strategies, Brainy, Chameleon, Transforming Data Structures, and CoCo introduce representation selection or representation changes with the goal to reduce memory overhead in collections. The focus in these efforts is on building and developing models that describe when certain representation is to be preferred, e. g., by offline training. In chapter 10, we speculate on how the techniques to build these models can be incorporated with our language design efforts.

9.3 Changing Computations

The focus of this dissertation is on *online representation changes* in the context of object-oriented programming languages. In this section of the related work chapter we discuss two programming language efforts where the focus is on *implicit changes of the computation* of a program during its execution.

9.3.1 PetaBricks

PetaBricks is an implicitly parallel programming language whose compiler is allowed to decide which algorithm, out of a set of equivalent implementations of an algorithm, is to be used at runtime [6]. An example of sorting in PetaBricks is shown in listing 9.10. Each implementation, called a *rule*, describes how to compute a recursive step for a certain region of the input data (*from* argument), optionally with a *where*-clause to denote in which (corner) cases the rule can be applied. In listing 9.10 there are three rules: one for the base case of singleton, one that implements *merge sort*, and one that implements *selection sort*. The combination of rules is called a *transform*, which is PetaBrick's equivalent of a function, but with *algorithmic choice* encoded. It is up to the compiler to find an optimal *sequence* of applicable rules to compute the final result. Such an optimal sequence is dependent on the target architecture and therefore the compilation process of a PetaBrick program includes auto-tuning.

PetaBricks can be seen as the algorithmic counterpart of JITds, i. e., *rules* are the computational equivalent of representation classes and a *transform* is roughly equivalent to our just-in-time classes. Besides the trivial differences, e. g., PetaBricks does not support an equivalent of transition functions as there is no data to be converted. PetaBricks

Listing 9.10: Three implementations of a recursive `sort` algorithm in PetaBricks .

```
1  transform Sort                          // Transform
2  from A[n]
3  to   A[n]
4  {
5    to   ( A[i] a)                        // Rule 1:
6    from ( A[i] a)                        // Already Sorted
7    {
8    }
9
10   to   ( A.region(i,j) a)               // Rule 2:
11   from ( A.region(i,j-i/2) a1,          // Merge Sort
12          A.region(j-i/2,j) a2)
13   {
14     Sort(a1);
15     Sort(a2);
16     Merge(a,a1,a2);
17   }
18
19   to   ( A.region(i,j) a)               // Rule 3:
20   from ( A.cell(i) a,                   // Selection Sort
21          A.region(i+1,j) a1)
22   {
23     swap(a[i], a[findMaxIndex(a)])
24     Sort(a1);
25   }
26 }
```

differs from our approach mainly in its pure implicit nature of providing choice: all responsibility lays with the compiler, potentially guided by hints to the auto-tuner in the form of user-defined *tuneable* parameters. This is possible because the number of rules per transform are finite and , more importantly, because a new algorithmic choice can only be made at computation boundary steps, e. g., (recursive) calls to a transform. This is different than in JITds, where any statement can potentially cause a representation change. Second, even though Ansel [5] also discusses online auto-tuning in his PhD dissertation, the original PetaBricks [6] focusses on offline choices, i. e., at compile or install time.

PetaBricks in Seven Questions and Answers: PetaBricks is a compiler (*environment*) that performs *online computation selection.* PetaBricks decides which algorithm is the best, based on *auto-tuning.* Because PetaBricks only supports *computation selection,* and not representation changes, it is not classifiable according to the seven dimensions of representation changes.

9.3.2 Dimensions of Method Dispatch

In JITds, just as in any other object-oriented language, method dispatch is based on the method name and the type of the receiver. Under the influence of representation changes, calling `x.foo()` is not guaranteed to always invoke the same method (over the lifetime of `x`). This adds another dimension to method lookup.

Other approaches exist where method resolution is dynamically decided. Context-oriented programming (COP), for instance, takes the "current context" (i. e., *any computable property*) into account when performing a method look-up. Hirschfeld et al. [43] identify in [43] that contemporary method lookup mechanisms can be as complex as four dimensional: 1. the method name (procedural languages), 2. the receiver's type (polymorphism), 3. the sender's type (subjective programming, e. g., Us, the subjective version of Self [73]), and 4. the "current context". Looking at JITds from the COP perspective, it is fair to say that the representation of a just-in-time object is the *current context*.

> **Context-oriented programming in Seven Questions and Answers:** Context-oriented programming is a programming paradigm where method resolution is dynamically decided depending on the current context. Context-oriented programming does not support representation changes as such, and hence it is not classifiable according to the seven dimensions of representation changes.

9.4 Summary

In Peta-Pricks and context-oriented programming, a developer can express *when* to use *which* algorithm. This is similar to JITds, where the developer can express *when* to use *which* data representation. Both rely on dedicated language construct to express this kind of variability.

9.5 Conclusion

There exist a fair amount of programming languages with support for online data representation changes (e. g., LDL (and ADRT), SmallTalk, Object Evolution, Fickle$_{II}$, Plaid, and Gilgul). Most of these languages introduce representation changes to be able to express *object evolution* which leads to the programming paradigm called *typestate-oriented programming* (i. e., Object Evolution, Fickle$_{II}$, Plaid, and Gilgul). In these programming languages it is the pure responsibility of the developer to define which representation changes are allowed and when they should occur. Moreover, the representation change incentive code in the language is expressed using an operator, and is thus static in all these languages. What these languages provide is the machinery to effectively change the representation while preserving identity. This is true for all languages discussed in section 9.1, except for LDL and ADRT where the environment is responsible for deciding when to change the representation.

Most frameworks, libraries, and environments that introduce representation changes aim at reducing memory overhead in collections (Storage Strategies, Brainy, Chameleon, CoCo). This can be explained because collections (in the broad sense) are arguably the most commonly-used type of data structure and have hence received a lot of attention by the performance analysis community.

Implicitly changing the computation at runtime is more commonly supported than changing the data representation of objects. Just-in-time compilers, for instance, improve the efficiency of code fragments during the execution of the program. These JIT compilers can be found in many contemporary language implementations or runtimes, e. g., the JVM, CLR. From a programming language design point of view, there are no apparent similarities between these *environments that change the computation at runtime* and JITds. PetaBricks, however, does have a lot of language design similarities with JITds. In both programming languages, the developer specifies a set of equivalent alternatives which are combined into a monolithic entity, a *transform* in PetaBricks and a just-in-time class in JITds. Context-oriented programming, on the other hand, can be seen as an overarching paradigm for JITds. The current representation of a just-in-time object is the context on which method look-up depends.

From the surveyed work, we can conclude that JITds is the first programming language with support for *non-functional* representation changes and where transformation incentive code can be disentangled from the application logic.

Chapter 10

Conclusions and Future Work

After 40 years, a main focus in software engineering still lies in finding the "best" data structure implementation for a data abstraction in a program. A program often consists of multiple algorithms that operate on the same data. In such a program, the "best" data structure is regarded as the data structure that yields the best *overall performance*, where performance can be interpreted as shorter execution time, lower memory consumption, higher throughput, or any other desirable non-functional feature. However, in section 2.2 we showed programs in which relying on a *single fixed data representation* is less efficient than what can be achieved when the *representation of the data is changed* at runtime. Hence, in this dissertation we argue in favour of a programming language that enables developers to design data structures that can change their representation *during* the execution of a program: just-in-time data structures. We developed the JITds programming language to fulfil this purpose.

We described our initial idea to investigate just-in-time representation changes from a language perspective in [21]. We presented the core contributions described in this text, i. e., the taxonomy of programming features that are needed to express just-in-time representation changes and the design of the programming language JITds, in [20]. Finally, in [18] we discuss how we think that the current implementation of JITds can be improved, e. g., with techniques from declarative programming and static and dynamic analysis.

In this last chapter of the dissertation we take a step back and verify whether JITds lives up to the expectations formulated in the introduction chapter (sections 10.1 and 10.2), and we formulate some directions for future work (section 10.3).

10.1 JITds in a Nutshell

JITds is a statically-typed class-based object-oriented programming language with support for functional as well as non-functional data representation changes. The support for non-functional representation changes is a novel contribution of JITds (in comparison with other programming languages).

In JITds there are two kinds of classes. There are *simple classes*, as they are known from other class-based object-oriented programming languages. *Just-in-time classes*, on the other hand, combine multiple simple classes, which we call representations. JITds allows just-in-time objects (i. e., instances of these just-in-time classes) to change between their representations at runtime. *Transition functions*, one of JITds' new programming constructs that can be seen as a generalisation of constructors, ensure an orderly transition between two representations.

Moreover, JITds introduces *swap rules*, which enable developers to separate traditional application logic from representation change incentive code. Swap rules come in three flavours:

- **External swap rules** augment a unit of computation (i. e., at method boundary in JITds) with representation change incentive code that can change the representation of objects used in that computation.
- **Interface swap rules** augment the data interface of a just-in-time object (methods of the object) with representation change incentive code that can change the representation of a just-in-time object.
- **Internal swap rules** augment the state (i. e., any computable property) of a just-in-time object with representation change incentive code that can change the representation of that just-in-time object.

A core subset of JITds was given a formal specification (cf. JIT$\Delta\sigma$) for which we proved type soundness. Because the type system of JITds is proven to be sound, we can guarantee that a well-formed JITds program can never encounter "method not found" or "field not found" exceptions at runtime. We subsequently evaluated how representations changes can reduce execution time (i. e., improve performance) of several programs, and evaluated how swap rules can be used to disentangle general application logic from the crosscutting concern of what representation to use.

10.2 Contributions

1. Our first contribution, as presented in chapter 3, is a **taxonomy** of programming language features that enable just-in-time representation changes. We categorise these programming language features into seven dimensions. Each of these dimensions answers a specific question w.r.t. representation changes:

- **Q1: Who** is responsible for data representation changes?
- **Q2: How** is a data representation change realised?
- **Q3: When** is a data representation change executed?
- **Q4: Which** data representation changes are possible?
- **Q5: How long** does a data representation change take?

- **Q6: What** is altered after a data representation change?
- **Q7: Why** are data representation changes introduced?

2. Our second contribution is the design of the experimental **programming language**, JITds, which supports non-functional representation changes. In chapter 1 we argued that such a language must meet the following three requirements:

- **R1** The language must enable data objects to change their data representation at runtime.
- **R2** The language must provide type safety guarantees in the face of representation changes.
- **R3** The language must allow programmers to develop application logic while being agnostic of representation changes.

Designing a novel programming language that meets these three requirements resulted in the following **four sub-contributions**:

- In chapter 4, we present the *design* of the programming language JITds. In JITds it is possible to combine multiple classes into one just-in-time class to enable a data structure to change its representation at runtime (R1). Concretely, we introduce **transition functions** and **swap statements** as the core language features to support representation changes.
- These features are formalised in an **operational semantics** for JITds in chapter 5 (**R1**).
- Also in chapter 5, we show that a well-formed JITds program never gets stuck, not even in the context of representation changes. Concretely, we proved progress and preservation, and thus **soundness**, of JITds's type system. With a sound type system, we can guarantee that a well-formed JITds program will never encounter a "method not found" or a "field not found" exception at runtime (**R2**).
- In chapter 6 we present the second part of the design of the JITds programming language. We show how to decouple representation change logic from application logic by using swap rules (**R3**). Concretely, we introduce three flavours of **swap rules** as the core language features to express representation change incentive code.

3. The technical contributions of this work are the *two prototype implementations* of the programming language JITds. These implementations were discussed in chapter 7. Concretely, we implemented **JITds-Java** and **JITds-C**. JITds-Java is a JITds-to-Java compiler that effectively encodes just-in-time classes as an variation on the bridge and strategy pattern [34]. The implementation technique used in this compiler is called "implementation by forwarding". While the advantage of JITds-Java is that it has full access to Java as a base language (i. e., all standard Java libraries can be used directly), it does

suffer from two key limitations as well. First, because of Java's type system, it is impossible to provide Liskov's substitution without altering existing code (e. g., classes from Java's standard libraries). Second, JITds-Java suffers from the problem of escaping pointers. This problem can be resolved by investing a lot of engineering effort in the compiler, e. g., in the form of a solid static data-flow and points-to analysis. **JITds-C** is our second prototype implementation of JITds. JITds-C is based on the formal semantics of JITds (see chapter 5) and thus does not suffer from the two limitations of JITds-Java. The downside of JITds-C is that, as opposed to JITds-Java, JITds-C can not piggyback on an existing base language. Hence, the applicability of JITds-C is limited to demonstrating the features of JITds. Again, this limitation can be resolved by investing sufficient engineering effort into the compiler implementation.

Finally, we implemented a set of five synthetic benchmark programs and used those to evaluate JITds (see chapter 8). Concretely, we compare the execution times and the code structure of these programs — written in JITds and compiled with JITds-Java — against the same programs written in plain Java.

10.3 Future Work

From the current state of our research we partition our future work into two categories. We have *software engineering* efforts on the one hand, and *language design and implementation* efforts on the other hand. For both categories we enumerate a non-exhaustive list of ideas for future research.

10.3.1 Software Engineering

JITds is a first attempt to promote the shift from choosing a single data structure to choosing a set of data representations. The focus of this work lies on the language design aspects of this idea, e. g., language constructs and a type system. The *software engineering* aspects of the idea of just-in-time data structures, however, were only touched upon (e. g., in section 4.5 and section 4.6) in this dissertation.

It took three decades after the introduction of object-oriented programming (cf. the programming language Simula in the 1960s [16]) to distill a set design patterns observed from real-world OO programs Gamma et al. [34]. Similarly, we foresee that research on software engineering techniques in the context of just-in-time data structures will require some incubation period until a sufficiently large corpus of real-world JITds programs exists. Such a corpus does currently not exist.

10.3.1.1 Popularising the Idea of Just-in-Time Representation Changes

The absence of a sufficiently large corpus of real-world JITds programs prevents research in software engineering techniques to help JITds programmers. This is mainly because JITds is the first language to promote non-functional just-in-time data representation changes, which is an uncommon and unknown technique. Foremost, just-in-time data representation changes need to be supported by many more programming languages. As a result more programs would use it, and more efficient implementations of these

languages would be developed. Then, the technique would get more traction and could advance the way we think about programming data structures and algorithms to another level. Hence, the technique should be made more commonly known, i. e., both in academia and in industry.

10.3.1.2 Tool to Find Programs that can Benefit from Representation Changes

Besides being used in new programs, existing programs may also benefit from using just-in-time representation changes. The ideal candidate is a program where phases of good resource usage (e. g., efficient CPU usage or low memory footprint) and phases of bad resource usage (e. g., inefficient CPU usage or high memory footprint) occur intermittently. Mitchell and Sevitsky [59] describe a way to detect data structures in a program that consume excessive amounts of memory (memory bloat). Sherwood et al. [70] describe a way to detect phases in a program's execution. A combination of these techniques could be used to improve performance in (real world) programs by identifying the potential need for just-in-time representation changes.

10.3.1.3 Identifying Patterns in Swap Rules

We foresee that when more and more programs will be written that rely on just-in-time data representation changes, patterns in code will start to emerge. We have seen this in the 1960s when Dijkstra promoted the use of high-level control constructs [25] and in the 1990s when the gang of four published their design patterns for object-oriented programming [34].

A first kind of pattern detected in JITds programs is that we often want to count the number of method invocations and change the representation of a just-in-time object when a certain threshold or ratio is reached. This can be implemented by using a combination of interface swap rules and counters. In section 6.2.1, we describe invocation counters, a language construct introduced to facilitate the expression of these pattern more easily. Other patterns might lead to more sophisticated swap rules or other language constructs.

10.3.1.4 Measuring the Cost of Transitions

Throughout the dissertation we state that performance engineering is hard and that the average programmer is no longer capable of managing the full extent of the layers of abstractions he relies on. We propose JITds as a solution to separate the application logic from all logic concerning changing and selecting the representation of the data. However, using JITds can also be more complex (cf. The Performance Pitfall of Unintended Specialised Swaps, see section 4.5.1.2). Tool support that helps programmers in understanding the (potential) cost of specialised swaps and explicit representation changes would be a welcome asset in the toolbox of the software engineer.

10.3.1.5 Resolving the Tension Between Application Developer and the Performance Expert

In this text we deliberately create a tension between what we call the application developer and the performance expert. The individual application developer is solely responsible for implementing the functional requirements of the software product, whereas the performance expert is solely responsible for designing just-in-time data structures, i. e., providing the required transition functions and swap rules.

In practice we foresee two possible scenarios: Either the application developer and the performance expert are two distinct individuals, each operating on their domain (separation of expertise), or alternatively, they are one and the same and he or she works on the distinct concerns (i. e., functional requirements and performance) separately (separation of concern). In either case, programming in this setting is new and unexplored, and we foresee collaboration problems due to lack of experience. Hence, over time, best software engineering practices will have to be developed to smoothen the collaboration between the application developer and the performance expert, being it one, two, or more persons.

10.3.1.6 Incorporating Offline Training in JITds

A significant part of the related work in chapter 9 focusses on changing the data representation of collections. A recurring facet in this related work is the construction of models that describe in which cases a representation change could result in improved performance. These models are created during an offline training phase. The creation of the models, and the (data) characteristics they take into account, are a significant part of the aforementioned research efforts. The limitations of these models is that they are specifically designed in the context of collections. We think that generalising the process of designing and creating these models and incorporating them in the design of JITds, could greatly benefit JITds's efficiency and adoption.

10.3.2 Language Design and Language Implementation

Currently, the language has two prototype implementations. Hence, there is a lot of work that can be done to improve upon these implementations. For instance, by incorporating the work of Miranda and Béra [58] who rely on read barriers to improve the efficiently of `become`-like constructs. Depending on how JITds will be used in the future, the language design will need to evolve along as well.

10.3.2.1 Dynamic Transition Graph

In the current design (and implementation) of JITds, when changing an object's representation, we use the *shortest path of subsequent transition functions*. In chapter 5, we formally specify that the shortest path is the path with the fewest intermediate representations, and the order in which the representations occur in the definition of a just-in-time class is used as a tiebreaker.

The shortest path, however, is not necessarily the most efficient path. A generic transition function might transform an object from `Ri` to `Rj` in one step, but could take twice the time needed to first transition `Ri` to `Rx` and then from `Rx` to `Rj`. A straightforward extension of JITds's current design is to assign a *cost* to each transition function. This change would cause the transition graph to become a *weighted* directed graph, and the transition path could then be defined as the cheapest path.

A possible avenue to extend the design of JITds to support dynamic transition graphs, are the introduction of a programmer defined construct to express the *expected cost* of each transition function, potentially parameterised by the current state of the data structure. Alternatively, the runtime of a future JITds implementation can be made responsible for *measuring the cost* of a transition function, in order to dynamically update the costs in the transition graph.

10.3.2.2 Declarative Swap Rules

Conceptually, the idea of an *internal swap rule* is to give the developer the power to provide *declarative input* to the environment about *when* a representation change is appropriate. Classically, such an internal swap rule describes which representation corresponds best with an observed *state* of the data structure. The intent is clearly declarative, for instance: "if the matrix *is* sparse, change to a sparse representation.".

Although internal swap rules are supposed to react to a change of the state of the data structure, the current implementations simply execute the body of the swap rule before the invocation of each operation, regardless of the actual occurrence of a state change. All of our examples show swap rules which exhibit an inherent declarative nature. Therefore, we conjecture that a more mature implementation should take advantage of the intended declarative nature of swap rules to avoid the overhead of excessive checks at runtime. In future implementations of JITds we want to incorporate established techniques from the area of dynamic analysis. The first steps towards the dynamic analysis community have already been taken in the form of a discussion paper [18]. For instance, we could use the RETE algorithm to *react* with a representation change when a certain predicate holds, instead of continuously computing the result of the predicate.

10.4 Epilogue

JITds is the first programming language that enables a shift from choosing a single data structure for a program to choosing a set of data representations. Concretely, the design of JITds enables developers to define data structures that can change their representation at runtime in order to implement both functional and non-functional features (e. g., performance). Improving the performance in practice remains difficult for the average programmer and should be delegated to an expert performance engineer. The design of JITds enables application developers and performance engineers to co-design their applications. Developers that are responsible for the general application logic can use just-in-time data structures while being agnostic about the current data representation and just-in-time data representation changes. A performance expert, on the other hand, can develop a just-in-time data structure without cluttering the general application logic.

Appendix A

The Case of the Missing Cache Hits

In section 2.1 we report on three examples of performance estimations that allows us to conclude that on contemporary hardware, the ladder of abstraction has become so high, that it is not longer feasible for an average programmer to fully understand the behaviour, and thus the performance characteristics, of computer programs. The first two examples were tailored toward showing the effects of caches and branch prediction, respectively. The third example, is a summary of a multi-month, multi-person endeavour of understanding and explaining the performance of a contemporary piece of hardware. In this appendix we report in detail on this endeavour, which is work in collaboration with Tom Van Cutsem and David Ungar, and which was integrally published at the International Conference on Performance Engineering 2013 [23].

This appendix shows that even the simplest hardware, running the simplest programs, can behave in the strangest of ways. Tracking down the cause of a performance anomaly without the complete hardware reference of a processor is a prime example of black-box architectural exploration. When doubling the work of a simple benchmark program (i. e., less than 20 lines of C code), that was run on a single core of Tilera's TILEPro64 processor, did not double the number of consumed cycles, a mystery was unveiled. After ruling out different levels of optimisation for the two programs, a cycle-accurate simulation attributed the sub-optimal performance to an abnormally high number of L1 data cache misses. Further investigation showed that the processor stalled on every Read-After-Write instruction sequence when the following two conditions were met: 1) there are 0 or 1 instructions between the write and the read instruction and 2) the read and the write instructions target distinct memory locations that share an L1 cache line. We call this performance pitfall a *RAW hiccup*. We describe two countermeasures, memory *padding* and the explicit introduction of *pipeline bubbles*, that sidestep the RAW hiccup.

A.1 Introduction

In the context of our research in parallel algorithms for many-core architectures [24, 55], we chose to explore the TilePro64 processor by writing a simple program and measuring

its absolute and relative performance on a *single core*. To verify the measured results, we doubled the work of our program expecting the running time to double as well.

The experiment, however, revealed that *duplicating the work did not double the execution time*. What could possibly be causing this anomalous performance? And how does one track down such changes in performance efficiency? We conducted a series of experiments, timing and simulating different code sequences. Each step answered one question only to ask another. Finally, we were able to pinpoint the instruction and memory reference sequence that was responsible. Our hunt for the performance anomaly is a nice example of black-box architectural exploration, as we did not have a access to a *complete* hardware design reference, and provides evidence for our claim that estimating performance is hard.

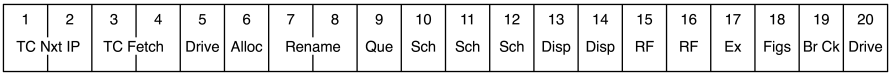
The rest of this appendix is organised as follows: First the hardware used is discussed (appendix A.2), followed by a presentation of the two benchmark programs that are used throughout the example (appendix A.3). Appendix A.4 elaborates on the *expected* and *measured* performance of both programs, revealing a significant discrepancy between the two. In appendix A.5 the compiler generated instruction sequences of the benchmark programs are discussed. Appendix A.6 focusses on processors stalls and how they map to the source code. Appendices A.7 and A.8 each describe a countermeasure that sidesteps the RAW hiccup in the benchmark programs. Finally, in appendix A.9 we compare the RAW hiccup to similar performance pitfalls.

A.2 The platform: TILEPro64 processor

At first sight, Tilera's TILEPro64 [83] might look like a complex many-core processor chip. Its 8x8 mesh network connects 64 processing cores, keeps two levels of distributed cache coherent, and supports inter-core communication. Each core utilises a three-way Very Long Instruction Word (VLIW) architecture to support explicit *instruction level parallelism* (ILP) by executing up to three *bundled instructions* simultaneously by one of the three pipelines of a single core.

But, when looked at in isolation, a single pipeline of a single core of the TILEPro64 processor has a relatively simple architecture. The in-order pipelines execute an in-

(a) 5 Stage Pipeline of the TILEPro64



(b) 20 Stage pipeline of the Intel Pentium IV.

Figure A.1: The Tilera TilePro64 processor has a much more shallow pipeline than for instance Intel's Pentium IV. Therefore, understanding programs and predicting instruction timings on the TilePro64 processor should be relatively easy

Listing A.1: Program 1

```
1   for ( long i = 0 ; i < N ; i++ ) {
2    long sum0 = 0;
3
4    for ( long j0 = 0 ; // init
5          j0 < i ;      // test
6          ++j0 ) {      // incr
7
8     sum0 += j0;       // body
9
10   }
11
12
13
14  }
```

Listing A.2: Program 2

```
1   for ( long i = 0 ; i < N ; i++ ) {
2    long sum0 = 0;
3
4    for ( long j0 = 0 ; // init
5          j0 < i ;      // test
6          ++j0 ) {      // incr
7
8     sum0 += j0;       // body
9
10   }
11   total += sum0;
12
13   long sum1 = 0;
14
15   for ( long j1 = 0 ; // init
16         j1 < i ;      // test
17         ++j1 ) {      // incr
18
19    sum1 += j1;       // body
20
21   }
22   total += sum01;
23  }
```

Figure A.2: Two simple benchmark programs where the second program has exactly twice the work to do compared to the first program. Program 2 is expected to consume twice as many cycles as Program 1.

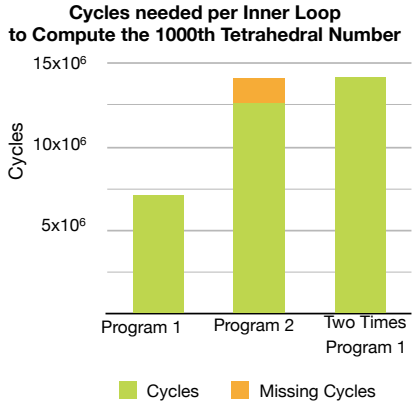
struction in 5 stages (figure A.1a): a rather shallow pipeline compared to contemporary hardware that have pipelines as deep as 20 stages (figure A.1b) [42]. With this short, in-order pipeline the TILEPro64 aims at low branch and low load-to-use latencies. Thus when considering a single pipeline on a single core, the TILEPro64 can arguably be described as simple hardware and performance prediction of any sequential programs it runs should be fairly easy. But even the simplest hardware running the simplest programs, can behave in the strangest of ways. How simple could those programs be?

A.3 The program: Tetrahedral Numbers

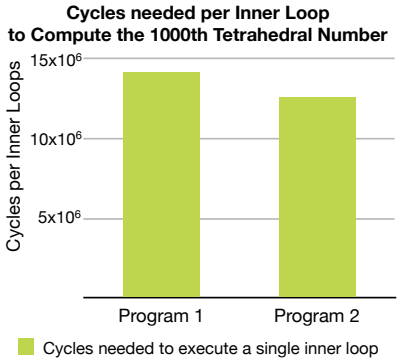
We chose a very simple algorithm for our benchmark: the computation of the nth tetrahedral number. To compute the nth tetrahedral number it suffices to accumulate the n first triangular numbers [71]. And to compute the nth triangular number it suffices to accumulate the n first non-zero integers [72]. Mathematically this could be written as $\sum_{i=0}^{N}\sum_{j=0}^{i} j$ and Program 1 (figure A.2) is the straightforward translation of this formula into C-code. Thus, Program 1 consists of one outer loop (lines 1-14 in figure A.2) with one inner loop (lines 4-10 in figure A.2), of which the body consists of a single statement accumulating a counter (line 8 in figure A.2). Program 2 (figure A.2) just about doubles the work of Program 1 by repeating the inner loop (lines 4-10 and lines 15-21 in figure A.2). When measurements revealed that *doubling the work did not double the execution time,* we were mystified.

A.4 Measured Performance

"To measure is to know", but in this case measuring the performance of both benchmark programs raised more questions than it resolved. We measured the absolute performance of both programs in *cycles needed to complete the outer loop*. Computing the 1000th tetrahedral number by running Program 1 requires 14×10^6 cycles. Doubling the work by running Program 1 twice, requires 28×10^6 cycles. Surprisingly, when running Program 2 which also does twice the work as Program 1, only 25×10^6 cycles were consumed. The difference of 3×10^6 cycles between expected and measured performance, depicted as the lighter coloured box in figure A.3a, is too large to be attributed to the overhead of running two outer loops when running Program 1 twice.



(a) Cycles consumed by Program 1 and Program 2, compared to the number of expected cycles for Program 2.



(b) Average number of cycles needed by Program 1 and Program 2 to compute a single inner loop. Surprisingly, Program 2 needs fewer cycles than Program 1.

Put differently, as depicted in figure A.3b, when looking at the average number of cycles needed to complete a single inner loop, we see that Program 1 needs 14×10^6 cycles, as opposed to the 12.5×10^6 cycles needed by Program 2. These data suggest that the loops in Program 2 run more efficiently. But why would the same source code run faster?

A.5 Comparing Instruction Sequences

The execution of the inner loops require quadratic time, actually the number of additions needed to compute the nth tetrahedral number is equal to the nth triangular number, and thus indeed *quadratic* in function of N. Since *the inner loops dominate the overall performance* of the benchmark programs, and the outer loops induce only linear overhead, and we focused on the inner loops.

Could it be that the compiler was generating different instruction sequences for syntactically equal inner loops? If this were true, the assumption that Program 2 did exactly twice the work of Program 1 would be false, and the unexpected execution times could

be explained. So the first step in unraveling the mystery was to guarantee that the generated machine instruction sequences for the two programs were similar for all inner loops.

Listing A.3: The inner loop of Program 1 compiled into assembler code (-O0). The last 19 instructions dominate the performance of Program 1, and are executed $\frac{N\times(N-1)}{2}$ times.

```
1   .L_init:          #
2   addi r1, sp, 16   # j0
3   sw   r1, zero     #
4   .L_test:          #
5   addi r2, sp, 12   # i
6   lw   r2, r2       #
7   addi r3, sp, 16   # j0
8   lw   r3, r3       #
9   slte r2, r2, r3   #
10  bnz  r2, .L_end   #
11  .L_body:          #
12  addi r5, sp, 16   # j0
13  lw   r5, r5       #
14  addi r6, sp, 20   # sum0
15  lw   r6, r6       #
16  add  r5, r5, r6   #
17  addi r4, sp, 20   # sum0
18  sw   r4, r5       #
19  .L_increment:     #
20  addi r7, sp, 16   # j0
21  lw   r7, r7       #
22  addi r7, r7, 1    #
23  addi r3, sp, 16   # j0
24  sw   r3, r7       #
25  j    .L_test      #
26  .L_end:           #
```

We used Tilera's gcc compiler 2.0.2 with optimisation flag -O0 which compiled the inner loop of Program 1 into the 21 instructions (and 5 labels) shown in listing A.3. The first two instructions (lines 2–3) initialise the inner loop and are only executed a linear number of times. More interesting were the remaining 19 instructions, which formed the heart of the computation and were executed $\frac{N\times(N-1)}{2}$ times each. Ignoring the small constant and linear overhead induced by the inner and outer loops, $19\times\frac{N\times(N-1)}{2}$ was a fair approximation of the total number of executed instructions for Program 1 when N was sufficiently large.

This instruction sequence can be further optimised. The most invasive and effective optimisation would be to reduce the program to compute the formula $\frac{N\times(N+1)\times(N+2)}{N}$ which would be semantically equivalent to Program 1 but would require only constant time for any input. But even if we wanted to keep the structure of the computation, many optimisations were possible. For example, the computation in Program 1 used only 4 variables and the value of the stack pointer (register) is not changed during the execution. Therefore it would be possible to compute the absolute addresses of the four variables once and store them in a dedicated register. This optimisation could save seven `addi` instructions in each iteration. The difference in relative performance would be accounted for if the compiler had applied this or any other optimisation to the inner loops of Program 2, but not to the inner loop of Program 1. But a comparison of the generated assembler instruction sequences for both Program 1 and Program 2 revealed that *all three inner loops were compiled into the same 21 instructions*. The instruction sequences for each loop differed only in the relative addresses of the variables. At this point, it was clear that twice as many assembler instructions were executed for Program

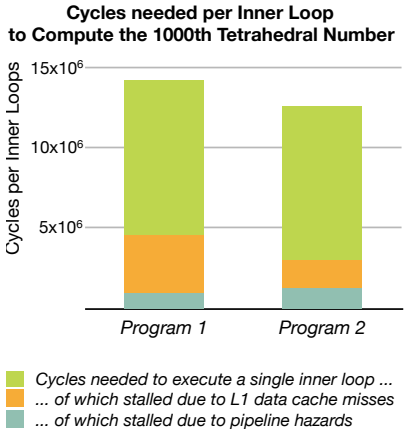


Figure A.4: Relative to the amount of work, Program 1 suffered from many more L1 data cache misses than Program 2. This discrepancy explained the difference in performance, but left us wondering about the cause of the discrepancy.

2, than for Program 1. Yet, Program 2 was executing those instructions more efficiently. What could explain this behaviour?

If an instruction sequence gets more densely packed into instruction bundles such that the instruction level parallelism supported by the VLIW architecture gets exploited, it will run faster. Could it be possible that the instructions of Program 2 got bundled more efficiently than those for Program 1? The assembly-level instruction sequence, as shown in listing A.3, did not make these bundles explicit. And since the processor cores operate in-order they are not responsible for any implicit instruction level parallelism themselves. When we decompiled the machine instructions back into assembler code we saw that all instructions were wrapped in a single bundle and thus issued to be executed sequentially without any instruction level parallelism. *Bundles were not the answer.*

Because all executed bundles were single-instructions, the processor executed the exact same instruction sequence once for Program 1 and twice for Program 2. Why did the processor work at different speeds in each program?

A.6 Cache misses caused the processor to stall

If a processor consumes a different amount of cycles for equal instruction sequences, it must be stalling somewhere. A cycle-accurate simulated execution of the applications reported that the processor was indeed stalling, and attributed these stalls to L1 and L2 instruction- and data-cache misses, to pipeline hazards, and to mispredicted branches. Of these categories, only the *L1 data cache misses* and the *pipeline hazards* were numerous enough to observably impact the performance. If these stalls were the true culprits, then the anomalous increase in efficiency of Program 2 over Program 1 should also have

been reflected in the number of corresponding stalls. For stalls attributed to pipeline hazards, doubling the work also doubled the number of stalls. A regular evolution, so that category was exonerated. Remarkably, *the number of stalls caused by L1 data cache misses was not affected by doubling the work*.

Even considering each program in isolation, the number of L1 data cache misses was unexpectedly high. Because both benchmark programs used only 4 and 6 variables respectively, we *expected no misses* at all. But the clue to the mystery was in the observation that the number of L1 data cache misses did not increase when the amount of work was doubled. Figure A.4 shows these data relative to the number of inner loops: on average a single inner loop of Program 2 suffered from half as many L1 data cache stalls as the loop of Program 1. Why was Program 2 more efficient?

Listing A.4: Program 1: The bulk of L1 data cache misses occur on the increment of the only inner loop.

```
1   for ( long i = 0 ; i < N ; i++ ) { //      1
2     long sum0 = 0;
3
4     for ( long j0 = 0 ; // init          199
5           j0 < i ;      // test
6           ++j0 ) {      // incr         4950
7
8       sum0 += j0;       // body
9
10    }
11
12
13
14  }
```

Listing A.5: Program 2: The bulk of L1 data cache misses occur on the increment of the second inner loop.

```
1   for ( long i = 0 ; i < N ; i++ ) { //      1
2     long sum0 = 0;
3
4     for ( long j0 = 0 ; // init
5           j0 < i ;      // test
6           ++j0 ) {      // incr
7
8       sum0 += j0;       // body
9
10    }
11    total += sum0;
12
13    long sum1 = 0;
14
15    for ( long j1 = 0 ; // init
16          j1 < i ;      // test
17          ++j1 ) {      // incr         4950
18
19      sum1 += j1;       // body
20
21    }
22    total += sum01;
23  }
```

Figure A.5: To understand the reason for all these cache misses, a better view is needed on where they actually occur. Program 2 has one normal and one affected loop. Since these are syntactically equivalent, memory layout of variables is a probable culprit of the discrepancy.

More rigorous simulation revealed that all the L1 data cache misses were read misses. This commonality limited the instructions where the misses could occur to the `lw` (load word) instructions of the program (lines 6, 8, 13, 15, and 21 in listing A.3). On the C-code level, the cycle-accurate simulator indicated that the loop-increment operations on $j0$ (line 6 in listing A.4) in Program 1, and on $j1$ (line 17 in listing A.5) in Program 2 were *responsible for almost all stalls*, as is shown in figure A.5 where the source lines of Program 1 and Program 2 are annotated with the number of observed cache misses. The synthesis of these two pieces of evidence allowed us to identify the slow instruction in each program. In Program 1 this would be the load instruction `lw r7 r7` shown on

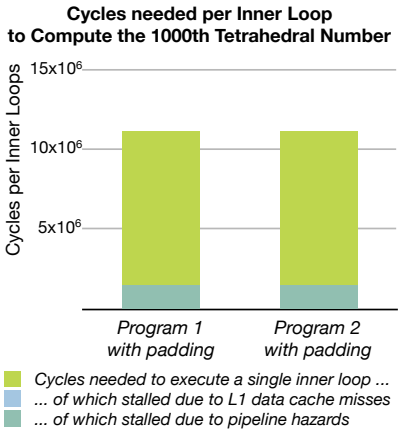


Figure A.6: Adaptations of Program 1 and Program 2 that add padding around the inner loop counters make all L1 data cache misses disappear. Consequently, also the change in efficiency is gone and both programs behave as expected.

line 21 of listing A.3. When N is 1000, the number of inner loop iterations is 4950 which was exactly the number of observed L1 data cache misses on line 6 and 17 of Program 1 and Program 2 respectively.

What we know now is that the inner loop of Program 1 ran slowly because *every iteration induced a cache miss*. One of the inner loops of Program 2 ran equally slowly for the same reason, but on average Program 2 was more efficient because the other inner loop never suffered from a cache miss. But why did the processor have to wait for data that should have been in its L1 cache in the first place?

A.7 Padding Resolves the Cache Misses

Recall that the difference in performance of our two benchmark programs was caused by an abnormally large number of L1 data cache misses that were not expected and moreover did not increase with the amount of work. A cache miss occurs when a processing unit fails to access a piece of data in the cache which results in a much more expensive operation that reroutes the memory instruction to the next level of memory. In this case reading from the L1 data cache fails, causing a load from the L2 cache which is 7 cycles away on the TILEPro64.

Consider Program 1, which used only 4 variables for its entire computation. By the time the instruction on line 15 in listing A.3 had been executed for the first time, we expected all the variables to reside in the L1 cache and to stay there for the remainder of the computation. This expectation arose because 8KB of L1 data cache is plenty of room for storing 4 values of 4 bytes each: they even fit on a single cache line as the TILEPro64's L1 cache lines are 16 bytes wide [83]. Even for the 6 variables of Program 2 the 8KB

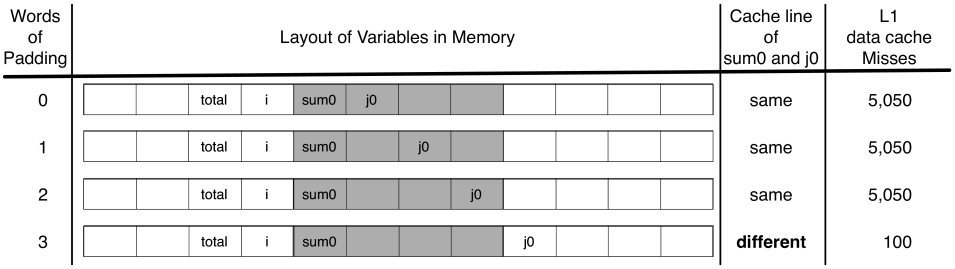


Figure A.7: With as few as 4 variables there is no reason to suffer from cache eviction. In parallel computing padding is a tried and true approach to tackle false sharing, a phenomenon where cache eviction is also unanticipated. Adding padding between $sum0$ and $j0$, such that they reside on different cache lines, causes a significant drop in L1 data cache misses.

should have more than sufficed. Thus any L1 data cache line that was invalidated in the execution of either program qualified as unanticipated behaviour.

In parallel computing, a case of unanticipated cache invalidation, called *false sharing* occurs when a computation writes to a memory location that resides on the same cache line of a distinct memory location used by a concurrent computation [65]. False sharing can be reduced by padding the memory layout of variables so that the memory locations used by concurrent computations do not share cache lines [29]. Padding is a low-level programming technique in which, usually unused, memory is allocated around variables to obtain a more suitable layout of variables in the caches and/or memory. Although this line of inquiry seems far-fetched in the case of a sequential program using only 4 variables, we experimented with memory layout anyway because the distributions of variables over cache lines can affect the number of misses.

Figure A.7 shows four possible layouts of the variables over different cache lines when padding is introduced. Each line shows three cache lines (alternating colours) of four times 4 bytes each. The first line in figure A.7 shows the memory layout as observed for Program 1 without any padding: the first cache line (white) contains the variables $total$ and i, the consecutive cache line (grey) contains the variables $sum0$ and $j0$, finally the third cache line (white) contains no variables relevant to our case. The others three layouts in figure A.7 show how introducing padding before $j0$ moved $j0$ into a different cache line than the other three variables.

Program 1, when adapted such that $j0$ resided in its own cache line, consumed only 11×10^6 cycles, as opposed to the 14×10^6 cycles it had consumed before. The cycle-accurate simulation showed that the *padding eliminated all the L1 data cache misses*. Further, when we moved the variable $j1$ of Program 2 to a separate cache line, the number of cycles consumed by Program 2 dropped from 25×10^6 to 22×10^6 . The import of this performance improvement is that neither benchmark program suffered from unanticipated L1 data cache misses when a change in the memory layout places the inner loop counter and the inner loop accumulator on different L1 data cache lines.

Summarised, the `lw` instruction (line 21 in listing A.3) causes the processor to stall if

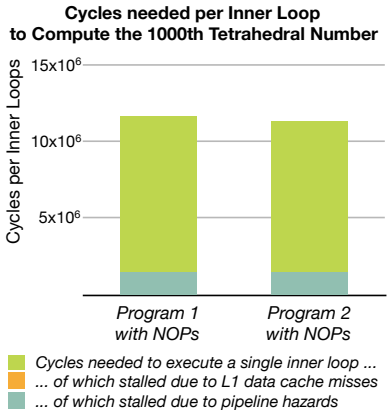


Figure A.8: Adding NOPs to the bodies of the slow inner loops of Program 1 and Program 2 make all L1 data cache misses disappear. Consequently, also the change in efficiency is gone and both programs behave as expected.

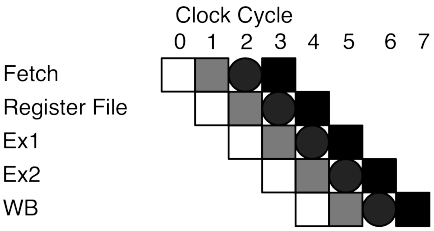
the preceding `sw` instruction (line 18 in listing A.3) targets the same L1 data cache line. For this reason we call this behaviour a *read-after-write hiccup*, or *RAW hiccup* for short. In our example, the `sw` and `lw` instructions are separated by an `addi` instruction (line 20 in listing A.3) which raised the question if, besides the memory layout, also the `addi` instruction plays a role in the RAW hiccup?

A.8 Injecting Pipeline Bubbles

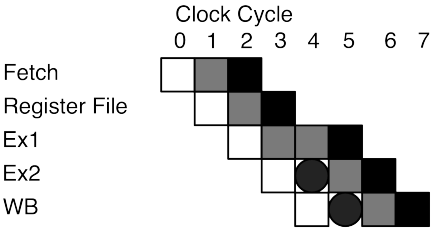
Isolating the missed variable in a different cache line sidestepped the RAW hiccup, but could instruction reordering accomplish the same?

Data hazards occur when subsequent instructions have data dependencies and are executed at the same time in a pipeline. Many contemporary processors use out-of-order execution to avoid these dependencies [41]. The TILEPro64 processor, however, supports only in-order execution. In that case the only way to avoid the data hazard is by introducing a pipeline bubble. Pipeline bubbling is an instruction scheduling technique that prevents data hazards from occurring by delaying the execution of dependent instructions in the pipeline. Typically this is done by the processor's logic by stalling the execution of the depending instruction. A compiler could simulate this behaviour by inserting a no-operation instruction (`NOP`), but what if we hardcoded such a `NOP` in the slow running inner loops of Program 1 and Program 2?

In our benchmark programs there was no actual data dependency between the memory instructions on lines 18 and 21 (see listing A.3), the targeted memory locations only shared a cache line. Thus, adding a *NOP* in the body of the slow running inner loop should have made the program run even slower because the inner loop now consisted out of 20 instead of 19 instructions. Perversely, the measured performance was closer to



(a) Pipelined execution of 3 instructions where the processor stalls on the Ex1 stage of the second instruction.



(b) Pipelined execution of 2 instructions, a NOP, and another instruction where the processor does not stall.

Figure A.9: The overall performance of a processor stalling for one cycle in a sequence of three instructions, is equivalent to the performance of executing that same sequence with an additional NOP instruction if the processor does not stall.

that of the fast programs with padding, than to the execution time of the original benchmark programs with all the cache misses. The cycle-accurate simulator indicated that *adding a* `NOP` *to the body of the slow inner loops removed all L1 data cache misses.*

Synthesising the effects observed when either padding or pipeline bubbles were introduced, allowed us to conclude that *on the TILEPro64 a RAW hiccup occurs in a read-after-write instruction sequence if two conditions are met. First, if both the store and the load instruction target distinct memory locations that share an L1 data cache line, and second if at most one instruction separates the store and the load.* If either condition is eliminated, the RAW hiccup is gone.

A.9 Similar Performance Pitfalls

From the software perspective, pinpointing the origin of anomalous performance to a specific instruction and memory reference sequence suffices to render a program more performant simply by avoiding that sequence. From a hardware perspective, the question remains what architectural design choices caused the anomalous performance. To the best of our knowledge the details of hardware implementations we are hitting in this concrete example are not documented by the chip producer. Without these details we can only make an educated guess about the concrete origin of the stalls.

However, three well know performance pitfalls exist that look similar to the RAW hiccup: *false sharing*, *load-hit-store*, and *write misses*.

False sharing *False sharing* only occurs in the case of concurrent processes, and thus does not apply in the case of a *RAW hiccup* which occurs in a single thread of control [29]. But, besides the number of processes the commonalities are omnipresent: both performance pitfalls occur when a sequence of memory instructions, with at least one write, target unrelated variables that share a cache line.

Load-hit-Store A single process can suffer from the performance pitfall *load-hit-store* (LHS) when a read is issued too soon after a write to read the new value [40]. The RAW hiccup also issues a read too soon after a write, but as opposed to a LHS, the memory instructions target *distinct memory locations*.

Write Stall Finally, when a processor must wait for writes to complete during *write through*, the processor is said to *write stall* [41]. The L1 data cache of the TILEPro64 uses the *write-through policy*, thus what we called a RAW hiccup could actually be a write stall. Other literature, however, refines the definition of write stalls in the context of write-through caches as the delay caused when *a write encounters another write in progress* [31]. Thus excluding read instructions as the origin of a write stall. This makes the instruction sequence causing a RAW hiccup different from the instruction sequence causing a write stall.

Identifying the actual implementation details that caused the RAW hiccup is notoriously difficult without a complete hardware reference. We recognised *false sharing*, *load-hit-stores*, and *write misses* as performance pitfalls similar to the RAW hiccup. They differ, however, from the RAW hiccup in either number of processes, targeted memory locations, or instruction sequence.

A.10 Conclusions

A simple C program exhibited a significant discrepancy between its expected and measured performance. The origin of the anomaly was found to be a counter-intuitive multi-cycle processor stall that occurred whenever a store instruction was followed within two instructions by a load instruction targeting the same L1 data cache line. We called this performance pitfall a RAW hiccup. Eliminating either of the two conditions caused the RAW hiccup to disappear: The introduction of sufficient padding on the one hand and the introduction of a manual pipeline bubble on the other hand removed all anomalous L1 data cache misses.

This appendix reports on the hunt for a performance anomaly observed when studying the behaviour of a trivially simple sequential program. The exact instruction and memory reference sequence that was responsible was found only after performing various experiments and simulations each one a step down the ladder of abstraction. This experience raises the question that if we can not predict the performance of trivially simple sequential code on a simple processing unit, how can we hope to understand the performance of parallel applications running on an 8x8 mesh network-on-chip?

Appendix B

Auxiliary Functions for JIT$\Delta\sigma$

- find-class(P, Cn) = cd — find a class definition in a program

$$\frac{\texttt{cd} \in P \qquad \textsf{class-name}(\texttt{cd}) = \texttt{Cn}}{\textsf{find-class}(P, \texttt{Cn}) = \texttt{cd}}\ \textsc{Find\_Class}$$

- class-name(cd) = Cn — extracts the name from a class definition

$$\frac{}{\textsf{class-name}(\textbf{class}\ \texttt{Cn}\ \{\overline{\texttt{fd}}\ \overline{\texttt{md}}\}) = \texttt{Cn}}\ \textsc{Simple\_Class\_Name}$$

$$\frac{}{\textsf{class-name}(\textbf{class}\ \texttt{Cn}_{jit}\ \textbf{combines}\ \overline{\texttt{Cn}_{rep}}\{\ \overline{\texttt{td}}\ \}) = \texttt{Cn}_{jit}}\ \textsc{JIT\_Class\_Name}$$

- simple-class(cd) — verifies if a class definition defines a simple class

$$\frac{}{\textsf{simple-class}(\textbf{class}\ \texttt{Cn}\ \{\overline{\texttt{fd}}\ \overline{\texttt{md}}\})}\ \textsc{Simple\_Class}$$

- jit-class(cd) — verifies if a class definition defines a just-in-time class

$$\frac{}{\textsf{jit-class}(\textbf{class}\ \texttt{Cn}_{jit}\ \textbf{combines}\ \overline{\texttt{Cn}_{rep}}\{\ \overline{\texttt{td}}\ \})}\ \textsc{JIT\_Class}$$

- class-fields(cd) = $\overline{\texttt{fd}}$ — extracts the field declarations from a class

$$\frac{}{\textsf{class-fields}(\textbf{class}\ \texttt{Cn}\ \{\overline{\texttt{fd}}\ \overline{\texttt{md}}\}) = \overline{\texttt{fd}}}\ \textsc{Class\_Fields}$$

- class-field-names(cd) = $\overline{\texttt{fn}}$ — extracts the field names from a class

$$\frac{\textsf{class-fields}(\texttt{cd}) = \overline{\texttt{fd}} \qquad \overline{\textsf{field-name}(\texttt{fd}) = \texttt{fn}}}{\textsf{class-field-names}(\texttt{cd}) = \overline{\texttt{fn}}}\ \textsc{Class\_Field\_Names}$$

- class-field-types(cd) = $\overline{\texttt{Cn}_\texttt{f}}$ — extracts the field types from a class

$$\frac{\textsf{class-fields}(\texttt{cd}) = \overline{\texttt{fd}} \qquad \overline{\textsf{field-type}(\texttt{fd}) = \texttt{Cn}_\texttt{f}}}{\textsf{class-field-types}(\texttt{cd}) = \overline{\texttt{Cn}_\texttt{f}}}\ \textsc{Class\_Field\_Types}$$

- class-methods(cd) = $\overline{\texttt{md}}$ — extracts the method declarations from a class

$$\frac{}{\text{class-methods}(\textbf{class}\ \texttt{Cn}\ \{\overline{\texttt{fd}}\ \overline{\texttt{md}}\}) = \overline{\texttt{md}}}\ \text{class-methods}$$

- parameter-name(pd) = pn — extracts the name from a parameter definition

$$\frac{}{\text{parameter-name}(\texttt{cn}\ \texttt{pn}) = \texttt{pn}}$$

- parameter-type(pd) = Cn — extracts the name from a parameter definition

$$\frac{}{\text{parameter-type}(\texttt{cn}\ \texttt{pn}) = \texttt{Cn}}$$

- field-name(fd) = pn — extracts the name from a field definition

$$\frac{}{\text{field-name}(\texttt{cn}\ \texttt{fn}) = \texttt{fn}}$$

- field-type(pd) = Cn — extracts the name from a field definition

$$\frac{}{\text{field-type}(\texttt{cn}\ \texttt{fn}) = \texttt{Cn}}$$

- method-name(md) = mn — extracts the method name from a method definition

$$\frac{}{\text{method-name}(\texttt{Cn}_{ret}\ \texttt{mn}\ (\overline{\texttt{pd}})\ \ \{\overline{\texttt{stmt}_B}\ \textbf{return}\ \texttt{x;}\ \}) = \texttt{mn}}$$

- method-type(md) = $(\texttt{Cn}_r \rightarrow \overline{\texttt{Cn}_a})$ — extracts the method type from a method definition

$$\frac{\overline{\text{parameter-type}(\texttt{pd}) = \texttt{Cn}_{arg}}}{\text{method-type}(\texttt{Cn}_{ret}\ \texttt{mn}\ (\overline{\texttt{pd}})\ \ \{\overline{\texttt{stmt}_B}\ \textbf{return}\ \texttt{x;}\ \}) = \overline{\texttt{Cn}_{arg}} \rightarrow \texttt{Cn}_{ret}}$$

- class-representation-names(cd) = $\overline{\texttt{Cn}_{rep}}$ — extracts the names of the representations from a class

$$\frac{}{\text{class-representation-names}(\textbf{class}\ \texttt{Cn}_{jit}\ \textbf{combines}\ \overline{\texttt{Cn}_{rep}}\{\ \overline{\texttt{td}}\ \}) = \overline{\texttt{Cn}_{rep}}}\ \textsc{Class\_Representations}$$

- class-transition-functions(cd) = $\overline{\texttt{td}}$ — extracts the names of the representations from a class

$$\frac{}{\text{class-transition-functions}(\textbf{class}\ \texttt{Cn}_{jit}\ \textbf{combines}\ \overline{\texttt{Cn}_{rep}}\{\ \overline{\texttt{td}}\ \}) = \overline{\texttt{td}}}\ \textsc{Class\_Transition\_Functions}$$

- find-method(cd, mn) = md — find the method with a given name in a class definition

$$\frac{\texttt{md} \in \text{class-methods}(\texttt{cd}) \qquad \text{method-name}(\texttt{md}) = \texttt{mn}}{\text{find-method}(\texttt{cd}, \texttt{mn}) = \texttt{md}}\ \textsc{Find\_Method}$$

- find-field(cd, fn) = fd — find the field with a given name in a class definition

$$\frac{\texttt{md} \in \text{class-fields}(\texttt{cd}) \qquad \text{field-name}(\texttt{md}) = \texttt{fn}}{\text{find-field}(\texttt{cd}, \texttt{fn}) = \texttt{fd}}\ \textsc{Find\_Field}$$

- find-transition-function($\texttt{cd}_{jit}$, $\texttt{cn}_{src}$, $\texttt{cn}_{tar}$) = td — find the transition function between two representations in a class definition

$$\frac{\begin{array}{c}\texttt{td} \in \text{class-transition-functions}(\texttt{cd}_{jit}) \\ \text{transition\_function\_source}(\texttt{td}) = \texttt{Cn}_{from} \qquad \text{transition\_function\_target}(\texttt{td}) = \texttt{Cn}_{to}\end{array}}{\text{find-transition-function}(\texttt{cd}_{jit}, \texttt{Cn}_{from}, \texttt{Cn}_{to}) = \texttt{td}}\ \textsc{Find\_Transition\_Functions}$$

Appendix C

`SparseMatrix` implementations

A sparse matrix is matrix where the number of *non-zero values* are greatly outnumbered by the zero's in that matrix. In scientific or engineering applications matrices are often sparse. In software, it is possible to take advantage of this property to store such matrices more efficiently, i. e., it is possible to store the the non-zero elements only. Besides the reduction in space complexity (denoted as S(x)), also a reduction in time complexity (denoted as T (x)) can be achieved, e. g., by using dedicated algorithms tailored towards sparse matrices.

In the main text (e. g., section 2.2) we extensively presented two implementations for dense matrices, i. e., `RowMajorMatrix` and `ColMajorMatrix`. `SparseMatrix` on the other hand is only introduced conceptually as a representation for matrices where the required space is proportional to the number of non-zero values in the matrix. Concrete implementations of `SparseMatrix` are omitted in section 2.2. Here, we present two: an implementation of the Compressed Row Storage (CRS) scheme and a diagonal matrix. With CRS it is possible to store any matrix, whereas the diagonal matrix representation can only store matrices that are actually diagonal matrices.

C.1 Compressed Row Storage (CRS)

The Compressed Row Storage (CRS) scheme stores for each row of a matrix only the non-zero values. The *row* (0 1 0 2), for instance, is stored as (1 2), which we refer to as the *data*. This compression, however, looses the information of to which columns these data elements belong. This information — that 1 is stored in column 1 and 2 is stored in column 3 (if we number the columns starting from 0) — can be represented as (1 3). Combined with the *index of the row*, 1, this is all the information there is to store. An implementation of a CRS matrix is given in listing C.1 and more information about CRS can be found in [26].

Listing C.1: Compressed rows storage matrix implementation in Java.

```
1  public class CRSMatrix {
2
3    int    cols;
4    int[] rowIdxs, colIdxs;
5    double[] data;
6
7    public int getRows() { return rowIdxs.length-1; }
8    public int getCols() { return cols; }
9
10   public CRSMatrix(int rows, int cols) {
11     this.cols    = cols;
12     this.rowIdxs = new int[rows+1];
13     this.colIdxs = new int[0];
14     this.data    = new double[0];
15   }
16
17   private ColIdx getColIdx(int row, int col) {
18     int lo = rowIdxs[row];
19     int hi = rowIdxs[row+1];
20     for (int i=lo ; i<hi ; i++) {
21       if ( colIdxs[i] == col ) return new ColIdx(true, i);
22       if ( colIdxs[i] >  col ) return new ColIdx(false,i);
23     }
24     return new ColIdx(false,hi);
25   }
26
27   public double get(int row, int col) {
28     ColIdx idx = getColIdx(row,col);
29     if (idx.found ) return data[idx.idx];
30     else            return 0.0;
31   }
32
33   public void set(int row, int col, double val) {
34     double currentVal = get(row, col);
35     if      ( currentVal == val ) /* do nothing */     ;
36     else if ( currentVal == 0   ) insert(row, col, val);
37     else if ( val        == 0   ) delete(row,col)       ;
38     else                          update(row,col,val)   ;
39   }
40
41   private void insert(int row, int col, double val) {
42     int colIdx = getColIdx(row,col).idx;
43     data       = ArrayUtils.add(data,    colIdx, val);
44     colIdxs    = ArrayUtils.add(colIdxs, colIdx, col);
45     for (int r=row+1 ; r<= getRows() ; r++) rowIdxs[r]++;
46   }
47
48   private void delete(int row, int col) {
49     int colIdx = getColIdx(row,col).idx;
50     data    = ArrayUtils.remove(data,    colIdx);
51     colIdxs = ArrayUtils.remove(colIdxs, colIdx);
52     for (int r=row+1 ; r<= getRows() ; r++) rowIdxs[r]--;
53   }
54
55   private void update(int row, int col, double val) {
56     int colIdx = getColIdx(row,col).idx;
57     data[colIdx] = val;
58   }
59
60 }
```

C.2 Diagonal Matrix

In mathematics, a diagonal matrix is a matrix with only non-zero values in the first diagonal. M is an example thereof.

$$M = \begin{pmatrix} \lambda_0 & 0 & 0 & 0 \\ 0 & \lambda_1 & 0 & 0 \\ 0 & 0 & \lambda_2 & 0 \end{pmatrix}$$

Actually a diagonal matrix is a special kind of sparse matrix which can benefit from a specialised implementation. First, a diagonal matrix takes at most $min(rows, cols)$ space, and second — due to the regularity of diagonal matrix — the storage scheme facilitates a lookup where virtually no computation is required. M can effectively be stored as $(3, 4, [\lambda_0, \lambda_1, \lambda_2])$, a three-tuple containing the rows, the columns and the diagonal respectively. Below we present an implementation in Java, which adheres to the data interface as presented in section 2.2.

Listing C.2: `DiagonalMatrix`

```
1  class DiagonalMatrix {
2    int rows;
3    int cols;
4    double[] data;
5
6    public DiagonalMatrix(int rows, int cols) {
7      this.rows = rows;
8      this.cols = cols;
9      data = new double[Math.min(rows,cols)];
10   }
11
12   int getRows() { return rows; }
13   int getCols() { return cols; }
14
15   double get(int row, int col) {
16     if (row==col) return data[row];
17     else          return 0.0;
18   }
19
20   void set(int row, int col, double val) {
21     if ( (row!=col) && (val!=0) ) {
22       throw new RuntimeException("Can not store data.");
23     } else {
24       data[row] = val;
25     }
26   }
27 }
```

The key idea here is that when `get` or `set` are called with `row==col` the diagonal represented by `data` is consulted (read or write). When `row==col` does not hold, however, we need dedicated behaviour. For `get` we simply return 0. For `set` we require the new value either to be 0 — no harm done — or we have to throw an exception because a `DiagonalMatrix` can only represent diagonal matrices. Note that the inverse is not true: a diagonal matrix is not necessarily represented by a `DiagonalMatrix` (cf. data characteristics versus representation characteristics, see section 2.2).

Appendix D

Multiple Inheritance and Mitigating Ambiguity: *How Do They Do It*

When a class inherits from more than one super class — and if we ignore types — there are two questions that directly pop into mind: "What about methods with the same name *(behaviour)*?" and "What about fields with the same name *(state)*?"

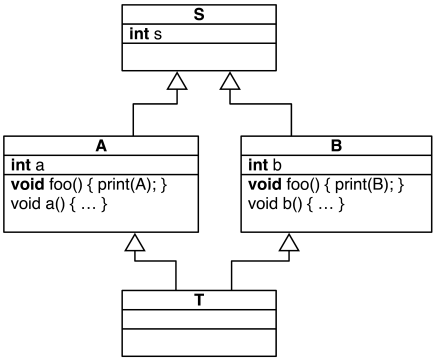


Figure D.1: Example of a typical *diamond*-hierarchy.

These questions identify the ambiguity that arises when dealing with multiple inheritance, which language designers tried to mitigate in various ways. We provide a concise overview:[1]

Ambiguity Rejection A trivial solution to resolve ambiguity is to *not allow* it. A compiler that detects ambiguity usually generates an error or warning, which puts the burden on the developer to work around the problem.

[1] *Traits* and *Mixins* are not discussed in this text: they are arguably different features than multiple inheritance and they run into the same problems.

Protocols allow classes to inherit only the protocol they have to adhere to: only method signatures are inherited, without an implementation. Since, this implies there is only a single implementation, namely that of the *implementing* subclass, no ambiguity on which code to execute is present. In Java for instance a class can inherit from only one super-class, but it can implement as many interfaces— this is what protocols are called in Java —as needed [37].

Linearisation Another approach is to statically define, e. g., as part of the language specification, how method and field lookup is going to be performed. To this end all features in the inheritance hierarchy, a graph, are *linearised* accordingly. The programming language Dylan, for instance uses the C3 algorithm to turn the hierarchy graph into a linear structure [7]. The simplest, and often used, linearisation algorithm is "last man standing", where after textual inclusion of the super classes, the last definition "wins". This is for instance the case in OCAML [57]. Lookups in a linearised hierarchy is similar to lookups in the context of single inheritance.

Select and Rename is a technique to resolve ambiguity by explicitly renaming features with colliding names in the subclass. In Eiffel [44], to remove the ambiguity from figure D.1, the version from `A` could be renamed to `bar` and the version from `B` could be renamed to `baz`.

```
1  class T inherit A rename foo as bar
2                  B rename foo as baz
```

Explicit Inheritance can be seen as a variant of "Select and Rename", where features with colliding names can only be used when they are explicitly qualified with the intended super-class' name. This is for instance allowed in C++, where the call `ptr->A::foo()` explicitly uses the method `foo` of `A` [30].

Method Combination is a technique where ambiguity in the body of an overriding method that needs access to, for instance, a super-method can be resolved programmatically. In Common Lisp (CLOS), for instance, a method can return the *sum* of the results of *all* its super methods [74]. Besides +, eight other primitive method-combinators are available, but defining one's own method-combinators is also possible.

```
1  (defgeneric leaf-count (tree)
2    (:documentation "Return the number of leafs.")
3    (:method-combination +))
```

Most Specific Argument When a subclass overrides a method of one of its ancestors, it is allowed in some languages to *refine* the signature, i. e., methods are co-variant in the return type and contra-variant in their argument types. In CLOS, for instance, the method that matches the argument types the most is invoked, i. e., including dispatch on the argument type. Note that this technique only reduces the "chances" of ambiguity, and when there is still ambiguity present CLOS resorts to other techniques to resolve the ambiguity.

Meta-object protocol Some languages make the behaviour of objects (and classes) first class. A Meta-object protocol (MOP) [50], for instance, allows redefining per object how it should behave. This meta-object protocol includes a definition of how an object should invoke a method. Thus, altering the meta-object of an object gives a programmer full control to implement his own feature resolution algorithm.

Virtual Inheritance All techniques presented above focus on *behaviour*. When *state* is also considered, there is another dimension that comes into play, one that does not make sense in the context of behaviour, i. e., replication versus unification. Consider the hierarchy from figure D.1: is an instance of `T` supposed to have two different `s`'s (replication) or is it supposed to have single, unified, field `s`? This ambiguity can be resolved with *virtual inheritance*, which ensures that the state of a shared ancestor is only present once [56]. C++ [30], for instance, takes this approach. Note that this approach only resolves the problems with replicated members of shared ancestors. The other issues, are mitigated by other techniques (see above).

All these solutions tackle the problem from a "spatial" point of view, i. e., how can we structure the lookup hierarchy in order to remove the ambiguity. Moreover, most of these approaches are static and fix the lookup strategy before execution. Only a few languages allow programmed modifications to the lookup hierarchy at run-time. To the best of our knowledge there is no language, with a static MOP, that dynamically changes the method lookup.

Bibliography

[1] Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition. 3, 25, 41

[2] ACM/IEEE-CS Joint Task Force on Computing Curricula (2013). Computer science curricula 2013. Technical report, ACM Press and IEEE Computer Society Press. 11, 12, 25

[3] Aldrich, J., Sunshine, J., Saini, D., and Sparks, Z. (2009). Typestate-oriented Programming. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '09*, pages 1015–1022. 56, 69, 76, 179, 182, 204

[4] Alpern, B., Cocchi, A., Fink, S., and Grove, D. (2001). Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '01*, pages 108–124. 158

[5] Ansel, J. (2014). *Autotuning Programs with Algorithmic Choice*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA. 219

[6] Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. (2009). PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI) '09*, pages 38–49. 29, 183, 218, 219

[7] Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K., and Withington, P. T. (1996). A Monotonic Superclass Linearization for Dylan. *SIGPLAN Notices*, 31(10):69–82. 250

[8] Bierman, G. M., Parkinson, M. J., and Pitts, A. M. (2003). MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge. 87

[9] Blackburn, S. M., McKinley, K. S., Garner, R., Hoffmann, C., Khan, A. M., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovik, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2008). Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM*, 51(8):83–89. 164

[10] Bolz, C. F., Diekmann, L., and Tratt, L. (2013). Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '13*, pages 167–182. 45, 208, 209, 210

[11] Bracha, G. (website). The Miracle of become. `http://gbracha.blogspot.be/2009/07/miracle-of-become.html`. Accessed: August 3, 2015. 43

[12] Chambers, C., Ungar, D., and Lee, E. (1989). An Efficient Implementation of SELF: a Dynamically-typed Object-oriented Language Based on Prototypes. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '89*, pages 49–70. 2, 154

[13] Cohen, T. and Gil, J. Y. (2009). Three Approaches to Object Evolution. In *Proceedings of International Conference on Principles and Practices of Programming on the Java Platform (PPPJ) '09*, pages 57–66. 51, 81, 82, 143, 179, 201

[14] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition. 12, 14

[15] Costanza, P. (2002). Dynamic Replacement of Active Objects in the Gilgul Programming Language. In *Proceedings of the Working Conference on Component Deployment (CD) '02*, pages 125–140. 2, 50, 54, 180, 206

[16] Dahl, O.-J. and Nygaard, K. (1966). SIMULA: An ALGOL-based Simulation Language. *Communications of the ACM*, 9(9):671–678. 226

[17] Damiani, F., Drossopoulou, S., and Giannini, P. (2003). *Theoretical Computer Science: 8th Italian Conference, ICTCS 2003, Bertinoro, Italy, October 13-15, 2003. Proceedings*, chapter Refined Effects for Unanticipated Object Re-classification:Fickle3, pages 97–110. Springer Berlin Heidelberg, Berlin, Heidelberg. 203

[18] De Wael, M. (2015). Just-in-time Data Structures: Towards Declarative Swap Rules. In *Proceedings of International Workshop on Dynamic Analysis (WODA) '15*, pages 33–34. 6, 129, 149, 223, 229

[19] De Wael, M., Marr, S., De Fraine, B., Van Cutsem, T., and De Meuter, W. (2015a). Partitioned global address space languages. *ACM Computing Surveys*, 47(4):62:1–62:27. 2, 6, 18

[20] De Wael, M., Marr, S., De Koster, J., Sartor, J. B., and De Meuter, W. (2015b). Just-in-time Data Structures. In *Proceedings of Onward! '15*, pages 61–75. 2, 5, 6, 223

[21] De Wael, M., Marr, S., and De Meuter, W. (2014a). Data Interface + Algorithms = Efficient Programs: Separating Logic from Representation to Improve Performance. In *Proceedings of Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS) '14*, pages 2:1–2:4. 2, 6, 223

[22] De Wael, M., Marr, S., and Van Cutsem, T. (2014b). Fork/Join Parallelism in the Wild: Documenting Patterns and Anti-patterns in Java Programs Using the Fork/Join Framework. In *Proceedings of International Conference on Principles and Practices of Programming on the Java Platform (PPPJ) '14*, pages 39–50. 2, 7, 11, 73, 74

[23] De Wael, M., Ungar, D., and Van Cutsem, T. (2013). When Spatial and Temporal Locality Collide: The Case of the Missing Cache Hits. In *Proceedings of International Conference on Performance Engineering (ICPE) '13*, pages 63–70. 6, 24, 231

[24] De Wael, M. and Van Cutsem, T. (2012). How to Achieve Scalable Fork/Join on Many-core Architectures? In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, pages 85–86. 2, 231

[25] Dijkstra, E. W. (1968). Letters to the Editor: Go to Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148. 227

[26] Dongarra, J. (website). Compressed Row Storage (CRS). `http://netlib.org/linalg/html_templates/node91.html`. Accessed: 3 March, 2014. 245

[27] Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. (2002). More dynamic object reclassification: Fickle(II). *Transactions on Programming Languages and Systems (TOPLAS)*, 24:153–191. 2, 49, 54, 77, 81, 202, 203

[28] Efftinge, S. (website). Xtext. `http://www.eclipse.org/Xtext/`. Accessed: February 29, 2016. 152

[29] Eggers, S. and Jeremiassen, T. (1991). Eliminating False Sharing. In *International Conference on Parallel Processing*, volume I, pages 377–381. 239, 241

[30] Ellis, M. A. and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 250, 251

[31] Emer, J. S. and Clark, D. W. (1984). A characterization of processor performance in the vax-11/780. *SIGARCH Computer Architecture News*, 12(3):301–310. 242

[32] Filman, R. E. and Friedman, D. P. (2000). Aspect-oriented programming is quantification and obliviousness. Technical report. 39, 47, 48, 140

[33] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (2012). Cache-oblivious algorithms. *Transactions on Algorithms (TALGS)*, 8(1):4:1–4:22. 12

[34] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc. 43, 147, 154, 182, 184, 225, 226, 227

[35] Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically Rigorous Java Performance Evaluation. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '07*, pages 57–76. 24, 164

[36] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2, 37

[37] Gosling, J., Joy, B., and Steele, G. L. (1996). *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition. 250

[38] Gwennap, L. (1995). New algorithm improves branch prediction. *Microprocessor Report*, 9(4):17–21. 22

[39] Harbulot, B. and Gurd, J. R. (2006). A Join Point for Loops in AspectJ. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development*, pages 63–74. 140

[40] Heineman, B. (website). Common Performance Issues in Game Programming. `http://www.gamasutra.com/view/feature/3687/sponsored_feature_common_.php`. Accessed: June 1, 2008. 242

[41] Hennessy, J. L. and Patterson, D. A. (2006). *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 240, 242

[42] Hinton, G., Sager, D., Upton, M., Boggs, D., Group, D. P., and Corp, I. (2001). The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 1:2001. 233

[43] Hirschfeld, R., Costanza, P., and Nierstrasz, O. (2008). Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151. 220

[44] Howard, R. (1993). The Eiffel Programming Language. *Dr. Dobb's Journal*, 18(11):68–73. 250

[45] Huang, S. S. and Smaragdakis, Y. (2011). Morphing: Structurally shaping a class by reflecting on others. *Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):6:1–6:44. 152

[46] Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight java: A minimal core calculus for java and gj. *Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450. 87

[47] Jung, C., Rus, S., Railing, B. P., Clark, N., and Pande, S. (2011). Brainy: Effective selection of data structures. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI) '11*, pages 86–97. 164, 208, 210, 211

[48] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of aspectj. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) '01*, pages 327–353, London, UK, UK. Springer-Verlag. 146

[49] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) '97*, pages 220–242. 36, 38, 140

[50] Kiczales, G. and Rivieres, J. D. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA. 251

[51] Knuth, D. E. (1974). Structured Programming with Go to Statements. *ACM Computing Surveys*, 6(4):261–301. 59

[52] Lieberman, H., Stein, L., and Ungar, D. (1987). Treaty of Orlando. In *Addendum to the Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '87*, pages 43–44. 81, 84, 85, 157

[53] Liskov, B. and Zilles, S. (1974). Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59. 3, 41, 59, 69

[54] Liu, L. and Rus, S. (2009). Perflint: A Context Sensitive Performance Advisor for C++ Programs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO) '09*, pages 265–274. 211

[55] Marr, S., De Wael, M., Haupt, M., and D'Hondt, T. (2011). Which Problems Does a Multi-language Virtual Machine Need to Solve in the Multicore/Manycore Era? In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 341–348. 2, 231

[56] Milea, A. (website). Solving the diamond problem with virtual inheritance. `http://www.cprogramming.com/tutorial/virtual_inheritance.html`. Accessed: June 11, 2015. 251

[57] Minsky, Y., Madhavapeddy, A., and Hickey, J. (2013). *Real World OCaml*. O'Reilly Media. 250

[58] Miranda, E. and Béra, C. (2015). A Partial Read Barrier for Efficient Support of Live Object-oriented Programming. In *Proceedings of the International Symposium on Memory Management (ISMM) '15*, pages 93–104. 144, 200, 228

[59] Mitchell, N. and Sevitsky, G. (2007). The Causes of Bloat, the Limits of Health. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '07*, pages 245–260. 227

[60] Oracle (website). Class "Object". `http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#toString()`. Accessed: February 29, 2016. 45

[61] Österlund, E. and Löwe, W. (2013). Dynamically transforming data structures. In *Proceedings of the International Conference on Automated Software Engineering*, pages 410–420. 164, 213, 214, 215

[62] Östlund, J. and Wrigstad, T. (2010). Welterweight Java. In *Proceedings of TOOLS '10*, pages 97–116. 87

[63] Overflow, S. (website). Developer Survey 2015. `http://stackoverflow.com/research/developer-survey-2015`. Accessed: July 24, 2015. 11

[64] Parnas, D. L. (1972). On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058. 3

[65] Patterson, D. A. and Hennessy, J. L. (2008). *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition. 239

[66] Perlis, A. J. (1982). Special feature: Epigrams on programming. *SIGPLAN Notices*, 17(9):7–13. 26

[67] Ralston, A., Reilly, E. D., and Hemmendinger, D., editors (2003). *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., Chichester, UK, 4th edition. 3, 41

[68] Ruparelia, N. B. (2010). Software development lifecycle models. *SIGSOFT Softw. Eng. Notes*, 35(3):8–13. 41

[69] Shacham, O., Vechev, M., and Yahav, E. (2009). Chameleon: Adaptive Selection of Collections. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI) '09*, pages 408–418. 134, 164, 208, 211, 212

[70] Sherwood, T., Perelman, E., Hamerly, G., Sair, S., and Calder, B. (2003). Discovering and exploiting program phases. *IEEE Micro*, 23:84–93. 227

[71] Sloane, N. J. A. (websitea). Tetrahedral (or triangular pyramidal) numbers. `http://oeis.org/A000292`. Accessed: February 29, 2016. 233

[72] Sloane, N. J. A. (websiteb). Triangular numbers. `http://oeis.org/A000217`. Accessed: February 29, 2016. 233

[73] Smith, R. B. and Ungar, D. (1996). A simple and unifying approach to subjective objects. *Theory and Practice of Object Systems*, 2(3):161–178. 220

[74] Steele, Jr., G. L. (1990). *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA. 250

[75] Strniša, R. (2010). *Formalising, improving, and reusing the Java Module System*. PhD thesis, University of Cambridge, St. John's College. https://rok.strnisa.com/lj/. 87

[76] Strniša, R., Sewell, P., and Parkinson, M. (2007). The Java Module System: Core Design and Semantic Definition. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '07*, pages 499–514. 87, 91, 104, 119, 120, 121

[77] Strzodka, R. (2011). Abstraction for AoS and SoA layout in C++. In mei W. Hwu, W., editor, *GPU Computing Gems: Jade Edition*. Morgan Kaufmann. 55

[78] Sunshine, J., Naden, K., Stork, S., Aldrich, J., and Tanter, E. (2011). First-class State Change in Plaid. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '11*, pages 713–732. 2, 67, 77, 179, 180, 181, 204

[79] Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3). 2

[80] Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, 1st edition. 136

[81] Team, T. A. (websitea). Join Points and Pointcuts. `https://eclipse.org/aspectj/doc/next/progguide/language-joinPoints.html`. Accessed: December 14, 2015. 131

[82] Team, T. G. (websiteb). MArray (The Glasgow Haskell Compiler User's Guide). `https://downloads.haskell.org/~ghc/4.08/docs/set/sec-marray.html`. Accessed: February 29, 2016. 56

[83] Tilera (2010). *Tile Processor User Architecture Manual*. Tilera. 232, 238

[84] Ureche, V., Biboudis, A., Smaragdakis, Y., and Odersky, M. (2015). Automating ad hoc data representation transformations. Technical report, EPFL. 51, 75, 195, 196, 198

[85] Ureche, V., Burmako, E., and Odersky, M. (2014). Late Data Layout: Unifying Data Representation Transformations. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA) '14*, pages 397–416. 45, 195, 196

[86] Victor, B. (website). Up and Down the Ladder of Abstraction. `http://worrydream.com/LadderOfAbstraction`. Accessed: July 24, 2015. 11

[87] Weisstein, E. W. (website). Cumulative frequency (a wolfram web resource). `http://mathworld.wolfram.com/CumulativeFrequency.html`. Accessed: July 27, 2015. 19

[88] Wirth, N. (1978). *Algorithms + Data Structures = Programs*. Prentice Hall PTR, Upper Saddle River, NJ, USA. 1, 2, 41, 59

[89] Wright, A. and Felleisen, M. (1994). A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94. 110

[90] Xu, G. H. (2013). Coco: Sound and adaptive replacement of java collections. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) '13*, pages 1–26. 134, 208, 211, 215, 216, 218