

EFFICIENT COMPOUND VALUES IN
VIRTUAL MACHINES

Tobias Pape



EFFICIENT COMPOUND VALUES IN VIRTUAL MACHINES

Tobias Pape

Dissertation
zur Erlangung des Doktorgrades der
Digital Engineering Fakultät
der Universität Potsdam

This work is licensed under a Creative Commons License:
Creative Commons Attribution-ShareAlike 4.0 International.
This does not apply to quoted content from other authors.
To view a copy of this license visit
<https://creativecommons.org/licenses/by-sa/4.0/deed.en>



Betreuer: Prof. Dr.-Ing. Robert Hirschfeld
Universität Potsdam,
Digital Engineering-Fakultät,
Software Architecture Group

Gutachter: prof. em. dr. Theo D'Hondt
Vrije Universiteit Brussel,
Software Language Lab

Prof. Dr. Hidehiko Masuhara
Tokyo Institute of Technology,
Department of Mathematical and Computing Science

Datum der Einreichung: 20. Dezember 2019

Datum der Disputation: 15. Dezember 2020

Published online on the
Publication Server of the University of Potsdam:
<https://doi.org/10.25932/publishup-49913>
<https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-499134>



To Julia
Who never fails to amaze me.



Abstract

Compound values are not universally supported in **virtual machine (VM)**-based programming systems and languages. However, providing data structures with value characteristics can be beneficial. On one hand, programming systems and languages can adequately represent physical quantities with compound values and avoid inconsistencies, for example, in representation of large numbers. On the other hand, **just-in-time (JIT)** compilers, which are often found in **VMS**, can rely on the fact that compound values are immutable, which is an important property in optimizing programs. Considering this, compound values have an optimization potential that can be put to use by implementing them in **VMS** in a way that is efficient in memory usage and execution time. Yet, optimized compound values in **VMS** face certain challenges: to maintain consistency, it should not be observable by the program whether compound values are represented in an optimized way by a **VM**; an optimization should take into account, that the usage of compound values can exhibit certain patterns at run-time; and that necessary value-incompatible properties due to implementation restrictions should be reduced.

We propose a technique to detect and compress common patterns of compound value usage at run-time to improve memory usage and execution speed. Our approach identifies patterns of frequent compound value references and introduces abbreviated forms for them. Thus, it is possible to store multiple inter-referenced compound values in an inlined memory representation, reducing the overhead of metadata and object references. We extend our approach by a notion of limited mutability, using cells that act as barriers for our approach and provide a location for shared, mutable access with the possibility of type specialization. We devise an extension to our approach that allows us to express automatic unboxing of boxed primitive data types in terms of our initial technique. We show that our approach is versatile enough to express

another optimization technique that relies on values, such as Booleans, that are unique throughout a programming system. Furthermore, we demonstrate how to re-use learned usage patterns and optimizations across program runs, thus reducing the performance impact of pattern recognition.

We show in a best-case prototype that the implementation of our approach is feasible and can also be applied to general purpose programming systems, namely implementations of the Racket language and Squeak/Smalltalk. In several micro-benchmarks, we found that our approach can effectively reduce memory consumption and improve execution speed.

Zusammenfassung

Zusammengesetzte Werte werden in `VM`-basierten Programmiersystemen und -sprachen nicht durchgängig unterstützt. Die Bereitstellung von Datenstrukturen mit Wertemerkmalen kann jedoch von Vorteil sein. Einerseits können Programmiersysteme und Sprachen physikalische Größen mit zusammengesetzten Werten, wie beispielsweise bei der Darstellung großer Zahlen, adäquat darstellen und Inkonsistenzen vermeiden. Andererseits können sich Just-in-time-Compiler, die oft in `VMS` zu finden sind, darauf verlassen, dass zusammengesetzte Werte unveränderlich sind, was eine wichtige Eigenschaft bei der Programmoptimierung ist. In Anbetracht dessen haben zusammengesetzte Werte ein Optimierungspotenzial, das genutzt werden kann, indem sie in `VMS` so implementiert werden, dass sie effizient in Speichernutzung und Ausführungszeit sind. Darüber hinaus stehen optimierte zusammengesetzte Werte in `VMS` vor bestimmten Herausforderungen: Um die Konsistenz zu erhalten, sollte das Programm nicht beobachten können, ob zusammengesetzte Werte durch eine `VM` in einer optimierten Weise dargestellt werden; eine Optimierung sollte berücksichtigen, dass die Verwendung von zusammengesetzten Werten bestimmte Muster zur Laufzeit aufweisen kann; und dass wertinkompatible Eigenschaften vermindert werden sollten, die nur aufgrund von Implementierungsbeschränkungen notwendig sind.

Wir schlagen eine Verfahrensweise vor, um gängige Muster der Verwendung von zusammengesetzten Werten zur Laufzeit zu erkennen und zu komprimieren, um die Speichernutzung und Ausführungsgeschwindigkeit zu verbessern. Unser Ansatz identifiziert Muster häufiger zusammengesetzter Wertreferenzen und führt für sie abgekürzte Formen ein. Dies ermöglicht es, mehrere miteinander verknüpfte zusammengesetzte Werte in einer eingebetteten Art und Weise im Speicher darzustellen, wodurch der Verwaltungsaufwand, der sich aus Metadaten und Objektreferenzen ergibt, reduziert wird. Wir erweitern

unseren Ansatz um ein Konzept der eingeschränkten Veränderbarkeit, indem wir Zellen verwenden, die als Barrieren für unseren Ansatz dienen und einen Platz für einen gemeinsamen, schreibenden Zugriff mit der Möglichkeit der Typspezialisierung bieten. Wir entwickeln eine Erweiterung unseres Ansatzes, die es uns ermöglicht, mithilfe unserer ursprünglichen Technik das automatische Entpacken von primitiven geboxten Datentypen auszudrücken. Wir zeigen, dass unser Ansatz vielseitig genug ist, um auch eine andere Optimierungstechnik auszudrücken, die sich auf einzigartige Werte in einem Programmiersystem, wie beispielsweise Booleans, stützt. Darüber hinaus zeigen wir, wie erlernte Nutzungsmuster und Optimierungen über Programmausführungen hinweg wiederverwendet werden können, wodurch die Auswirkungen der Mustererkennung auf die Leistung reduziert werden.

Wir zeigen in einem Best-Case-Prototyp, dass unser Ansatzes umsetzbar ist und auch auf allgemeinere Programmiersysteme wie Racket und Squeak/Smalltalk angewendet werden kann. In mehreren Mikro-Benchmarks haben wir festgestellt, dass unser Ansatz den Speicherverbrauch effektiv reduzieren und die Ausführungsgeschwindigkeit verbessern kann.

Acknowledgments

THESES are monographs but seldom a solo endeavor. I hence want to give thanks for all the support towards the completion of this work. I thank my advisor, Robert Hirschfeld, for going above and beyond the duties of an advisor. Michael Haupt introduced me to virtual machines and set my path long before this work commenced. Michael Perscheid supported me right until my work on this topic started.

Here is the place to thank Carl Friedrich Bolz-Tereick, who laid the foundation for this work; in numerous occasions, he guided me in devising the early parts. Similarly, Alan Borning, Theo D'Hondt, Hidehiko Masuhara, and Richard P. Gabriel had significant, direction-changing impact at different stages of this work.

An important part in supporting me during this work has played the Software Architecture Group led by Robert Hirschfeld, and the HPI Research School. Without Sabine Wagner, no progress would have been possible whatsoever. I am grateful to Tim Felgentreff and Patrick Rein, who both have been a great source of inspiration and model in more than solely scientific matters. Over time, Anton Gulenko, Damien Cassou, Fabio Niephaus, Jens Lincke, Johannes Henning, Kit Kuksenok, Lars Wassermann, Lauritz Thamsen, Marcel Taeumel, Marcel Weiher, Marco Roeder, Philipp Tessenow, Stefan Ramson, Stephanie Lehmann, Tom Beckmann, Toni Mattis, Vanessa Freudenberg, and Vasily Kirilichev have helped me advance and keeping my head level.

Nevertheless, the FutureSOC Lab and the SCORE Lab, as well as the OSM group led by Andreas Polze, have supported me in the later stages of my work. I particularly want to thank Bernhard Rabe and Ayleen Oswald, as well as Daniel Richter and Max Plauth.

Keeping on pursuing this work was only possible due to the financial support of HPI. Grants of the HPI Research School on Service-Oriented Architectures and the HPI as well as the Labs that have supported me made sure I had freedom to explore, investigate, and discover.

Surely my family deserve thanks in their own right: my children, parents, and brothers, my parents in-law and, most importantly, wife Julia, whom this work is dedicated to. Thanks to Hans who shared my fate more than he knew. I want to point out my grandfathers, who enabled my interest in computers when I was twelve years old, with Klaus providing my first machine and the late Werner [39] guiding my first steps in programming.

Contents overview

1	A need for compound values	1
2	Values and data structures in virtual machines	9
3	Efficient compound values	21
4	Extended shape-based optimizations	33
5	Compound values in action	51
6	Compound values in practice	61
7	Value optimization quantified	83
8	Related work	121
9	Conclusion	133
A	Key to visual language	137
B	Comprehensive benchmark results	139
C	Source code listings	195
D	Immutable Boolean Field Elision	211

Contents

- 1 A need for compound values *1*
 - 1.1 Challenges *5*
 - 1.2 Contributions *7*
 - 1.3 Outline *7*

- 2 Values and data structures in virtual machines *9*
 - 2.1 Values *9*
 - 2.2 Data structures *13*
 - 2.3 Virtual machines *16*

- 3 Efficient compound values *21*
 - 3.1 Shapes as data structure descriptors *21*
 - 3.2 Compound value representation with shapes *22*
 - 3.3 Compressions through inlining *26*
 - 3.4 Field access *30*
 - 3.5 Configuration parameters *31*
 - 3.6 Benefits *32*

- 4 Extended shape-based optimizations *33*
 - 4.1 Restricted mutability with cells *33*
 - 4.2 Shape-guided automatic unboxing *40*
 - 4.3 Immutable boolean field elision via shapes *43*
 - 4.4 Stability for sustainable performance *46*
 - 4.5 Discussion *49*

- 5 Compound values in action *51*
 - 5.1 Best-case prototype system: Theseus *51*

5.2	Prototype implementation	54
5.3	Interaction with the JIT compiler	58
5.4	Discussion	59
6	Compound values in practice	61
6.1	Candidate systems	61
6.2	Structures in Pycket	63
6.3	RSqueak values	68
6.4	Discussion	80
7	Value optimization quantified	83
7.1	Benchmark setup	83
7.2	Shape recognition assessment	84
7.3	Derivation of optimization configuration	86
7.4	Comparative micro-benchmarks	90
7.5	Discussion of reverse as representative example	96
7.6	Limitations	119
8	Related work	121
8.1	Related concepts	121
8.2	Language-level optimization	124
8.3	Just-in-time compilers	124
8.4	Data structure implementation	125
8.5	Values in systems and languages	127
8.6	Non-values in systems and languages	130
8.7	Java and the JVM	131
9	Conclusion	133
9.1	Future work	133
9.2	Summary	133
A	Key to visual language	137

B	Comprehensive benchmark results	139
B.1	Environment	139
B.2	Parameter determination	143
B.3	Detailed results	178
C	Source code listings	195
C.1	Grammar for the language of Theseus	195
C.2	Full example of a Theseus program	198
C.3	Source code of the benchmarks	199
D	Immutable Boolean Field Elision	211
1	Introduction	211
2	Background	212
3	Structure Usage in Racket	213
4	Optimizing Records	217
5	Structures in Pycket	219
6	Evaluation	221
7	Related Work	225
8	Conclusion and Future Work	225

List of figures

- Figure 1 Illustration of shapes 22
- Figure 2 Straightforward compound value representation 23
- Figure 3 Shapes, sub-shapes, transformation rules, and history 25
- Figure 4 Transformation of compound values upon creation 27
- Figure 5 Referenced compound value reconstruction 31
- Figure 6 Cells connect compound values with mutable objects. 35
- Figure 7 Typed cells and mutable objects 37
- Figure 8 Shaped cells interconnect compound values 38
- Figure 9 Observed shape transitions, impact of caching 48
- Figure 10 Execution times for reversing lists of different lengths 85
- Figure 11 Parameter determination result 89
- Figure 12 Absolute benchmarking results 93
- Figure 13 Relative benchmarking results 94
- Figure 14 Unoptimized trace for the prototype 101
- Figure 15 Optimized trace for Theseus 103
- Figure 16 Parameters `reversen`, Theseus 144
- Figure 17 Parameters `reversen`, Pycket (optimized) 145
- Figure 18 Parameters `reversen`, RSqueak (optimized) 146
- Figure 19 Parameters `reverseE`, Theseus 147
- Figure 20 Parameters `reverseE`, Pycket (optimized) 148
- Figure 21 Parameters `reverseE`, RSqueak (optimized) 149
- Figure 22 Parameters `appendn`, Theseus 150
- Figure 23 Parameters `appendn`, Pycket (optimized) 151
- Figure 24 Parameters `appendn`, RSqueak (optimized) 152
- Figure 25 Parameters `appendE`, Theseus 153
- Figure 26 Parameters `appendE`, Pycket (optimized) 154
- Figure 27 Parameters `appendE`, RSqueak (optimized) 155

*List of
figures*

- Figure 28 Parameters map_n , Theseus 156
- Figure 29 Parameters map_n , Pycket (optimized) 157
- Figure 30 Parameters map_n , RSqueak (optimized) 158
- Figure 31 Parameters map_E , Theseus 159
- Figure 32 Parameters map_E , Pycket (optimized) 160
- Figure 33 Parameters map_E , RSqueak (optimized) 161
- Figure 34 Parameters filter_n , Theseus 162
- Figure 35 Parameters filter_n , Pycket (optimized) 163
- Figure 36 Parameters filter_n , RSqueak (optimized) 164
- Figure 37 Parameters filter_E , Theseus 165
- Figure 38 Parameters filter_E , Pycket (optimized) 166
- Figure 39 Parameters filter_E , RSqueak (optimized) 167
- Figure 40 Parameters tree_n , Theseus 168
- Figure 41 Parameters tree_n , Pycket (optimized) 169
- Figure 42 Parameters tree_n , RSqueak (optimized) 170
- Figure 43 Parameters tree_E , Theseus 171
- Figure 44 Parameters tree_E , Pycket (optimized) 172
- Figure 45 Parameters tree_E , RSqueak (optimized) 173
- Figure 46 Accumulated parameter exploration, Numeric elements 174
- Figure 47 Accumulated parameter exploration, Niladic elements 175
- Figure 48 Accumulated parameter exploration 176
- Figure 49 Accumulated parameter exploration result 177
- Figure 50 GC share results 178
- Figure 51 Results for benchmark reverse, Numeric elements 183
- Figure 52 Results for benchmark reverse, Niladic elements 184
- Figure 53 Results for benchmark append, Numeric elements 185
- Figure 54 Results for benchmark append, Niladic elements 186
- Figure 55 Results for benchmark map, Numeric elements 187
- Figure 56 Results for benchmark map, Niladic elements 188
- Figure 57 Results for benchmark filter, Numeric elements 189
- Figure 58 Results for benchmark filter, Niladic elements 190
- Figure 59 Results for benchmark tree, Numeric elements 191
- Figure 60 Results for benchmark tree, Niladic elements 192
- Figure A Racket structure field distribution 213

- Figure B Racket structure field types 216
- Figure C Structure with immutable boolean field (IBF) indicator 218
- Figure D Benchmark results (time/memory) 221
- Figure E Illustration of JIT warm-up. 226

*List of
figures*

List of tables

Table 1	Qualitative differences in the three systems	81
Table 2	Operation counts, reverse _n in Theseus	105
Table 3	Operation counts, reverse _n in Pycket	108
Table 4	Operation counts, reverse _n in RSqueak	109
Table 5	Operation counts, reverse _ε in Theseus	112
Table 6	Operation counts, reverse _ε in Pycket	114
Table 7	Operation counts, reverse _ε in RSqueak	116
Table 8	Reduction/increase factors	117
Table 9	Cache organization of the host machine	140
Table 10	Absolute execution time for all benchmarks	179
Table 11	Absolute memory consumption for all benchmarks	180
Table 12	Absolute garbage collection time for all benchmarks	181
Table 13	Relative execution time for all benchmarks	182
Table 14	Relative memory consumption for all benchmarks	182
Table 15	Garbage collection share for all benchmarks	193
Table A	Static analysis results.	214

List of algorithms

- Algorithm 1 Shape and field determination during creation 28
- Algorithm 2 Adaption of shape and field determination to cells 36
- Algorithm 3 Adaption of shape and field determination to unboxing 42
- Algorithm 4 Adaption of shape and field determination to IBFE 45

List of listings

- Listing 1 Racket structures 15
- Listing 2 Smalltalk hybrid object 16
- Listing 3 Example of accessing *cons* cells in Theseus 54
- Listing 4 Example program for Theseus with a map function 54
- Listing 5 Merging and inlining in Theseus 55
- Listing 6 Shape integration and storage selection in Theseus 56
- Listing 7 *Cons* with structures 64
- Listing 8 Integration of optimization in Pycket 66
- Listing 9 Cells in Pycket with shapes 67
- Listing 10 Pycket eq? logic 68
- Listing 11 Integration of compound values in RSqueak 70
- Listing 12 Integration of optimization in RSqueak 71
- Listing 13 RSqueak primitives for compound values 72
- Listing 14 Smalltalk primitives for compound values 73
- Listing 15 Squeak compound value vector 74
- Listing 16 Squeak compound value *cons* lists 76
- Listing 17 Squeak compound value *cons* lists terminator 77
- Listing 18 Collect with Squeak compound value *cons* list. 77
- Listing 19 Map with Squeak compound value *cons* list. 77
- Listing 20 reverse in Theseus 96
- Listing 21 reverse in Racket 97
- Listing 22 reverse in Squeak 98
- Listing 23 Trace for reverse_n in Theseus (not optimized) 100
- Listing 24 Trace for reverse_n in Theseus 102
- Listing 25 Trace for reverse_n in Pycket (original) 106
- Listing 26 Trace for reverse_n in Pycket (optimized) 107
- Listing 27 Trace for reverse_n in RSqueak (original) 110

- Listing 28 Trace for reverse_n in RSqueak (optimized) 111
- Listing 29 Trace for reverse_E in Theseus (not optimized) 114
- Listing 30 Trace for reverse_E in Theseus 114
- Listing 31 Trace for reverse_E in Pycket (original) 115
- Listing 32 Trace for reverse_E in Pycket (optimized) 115
- Listing 33 Trace for reverse_E in RSqueak (original) 118
- Listing 34 Trace for reverse_E in RSqueak (optimized) 119

*List of
listings*

List of abbreviations

AOT	ahead-of-time
API	application programming interface
CEK	control, environment, and continuation
CIL	Common Intermediate Language
COP	Context-oriented Programming
GC	garbage collector
IBF	immutable boolean field
IBFE	immutable boolean field elision
IDE	integrated development environment
JEP	JDK Enhancement Proposal
JIT	just-in-time
JVM	Java Virtual Machine
MRE	managed runtime environment
OOP	object-oriented programming
PIC	polymorphic inline cache
RTD	record-type descriptor
SSA	static single assignment
VM	virtual machine

1 A need for compound values

Not many **virtual machine (vm)**-based programming systems support compound values, but doing so would be beneficial. The notion of values as “abstractions, and hence atemporal, unchangeable, and non-instantiated” [70] — and thus immutable, identity-less entities — carries over from mathematics and is often contrasted with objects that “correspond to real world entities, and hence exist in time, are changeable, have state, are instantiated, and can be created, destroyed, and shared” [70]. Restricting the possibility to change objects, that is, introducing immutability to a programming system, has been shown to be beneficial in certain areas:

Immutability information is useful in many software engineering tasks, including modeling [...], verification [...], compile- and run-time optimizations [...], program transformations such as refactoring [...], test input generation [...], regression oracle creation [...], invariant detection [...], specification mining [...], and program comprehension [...]. [98]

Compound values as a concept in the above sense can already provide this immutability. Further, the concept of object identity does not apply to compound values.

Systems benefit from values The idea of numbers being values in programming languages is widely accepted and typically taken for granted. Numbers, especially those that can fit into processor registers, exhibit the nature of values exactly. However, there are other abstractions outside of computing that present as values, but which are not universally found as values in programming languages. In most programming languages, other — even purely mathematical — numeric values, such as complex numbers or fractions, are rarely

treated the same way register-fitting integers are. Rather, they have to be instantiated, have temporal aspects, and are subject to questions of identity.

This is at least a source of confusion for developers, as exemplified by a recurring series of questions on a relevant question-and-answer website regarding such effects in Python [54, 61, 96] (see left). Also, it opens up a broader range of questions regarding equality checks [79].

Moreover, other abstractions, that in fact *are* values, but exhibit a degree of compositeness, are rarely found in general purpose programming systems. These abstractions typically lack “valueness” altogether, as they are commonly represented by allocated, lifetime-bound, and mutable object of some sort. These include mathematical concepts that are not naturally or easily representable by integral numbers — or machine words for that matter, such as points, complex matrices, or matrix-represented tensors. Similarly, physical quantities that comprise these values, such as impedance, acceleration, or points in space, cannot be as easily represented as non-composite quantities such as length, time, or mass. However, most execution

environments, *VMs*, or interpreters are not aware of the value-like nature of these *compound values*. For example, in a pristine Squeak/Smalltalk 5.2 system, there are about 8300 instances of the class Point, where around 3700 of which contain unique coordinates (see left). While clearly being compound values, the mere question of “how many points are there right now” shows that lifetime can be observed. Moreover, the fact that the points are by and large not unique shows that identity can be observed, too. That means they *are* not factually values, at least from a *VM* point of view. That being said, points in a Squeak/Smalltalk system are *values by convention*: they are expected to never be mutated or compared by identity.

Programming systems with predominantly functional character typically comprise type systems that allow compound values to be expressed [70], for example, algebraic data types in the ML family of languages [69, 73]. That said, a more broad availability of compound values, particularly for *VM*-based programming systems, can be beneficial to express certain programs more clearly.

Python 3

```
>>> x = 10 ** 100 + 1
>>> y = 10 ** 100 + 1
>>> x is y
False
>>> a = 100 + 1
>>> b = 100 + 1
>>> a is b
True
>>> x = 3+4j
>>> y = 3+4j
>>> x is y
False
```

"Squeak 5.2"

```
Smalltalk garbageCollect.
x := Point allInstances.
{ x size. x asSet size }
"=> #(8298 3698)"
```


Just-in-time compilers benefit from values In systems that use **VMS** to execute programs, **just-in-time (JIT)** compilers are often used to achieve high performance without the need to optimize or transform programs before their execution. To do that, **JIT** compilers typically build upon profiling information collected while a program is executed in a preceding stage to then apply program optimization techniques. This includes both typical “compiler optimizations” as also found in **ahead-of-time (AOT)** compilers as well as certain optimizations that draw conclusions from run-time profiling data. In both cases, however, **JIT** compilers often speculate on the state of certain things—be it variables, call sites, or function arguments, to name a few—to *not* do what is done only seldom and quickly do what is done excessively often. Accordingly, things that are stable, as in they do not change or only change rarely, act catalytically to the **JIT** compiler. That is why determining constant parts of programs is a large part of many optimization techniques.

A large body of techniques to deduce factual constant-ness from nominal variability has been evolving for as long as compilers have been optimizing programs [78]. To name a few representative optimizations, *constant folding* or *constant propagation* [66] builds on the fact that some variable parts are actually assigned constants, for example, a variable initialized with a literal number. Whenever such a variable is subsequently used in a program, a compiler can replace the occurrence of the variable with its original, constant content. The constant is *propagated* throughout the program. Similarly, *escape analysis*—a special form of lifetime analysis [101]—builds on the lifetime of objects in a broad sense. For variable parts in programs, it often has to be tracked how many locations *share* the variable, because when such a part is actually changed, all locations shall observe the change. In these (ubiquitous) cases, programs typically allocate memory all such locations share and can observe change in. However, when a variable is only possibly shared between locations, but a compiler can infer that the variable does in fact not “outlive” a certain, limited scope, an allocation for sharing can be avoided. Subsequently, the variable is often found to be effectively constant. This, in turn, makes it possible to employ constant folding, as above. For **JIT** compilers, this can be of particular interest, as they can incorporate run-time data in their reasoning, and thus find constants that **AOT** compilers cannot detect. Other techniques in the same

vein include allocation removal [12], load-store forwarding, or even common subexpression elimination [29], to name a few. Using these techniques, JIT compilers allow programs with nominally variable parts to execute quickly when these parts are actually constant.

However, if the source of operation, that is, the program already has stationary parts, this job is easier for the JIT compiler. Compound values are especially suited here. In a way, they are a sweet spot of JIT compiler optimization, because just like other constant data, they do not change, and just like other compound data structures, can idiomatically express more than scalar or numeric information. Contrary to other data structures, the constant-ness of compound values is inherent, and there is no need to decide, whether or not to make parts of or whole data structures immutable.

A more broad availability of compound values, particularly for programming systems with JIT compilers, can be beneficial to improve the performance characteristics of certain programs.

**

Compound values present room for potential optimizations in VM-based programming systems. As more functional systems and their type systems show, the properties that compound values exhibit can be employed to increase a program's or system's efficiency. Therefore, assuming that mutability, identity, and lifetime of values cannot easily be observed in a programming system, VMs are free to represent them as they please. This is already true for "small" numbers, floating point numbers, or small data structures in certain systems where, for example, integers are represented as tagged pointers. Extending this to less trivially structured values can allow VMs to apply a range of optimizations not available when programs are able to observe any non-value characteristics of compound values, such as speculations on the constant-ness of values during constant propagation.

*A need for
compound
values*

1.1 CHALLENGES

Adopting compound values for programming systems with VMs in a memory and execution time efficient way faces certain challenges.

All VM-based languages and systems provide a certain amount of standard data structures to developers. Some of them are provided at the VM-level *from inside* the VM, and others at the language-level *atop* the VM. The reasons to choose one level or the other are diverse and have an impact on how data structures can be perceived by developers. As an example, in Smalltalk systems, the VM essentially only provides a notion of objects that can have a fixed number of fields — which is the same for all objects of a class — and a variable number of fields — which can differ on a per-object basis — or both [51]. That way, an array structure can be used at the language level via an object with just variable fields. However, all other data structures in Smalltalk, such as ordered collections or dictionaries/maps, are provided at the language level and eventually implemented in terms of the fixed/variable-fields object of the VM. This approach minimizes the interface between language and VM, which is valuable for modularity and modifications. However, when compound values are to be provided in such an environment, optimizations that could be applied to compound values can become noticeable to developers. While this can be helpful to understand such an optimization, it might impede the understanding of the uses of compound values. This means that, when optimizations are applied to compound values, the language's and developers' view on these should nonetheless reflect the program code as written in the first place.

Challenges

CHALLENGE 1 Maintaining the original structure of compound values both in the presence and absence of optimizations applied to them

The representation of data structures, and hence compound values, influences their performance; also, the same data structure might be used for different tasks or workloads. Assuming that a variety of workloads make use of the same data structure at run-time, it is necessary to provide a way of dynamically applying possibly different optimizations to the same kind of data structure. This is increasingly important the more general the programs are that run on

a *VM*, peaking in *VMS* or language implementations running stacked on top of another *VM*. For example, JRuby [81] — a Ruby implementation — and Jython [62] — a Python implementation — both are implemented on top of the Java Virtual Machine (*JVM*) and cannot predict statically how *their* data structures are going to be used. Consequently, any data structure optimization has no information on that, either. Language implementation frameworks such as RPython [2] and Graal/Truffle [117] try to provide language implementers with more means of communication with the underlying implementation. Ultimately, a definitive choice of what optimizations to apply should incorporate the way data structures are actually used at run-time.

CHALLENGE 2 *Anticipating the dynamic usage patterns of compound values at run-time*

The most predominant *VM* environments in use at the time of writing are object-oriented systems or systems that at least support object-oriented programming paradigms at the language-level, among them the *JVM* [85], the .NET Common Intermediate Language (*CIL*) [38], but also the implementations of Ruby, Python, PHP, or Smalltalk, to name a few. Most of these support objects bearing the familiar identity–behavior–state triplet, and typically all of these are *observable* by developers; state is typically *mutable*. However, observable identity and mutable state are incompatible with compound values, when taken strictly, but hard to get rid of. For example, there have been numerous attempts to add immutability to systems such as the ones mentioned above. An immutability-in-languages survey [28], found, that only few of the surveyed languages or systems support the notion of immutability to an extent that it would cancel out the incompatibility of mutable state with compound values, which — in the survey’s language — require object & class-based, transitive, enforced immutability. Similarly challenging are the notions of identity (objects do have them, compound values do not) or lifetime (objects can be created or destroyed, compound values *are*). To facilitate compound values in *VMS*, the observability of these effects needs to be restricted for the entities representing compound values, which, in turn, requires compromises for interaction with non-values.

CHALLENGE 3 *Reducing the observability of effects in virtual machines that are incompatible with compound values*

These primary challenges form the environmental constraints in which compound value optimizations in `VMS` are to be investigated in this work.

1.2 CONTRIBUTIONS

Contributions

This work makes the following main contributions:

- We propose an approach for finding patterns in compound value usage at runtime and a data structure descriptor variant that incorporates these patterns.
- We present a compressed layout for compound values that makes use of those patterns to store compound values more efficiently.
- We provide four extensions to our approach that can broaden its applicability to other use cases.
- We report on the performance of micro-benchmarks for a small prototype language, a Racket implementation, and a Squeak implementation.

Parts of this work have already been published in earlier versions at other venues, namely parts of [chapter 3](#) have appeared in SAC/OOPS 2015 [89] and Science of Computer Programming [88], in different versions. All artifacts and measurement data are publicly available [86, 87, 90, 93].

1.3 OUTLINE

This work is structured as follows:

[Chapter 2](#) provides background information on `virtual machines`, their JIT compilers, and implementations, as well as data structures and compound values in general.

[Chapter 3](#) introduces our approach to compound value optimization in `VMS`. For that, it proposes *shapes* as data structure descriptors, a technique for

recognizing data structure usage patterns at run-time, and a compact representation of compound values within a VM.

Chapter 4 offers four extensions to the initial approach. Cells provide a way to include a restricted notion of mutability when using compound values. Shapes can be employed to more efficiently represent certain data types while eliminating the need to manually introduce special representations for them. With shapes, it is possible to avoid storing well-know constants, and to effectively provide a general form of immutable boolean field elision. The learned information from the usage pattern recognition can be represented in a way that can accelerate the start up of VMs that use our approach.

Chapter 5 demonstrates that our approach is feasible with a best-case prototype implementation. It presents a custom programming system that completely focuses on compound values and their optimization and lays out the effects that the JIT compiler has on the language implementation.

Chapter 6 carries over the learnings from the prototype to existing general purpose programming systems to qualitatively assess the real-world viability of our approach. It shows the effort necessary to integrate our approach into existing implementations.

Chapter 7 quantitatively evaluates the approach. In a benchmark setting, the implementations provided are compared to their unoptimized variants with respect to execution time and memory consumption when working on compound values.

Chapter 8 lays out related work, including optimization approaches for data structures similar to compound values, concepts of compound values in certain systems, and relevant methods of performance optimizations.

Chapter 9 summarizes the findings and gives directions for future work.

SUMMARY

Compound values need more support in programming systems. We identified three challenges for their adoption. This work presents an optimized compound value representation, four extensions, and qualitative and quantitative evaluation of the approach.

2 Values and data structures in virtual machines

*Programming is object-oriented mathematics.
Mathematics is value-oriented programming.*
(Bruce J. MacLennan)

This chapter provides fundamental concepts and background information necessary to understand our approach. We explain our understanding of values and, specifically, compound values. Moreover, we provide background on data structures and their implementation concerns that are relevant for our work. Finally, **VMS** are a foundation for numerous programming system implementations and often employ **JIT** compilers for fast execution of programs; some of their properties can be beneficial for our approach.

2.1 VALUES

We subscribe to the notion of values as “abstractions, and hence atemporal, unchangeable, and non-instantiated” [70] entities. This includes both simple values like natural numbers and compound values like geometric points. We provide a description of our understanding of these concepts. We provide brief information on two special cases, as well.

2.1.1 Simple values

The notion of *values* in programming stems from mathematics. In their simplest forms, when concerning simple arithmetics of integral numbers in a certain range, values in mathematics and programming are equivalent. Although

programming languages have to cope with a mismatch between machine-oriented representation and mathematics-oriented abstraction, in general, all values that can be directly represented by numbers are well-supported.

Small and large integral numbers are typically presented either from a perspective of the machine, from the perspective of mathematics, or both. That is, most programming languages provide integer *data types*, in the first case commonly as reflection of word sizes of a processor, in the second case, usually as an abstraction called arbitrary-precision integers. There is typically a performance trade-off between them, and often both variants are found.

This scheme is typically also found for real numbers, however, as per Cantor, with greater obstacles, because machine representation is in most cases confined to the size of processor registers, with sometimes mathematically surprising results from arithmetic operations. Both floating-point and fixed-point representations for real numbers already suffer from slight deviations from the mathematical notion of values.

Enumeration types and truth values more closely reflect mathematical values, as they can typically be directly mapped to integers. Other counterparts to mathematical values are less common in mainstream, non-applicative, general purpose programming systems.

2.1.2 Compound values

“Values are used to model abstractions.” [70] However, besides simple abstractions like integers there are compound abstractions. That is, they consist of not exactly one component. Examples for these from mathematics include complex numbers with real and imaginary part, points with one component per dimension, or fractions with numerator and denominator, but also matrices, vectors, or sequences. Another example are units of measurements, which add yet another component to compound values, as in physical quantities like time. Nevertheless, from a mathematical point of view, compound values are not different from simple values; they are still abstractions with the same properties simple value have.

However, there is one aspect to compound values that can be important for programming systems. There is no concept of identity among values as there

is, for example, among objects in the **object-oriented programming (OOP)** sense; the question whether this 17 printed here is the same as this 17 one is not meaningful. Both represent the abstraction over the integer seventeen. The same holds for compound values and their components.

Further, the kind of abstraction itself plays a role in the relation between two values. The complex number $3 + i4$ has components that are equal to the components of $(3, 4)$, a point in two-dimensional space, and the rational number $3/4$. However, these three entities do not represent the same abstraction. From an **OOP** point of view, one could say that the “class” of two entities has to match.

A compound value is a composition of other values. Thus, compound values can form the components of other values, for example, matrices of complex numbers or vectors of quaternions.

It is often possible to omit the *kind* of abstraction for a value when working with scalar values; context and presentation usually provide enough information to discern, for example, real numbers from integers. The kind of abstraction is, nevertheless, involved.

In addition to the kind of abstraction, certain compound values can be grouped by their *arity* (or cardinality), that is, the number of components. For example, there are geometric points in one-, two-, and three-dimensional space with one, two, and three components, respectively. Likewise, vectors—in the sense of members of n -dimensional vector spaces—have n components.

When working with concrete instances of compound value abstractions, we will refer to the concrete components of such a compound value as its *constituents*.

2.1.3 Niladic compound values

There can be compound values that have no components whatsoever, their arity is zero. For example, given that members of vector spaces are indeed values, the sole member of the zero-dimensional vector space certainly is a value, too. However, since the space has no dimensions, its member vector has no components.

As a consequence, the only information such compound values can convey is their “grouping”, that is, what kind of abstraction they represent.

The general usefulness of such compound values is probably limited, but under certain circumstances, they can be used to convey the information of singleton-like entities that are void of information beyond their existence, for example, the abstract idea of an empty list or set, such as *nil*.

Since the applicability of such compound value is limited, there is no common name. Nevertheless, at least two variants exist, *nullary* compound values (from binary, unary, nullary), and *niladic* compound values (from dyadic, monadic, niladic). We will use the latter variant here.

2.1.4 Simple values as special case of compound values

The idea that values convey information about the kind of abstraction they represent can be extended further.

One point of view is that “simple” values, such as integers, are just unary/monadic compound values, and the kind of abstraction they represent is “integer”. Yet, since compound values compose, it is questionable what *kind* of abstractions the components of a unary integer compound value are.

Another point of view is that “simple” values can be understood as niladic compound values. Each representative of the *kind* of abstraction is then actually its own *kind* of niladic compound value. For example, each integer can be viewed as a compound value of the kind of itself with no components, such as the niladic compound value “seventeen”. This view is actually supported in parts of literature and at least one programming system, BETA. In agreement with Hoare [57] BETA’s authors state, that “a value, like four, is an abstraction over all collections of four objects” [67]. Therefore, in this programming system, each number is at the same level of abstraction as any class in that system.

However, in most practical circumstances, it is not as helpful to elevate each simple value to abstractions in their own right as it is to treat them as non-composite leaves.

2.2 DATA STRUCTURES

Data structures and their implementation are a key part of virtually all programming systems. Data structures are entities to store and manage assortments of data. For the purpose of this work, we differentiate homogeneous structures and heterogeneous structures.

The way programming systems manage metadata varies, particularly in `VMs`. Since there is frequently the need to represent metadata differently than the actual data, we briefly describe data structure descriptors as an abstraction of data structure metadata management.

This description also serves as abridged, initial presentation of what data structures look like in the programming systems used in later parts of this work, namely Racket [45] and Squeak/Smalltalk [60].

Collections as homogeneous structures Collections are assortments of homogeneous elements, such as arrays, vectors, or similar and often exist in indexed and ordered variants. They typically do not form types. Individual collection instances may differ in size.

Records as heterogeneous structures Record data structures, or *records*, are collections of named fields of heterogeneous values. Records may form a type, instances of record types are typically of equal size — all in contrast to homogeneous data structures. Moreover records may have various additional features, which may differ between programming languages.

2.2.1 Data structure descriptors

One of the key responsibilities of programming systems, and `VMs` in particular, is to provide programs with data structures. This means that the system offers a way to query for new instances of a given kind. For example, a Java program might want to create a new *Array*, a Python program a new *dict* instance, or a Ruby program any new object, to name a few popular variants. These core functionalities are so ubiquitous that they are seldom mentioned outside of

the responsibilities of memory management and, if applicable, the **garbage collector (GC)**.

To properly manage data structures, every instance of a data structure usually has to provide some metadata besides the actual data it represents. This can be metadata specific to the individual data structure instance, such as **GC** and lifetime information, or data that is the same for a group of similar instances, such as the size of similar record structures or class information for object instances. We will refer to the holders of such metadata for data structures as *data structure descriptors*, descriptors for short.

There are some common patterns and constraints in implementing data structure descriptors. Typically, the memory management or a **GC** need to know the size of individual data structure instances. Also, to be able to access the proper contents of a data structure instance, a descriptor has to provide the abstract kind of the data structure as quickly as possible. Likewise, it is often necessary to quickly determine what operations are available on a certain data structure, in particular for **OOP**. All this information usually needs to be conveyed with as little overhead as possible, lest the metadata outweighs the concrete data structure instance.

In class-based **OOP**, classes often provide all necessary information about size and lookup data. Hence, the implementation-level representation of a class is often used as a descriptor for all of its instances. Depending on language semantics, these descriptors can be hidden from the language level, can be available via reflection and reification as in Java, or are first class citizens of the language itself as in Smalltalk. In fact, the language-level class objects *are* the **VM**-level descriptors in many Smalltalk implementations [51, 60, 74]. These descriptors are suitable as long as the layout of object cannot change. However, in prototypical languages or languages that allow objects to be extended arbitrarily, such descriptors often become a performance bottleneck. Accordingly, these languages typically have different descriptors: Self groups its objects into *families* which are described by a **VM**-level map [24]. The V8 implementation of JavaScript/ECMAScript uses *hidden classes* [3] for a similar purpose, but adapted for typical language usage patterns.

2.2.2 Racket structures as records

We briefly introduce Racket's records as an example for what form records can take in a concrete language, since later parts of this work will make use of this feature. Racket [45] is a dynamically typed, multi-paradigm programming language from the Scheme family [107]. It differs from Scheme in certain aspects,

such as immutable-by-default lists, support for *design by contract* [77], and an advanced record data structure concept called *structures* with features beyond the mere ability to store values in their fields. Racket structure types can form *hierarchies*, supporting inheritance. Structures in Racket are *immutable* by default, but can be explicitly declared to be partly or fully mutable. Structure type *properties* allow to store arbitrary data inside the structure type. Typically, properties are used for procedures that work on a structure's field values. Certain properties can be used to make structure instances *callable*; these structures can then act as procedures. We include a brief example of a Racket structure in [listing 1](#), which includes examples of hierarchy, and optional mutability among others. With a definition of a structure type (lines 2 and 4), the name, parent structure type, and fields are determined, and procedures to create (lines 3 and 10), query (lines 8 to 12), and maybe modify (line 13) structures of that type.

LISTING 1: Racket structures may be mutable and form hierarchies

```
(struct person (name))
(define customer (person "Sam"))
(struct employee person
  (position [office #:mutable]))
(define worker
  (employee "Sally" "Developer" 'branch))
(person? 0) ; -> #f
(person? customer) ; -> #t
(person? worker) ; -> #t
(employee? customer) ; -> #f
(employee? worker) ; -> #t
(set-employee-office! worker 'main)
(employee-office worker) ; -> 'main
```

2.2.3 Smalltalk objects as hybrid structures

Objects are at the heart of object-oriented languages and the choice of how to represent them in memory is crucial for the performance of a language implementation [5, 80, 110]. The standard way of representing objects involves an indirection for references to other objects, for example by using direct pointers or object tables. Typical best practices of object-oriented modeling and design — such as delegation or the composite design pattern — have an influence on performance when using such representations. Every additional indirection between delegators and delegates or composites and their parts

has the overhead of a new object. This includes memory consumption, but also execution time to navigate the referenced objects.

LISTING 2: Smalltalk objects with both homogeneous and heterogeneous parts

```
Object subclass: #LEDStrip
  instanceVariableNames: 'voltage'
  LEDStrip class
  leds: aNumber
    ^self new: aNumber withAll: Color black
  LEDStrip
  voltage: aNumber
    voltage := aNumber.
  at: aNumber r: red g: green b: blue
    self
      at: aNumber
      put: (Color r: red g: green b: blue).
  dim
    [self isDark] whileFalse:
      [self voltage: self voltage - 0.1.
       self darkenLEDs].
  darkenLEDs
    1 to: self size do:
      [:i | | color |
       color := (self at: i) darker.
       self at: i put: color].
```

Smalltalk objects support single inheritance, provide a string encapsulation of their state, and virtually fully control the set of operations they support. Objects in Smalltalk have a special property. They can have heterogeneous parts like records and at the same time homogeneous parts like arrays. While objects are usually used in a heterogeneous way, in particular when representing domain objects, collections are also objects and accordingly used in a homogeneous way. It is typical to compose both kinds of objects, but it is also possible to have both kinds in just one object. In listing 2, we show a small class for objects modeling an LED strip. Initialized with the number of LEDs, this number determines the final size of the object. The voltage instance variable is not part of the homogeneous object part. Both object parts are equally simple to access, use, and modify, but still encapsulated in methods. That way, interesting use cases can be supported.

2.3 VIRTUAL MACHINES

We use the term **virtual machine (VM)** to collectively refer to **managed runtime environments (MRES)**, **VMS**, and execution environments in general. Moreover, we distinguish between a **VM level** and a **language level**. The former denotes the implementation blocks and concepts used to provide a language implementation within a **VM**—this is sometimes called *host* and the programming language used to implement on that level the *host language*. The latter level denotes the concepts provided by a **VM** or **VM-level** program to application developers—accordingly it is sometimes called *guest* and the programming language or system provided by the **VM** *guest language*. Note that a guest

language might not be a programming language as such but rather an agreed-upon exchange format, such as bytecodes. Accordingly, in case of metacircular **VMS**, host language and guest language are the same. Also, for the sake of simplicity we subsume libraries, frameworks, or applications running on a **VM** — that is, at language level — under *programs*.

2.3.1 Value objects

In the variety of data structures that typical **VMS** with support for **OOP** provide, values do not fit in easily:

Abstractions [and hence, compound values] are not physical objects [...], so to deal with them they must be represented or encoded into objects. [...] Once a value has been represented as an object it acquires some of the attributes of objects. Clearly, whenever a value is to be manipulated in a computer it must be represented as the state of some physical object. [70]

Accordingly, object-oriented programming systems and languages that support compound values in any way typically provide an object-oriented view on these values. Hence, without loss of generality, in those systems “compound value” and “value object” can be used synonymously. Accordingly, the data structure descriptor for such value objects is called “value class” in class-based **OOP** systems — complementing value types as given above.

2.3.2 Identity

One of the implications of supporting compound values in **VMS** is that the already complex question of object identity [79] becomes complicated. Compound values are abstractions, but when represented in programming systems that have strict notions of identity, two values that represent the same abstraction might not be coalesced. This is less of a concern in languages with predominantly functional aspects, since the many operations on data structures create new data structures as a result. Therefore, relying on the identity of

data structures would not work. That said, the question whether two entities represent the same “thing” can and should be answered differently depending on whether the data structures at hand have object character or value character, which is the way the EGAL operation works [7]. The two languages that are used later in this work notably do *not* provide EGAL-esque comparison methods by default. Racked provides `eq?`, `eqv?`, and `equal?`, for identity, numeric equality, and user-defined equivalence, respectively. Smalltalk provides `==` and `=`, which are object identity and user-defined equality, respectively. For both system, the identity-based comparison methods cannot be changed and the user-defined comparison method often defaults to the identity one. This can hamper the adoption of compound values in both systems.

Nota bene: The question of whether two things are “the same” is a complex one not only in concrete programming systems or abstract mathematics, but also in several fields of philosophy [33]. An important question, which is related to the optimization approach, is, whether a thing remains the same when all of its parts are successively exchanged for new ones. There is no definitive answer.

2.3.3 JIT compilers

JIT compilation has become a mainstream technique for, among other reasons, accelerating the execution of programs at run-time. After its first application to Lisp in the 1960s, many other language implementations have benefited from JIT compilers — from APL, Fortran, or Smalltalk and Self [4] to more recent languages such as Java [85] or JavaScript [59].

One approach to writing JIT compilers is using *tracing* [8]. A tracing JIT compiler records the steps an interpreter takes in common execution paths such as hot loops. The obtained instruction sequence is commonly called a *trace*. This trace can on itself be optimized or transformed to machine code and used instead of the interpreter to execute the same part of that program [76] at higher speed. Tracing produces specialized instruction sequences, for example, for one path in if–then–else constructs; if execution takes a different branch later, execution switches back to use the interpreter. Tracing JIT compilers have been successfully used for optimizing native code [8] and also for efficiently executing object-oriented programs [49].

Meta-tracing [12] takes this approach one step further by observing the execution of the interpreter instead of the execution of the application program. Hence, a resulting trace is not specific to a particular application but the underlying interpreter [15, 19]. Therefore, it is not necessary for a language implementer to program an optimized language-specific **JIT** compiler but rather to provide a straightforward language-specific interpreter in RPython, a subset of Python that allows type inference. *Hints* to the meta-tracing **JIT** enable fine-tuning of the resulting **JIT** compiler [14]. RPython's tracing **JIT** compiler also contains a very powerful escape analysis [13], which is an important building block for the optimization described in this paper. Meta-tracing has been most prominently applied to Python with PyPy [99].

SUMMARY

We laid out our notion of values, compound values, and niladic compound values. We described homogeneous and heterogeneous data structures, and gave examples for both. Further we introduced the concept and concrete variants of data structure descriptors. We gave a synopsis of **vms** and potential issues when providing compound values in **vms**.

3 Efficient compound values

*Any problem in computer science can be solved with
another level of indirection* (David J. Wheeler)
(Butler W. Lampson)

This chapter proposes a memory efficient data structure representation that is especially suited for compound values. Employing a specialized data structure descriptor, we present an approach to inline compound values within other compound values that reference them, thus using less memory. We suggest a simple technique to access the contents of inlined compound values and explain its contribution to an accelerated execution in the presence of certain `JIT` compilers.

The key idea of our optimization is to look for common patterns in the object graph at run-time. If a frequently appearing pattern is identified, we introduce an abbreviated, inlined form of this pattern. Newly created compound values that exhibit this pattern use the abbreviated form to save memory. For this chapter, we assume `VMS` with object-oriented capabilities, and without loss of generality use the terms “compound value” and “value object” interchangeably (cf. [section 2.1](#)).

3.1 SHAPES AS DATA STRUCTURE DESCRIPTORS

When providing data structures in a `VM` for the language level, it is usually necessary to keep track of certain information. Depending on language semantics, this typically achieved by equipping concrete instances of a data structure with a descriptor that provides abstract information, for example with class pointers, hidden classes, or maps (cf. [section 2.2.1](#)). Similar concrete instances

share a descriptor as they are the same in an abstract sense, such as in concrete instances of a class which share that same class.

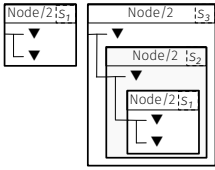


FIGURE 1: Illustration of a default shape for a binary Node (left), and a shape for an optimized variant of such a Node (right). Excerpt from figure 3.

We introduce the *shape*, an extended data structure descriptor that groups structurally similar concrete values. Just like ordinary descriptors, a shape describes the abstract content of a concrete value, including field layout and possible behavior. For value classes, shapes can wrap a traditional class pointer or vtable, and provide additional information. This includes the *arity* of a compound value — that is, the number of fields it has. Moreover, when a compound value refers to other compound values, the shape contains information about the shapes of each referred-to *constituent*. Thus, shapes can be structurally recursive and form a composite (cf. figure 1). However, in the base case the shape has no information beyond the mere existence of a field. This is similar to traditional descriptors and called the *default shape* for that value class (for example, cf. left of figure 1). The extended case is described below.

3.2 COMPOUND VALUE REPRESENTATION WITH SHAPES

A straightforward representation for a given object in memory is a chunk of memory that stores a reference to the object's descriptor (for example, its class) first, followed by references for each of its fields. We call the latter the *storage* of the compound value. The contents of this storage respectively refer to other compound values, forming its *constituents*. An example of this straightforward representation can be seen in figure 2, which shows a linked list and a tree structure.

However, when common patterns in the reference graph of compound values exist, the shape allows to convey that information without indirections and provide inlined variants of a compound value cluster. That is, instead of storing references to a sub-object, the sub-object's fields are inlined into the referencing object's fields. This saves the pointer from the outer object to the inlined one, the overhead of maintaining a separate object and the reference to the inlined object's class. This inlining is done recursively, if possible. During

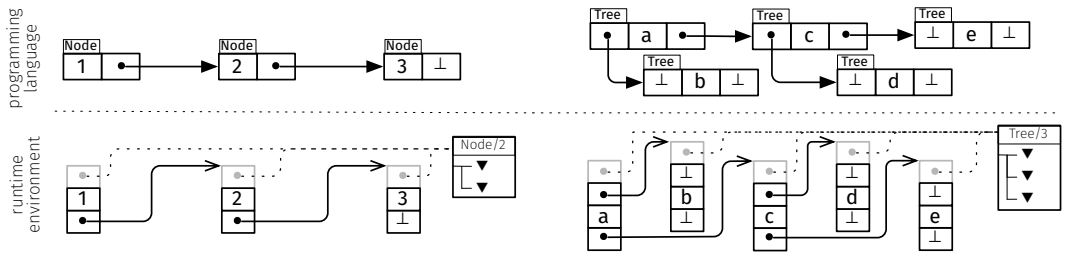


FIGURE 2: Straightforward compound value representation for a linked list and a tree. Top: the language view; bottom: execution environment view with storage and shape

Compound value representation with shapes

the inlining process, we need to maintain certain meta-information to keep track of which fields belong to which level of an inlined compound value and in order to remember the classes of the inlined objects of the object. This information is provided by a compound value's shape: the shape references a sub-object's shape directly if that sub-object was inlined into its outer object. If no inlining occurs, we still give the object the default shape, which also conveys the absence of inlining.

It is important to not just arbitrarily inline objects but to do so only for frequent combinations of outer classes and inner classes. Since the shape needs memory too, introducing shapes that are solely used by a single object would actually waste memory.

3.2.1 Shapes and inlining

Shapes can be nested; they consist of sub-shapes for each field in the storage of a compound value. A special, flat shape denotes unaltered access to object fields (*direct access shape*, in all figures) and termination of shape nesting. It conveys no more information than that a field exists and may contain data. Compound values with these shapes are treated as black boxes, for example scalar data or unoptimized objects that are stored directly (cf. default shapes above). This is depicted in the bottom part of figure 2; all three nodes in the list share the same shape, which denotes that each node consists of two references

with *direct access* shapes. Likewise, the nodes of the tree in that figure share a shape just like the first one, but with three references.

As long as no optimization has taken place, a compound value refers to the *default shape* of its value class that solely consist of *direct access* sub-shapes. The shapes in figure 2 are the default shapes for their compound values. Initially, all compound values use a default shape. To reach a state where more complex shapes can be used, our approach depends on auxiliary data.

To guide the overall optimization process, we keep track of all shapes that we encounter during object creation. That way, we create a histogram of all shapes used in the fields of compound values. We explain this profiling data, which we call the *history*, in section 3.2.2.

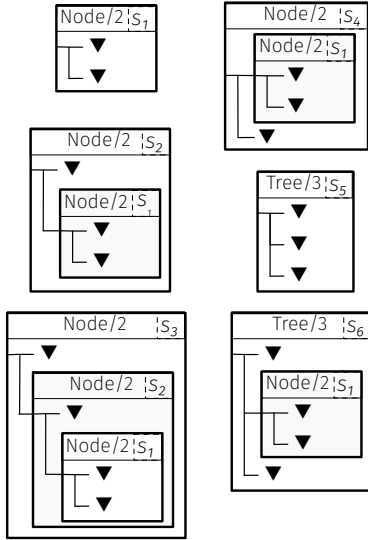
Based on the history profiles, we determine the fields in a compound value where inlining referenced compound values could be worthwhile. We infer new shapes for value objects with certain referenced compound values inlined, and record a transition from the old to the new shape. We call this process *shape recognition* and explain it in section 3.2.3.

We collect all results from the shape recognition in a table that we call the *transformation rules*. We explain its structure briefly in section 3.2.4.

3.2.2 History

The *history* is a table that maintains count of how often certain sub-shapes are found in the fields of new compound values. It is a histogram of all sub-shapes and simple to maintain, because the immutability of compound values makes modifications to this table only necessary during value object creation. At this point, all constituents of the new compound value are available and we can count the occurrences of *sub-shapes* a specific positions in the compound value. For example, the history table in figure 3 shows that for the shape s_1 , there were 17 observations of the shape s_1 as sub-shape in position 1, while shape s_2 has been observed 5 times in that position.

The most important operation on the history table is updating the count of a [shape \times position \times sub-shape]–entry, besides initializing it to 1 on the first encounter. It is possible to remove a history entry after it had been used for creating a transformation rule, if desired.



transformation rules			
shape × position × sub-shape → substitution			
s_1	I	s_1	s_2
s_1	I	s_2	s_3
s_1	O	s_1	s_4
s_5	I	s_1	s_6

history			
shape	position	sub-shape	occurrences
s_1	O	s_1	17
s_1	I	s_1	28
s_1	I	s_2	5
s_2	I	s_1	4
s_2	2	s_1	19
s_2	2	s_2	8
s_3	O	s_1	1
s_3	I	s_1	3
s_3	I	s_2	1
s_4	I	s_1	1
s_5	I	s_1	13
			...

FIGURE 3: Left: *Shapes* comprise a class reference, an arity, and a graph of sub-shapes. Right: *Transformation rules* describe substitutions for shapes which are consulted during the inlining process; *history* contains a histogram of all sub-shapes encountered at a certain position in a certain shape collected during all compound value creation. (Key in [appendix A](#))

3.2.3 Shape recognition

During the creation of a compound value we first update the shape history table and then check the counters associated with the shapes of the object's fields. Whenever one of these counters exceeds a preset threshold, we create a new shape that combines the compound value's current shape with the sub-shape that exceeded the threshold. In this new shape, we replace the *direct access* sub-shape at the position where the threshold was reached with the sub-shape found in the history entry. We then create a new transformation rule that maps from the old shape, the position, and the sub-shape at that position to the newly created shape.

*Efficient
compound
values*

Considering [figure 3](#) as example, shape s_2 would be the result of turning the history entry $(s_1, 1, s_1, 17)$ into the transformation rule $(s_1, 1, s_1) \mapsto s_2$.

3.2.4 Transformation rules

We maintain the set of all transformation rules as a lookup table that is used during compound value creation. This table is only ever updated during shape recognition and typically, rules are never removed from it. However, it is usually much smaller than the history table. An example transformation rule table is shown in the top right of [figure 3](#). Conceptually, we consider both history and transformation rules to be tables. However, depending on implementation circumstances, it may be advisable to merge them into one table or make them part of a shape object.

3.3 COMPRESSIONS THROUGH INLINING

The information of what shapes occur often and which shape transformations to use can be applied at run-time to create compound values in a compressed representation. The process of creating such a compressed compound value is outlined as follows. As running example, we will use the combination of the primitive datum "1" with a linked list into a new linked list shown in [figure 4](#), using the shapes and transformation rules shown in [figure 3](#).

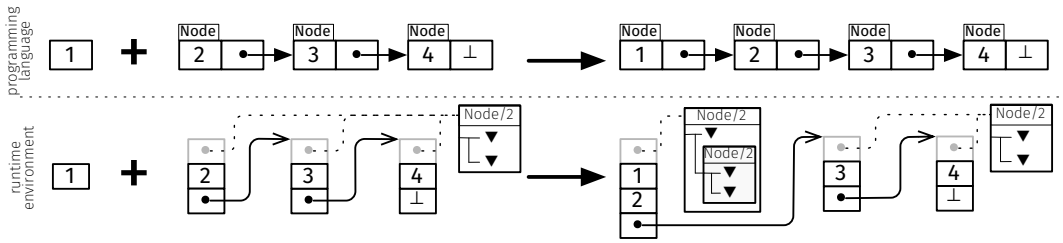


FIGURE 4: When creating a new node compound value that should contain “1” and the list “Node/2[2, Node/2[3, Node/2[4, ⊥]]”, a new compound value that merges the “1” with the “2” object and a different shape is created instead.

Compressions through inlining

First, it is only necessary to consider compression when creating new compound values. Since they are immutable, there is no need to consider compression on mutation. Therefore, the inlining process starts with the following two components:

1. the value class of the object that is to be created, and
2. the elements that should constitute said object’s new fields.

In the given example, the class is Node/2 and the new fields are “1” and a Node/2 compound value (“Node/2[2, ...]”). As pointed out earlier, every value class has an associated default shape equivalent to a straightforward representation. In the case of the class Node/2, this default shape corresponds to shape s_1 in figure 3.

With the default shape and fields, the inlining algorithm as specified in algorithm 1 can now commence. In our example, the initial shape s provided as input to the algorithm is the default shape s_1 and the fields f are “1” and “Node/2[2, ...]”.

We now iterate over the fields (line 3) and consider each new field f_i separately. For that, we look at the sub-shape s_i of the new field f_i and try to look up a substitute shape s' (line 5). If we have no substitution, for example because none has been recorded yet or the new field f_i is primitive data, the shape is not substituted and we continue with the next element. However, if we find a substitute (line 6), we replace the compound value f_i with a copy of its *storage*

ALGORITHM 1: Determining shape and fields of a compound value during its creation. The final shape is determined based on the object's initial shape, its constituents' sub-shapes, and the transformation rules. The fields are then inlined based on the shapes resulting in each step until final shape and the final constituents have settled.

*Efficient
compound
values*

```

1  Input:  $s : Shape, f : [Value Object]$ 
2   $i \leftarrow 0$ 
3  while  $i < |f|$  do
4       $s_i \leftarrow f_i\{shape\}$ 
5       $s' \leftarrow transformations_{s_i, s_i}$  or  $s$ 
6      if  $s' \neq s$ 
7           $f \leftarrow [f_{0, \dots, i-1}, f_i\{storage\}, f_{i+1, \dots, |f|}]$ 
8           $s \leftarrow s'$ 
9          // restart with new storage
10          $i \leftarrow 0$ 
11     else
12          $i \leftarrow i + 1$ 
13     end
14 end
15 return  $s, f$ 

```

in the new fields f (line 7); the compound value f_i is now *inlined*. The new shape s' becomes the new compound value's shape s (line 8) and the inlining process is *restarted* (line 10) with the new shape and fields. That way, other transformation rules can be applied that take effect because of the new shape. Once no further transitions are found, the compound value's shape s and the current fields f are returned as the shape and storage of the new compound value (line 1).

In our example, the following happens: while iterating over the new fields f , we encounter "1" as the first field f_0 . Since this is a primitive datum, no new shape can be found and no shape change happens. The next new field f_1 to consider is "Node/2[2, ...]". The sub-shape s_1 of this value object is s_1 and we can now look up a transformation rule for $(s_1, 1, s_1)$ and find a substitution

s' , s_2 (line 5). Thus, we inline the storage of f_1 by copying it into the new fields f at position 1. The original compound value “Node/2[2, ...]” remains untouched and can still be referenced from other objects. The fields of f are now “1”, “2”, and “Node/2[3, ...]”. Furthermore, we change the shape of the new compound value to s_2 (line 8). At that point, we restart the inlining process by resetting the counter (line 10). This means, we again encounter “1” as first field f_0 and no substitution happens. Moreover, the second field f_1 is now “2”, so no substitution happens either. We continue with the third field f_2 , which is “Node/2[3, ...]”. The sub-shape of this compound value is s_1 , and since s is s_2 , we can look up a transformation rule for $(s_2, 2, s_1)$ in the table. However, no such transformation rule exists and, hence, no further inlining is possible. Since we visited all fields, the algorithm terminates and returns the compound value’s new shape s_2 and its new fields [1, 2, Node/2[3, ...]] (line 15).

During the inlining process, potentially short-lived objects might be created. This can happen when the storage of a compound value is inlined into its surrounding list of fields. Typically, a new list of correct lengths is created and the old list will no longer be referenced. In subsequent inlining steps, this new list itself may be short-lived. To retain simplicity in our approach, we refrained from introducing sophisticated mechanisms to avoid the allocation but rather rely on the capabilities of elaborate GCs and JIT compilers. We expect those allocations to happen in tight loops, but more importantly, in a very restricted scope. Hence, JIT compilers that provide good escape analysis and allocation removal should be able to completely remove all allocations during the inlining process, for example meta-tracing JIT compilers [13].

This shape inlining technique has two main advantages. First and foremost, inlined compound values take up less space than individual, inter-referenced compound values. But even more, the shape of a compound value provides structural information in a manner the meta-tracing JIT compiler can speculate on. This is crucial for optimizing field access in a compound value.

3.4 FIELD ACCESS

Efficient compound values

While optimization of data structures takes place during construction, we have to apply the reverse during “deconstruction”, that is when accessing a compound value referenced by another. This is no longer trivial, as several (formerly referenced) compound values may have been inlined into their referencing value objects. Therefore, we create new compound values whenever a reference is navigated, essentially *reconstructing* a copy of the original compound value. We use the information from the shape to identify the parts of the compound value’s storage that comprise the compound value to be reconstructed. The structural information allows a direct mapping from the language view of the data structure to the actually stored elements. In [figure 5](#), the structural information in the shape of the leftmost list implies that the first element of the storage is equivalent to the head of the language level node compound value and the remaining three storage elements are equivalent to the tail of that value object, as recorded in the shape. This explains the middle view in that figure, where both the element “1” and the rest list have been reconstructed. Likewise, in the right part, the element “2” and another rest list have been reconstructed.

Every language-level access to a field of compound value with compressed storage will result in the reconstruction of the inline compound value. However, this reconstruction is completely invisible to developers. For example, whether it is accessing the tail of a node compound value or accessing the third element of a ternary tree repeatedly, the operations at the language level remain unchanged and are not influenced by the shape inlining status of the compound values at the implementation level.

Related approaches optimize this step in a different way. For example, object inlining [114] uses inlining information to modify accessors. That way, the access to contents of inlined objects can be as simple as pointer arithmetic with offsets. This is particularly useful with mutable objects. However, this needs knowledge about all callers of methods and, depending on the case, identification of combined accessors. For example, the *head(tail(...))*-call in [figure 5](#) would have to be altered to directly access the second field of the compound

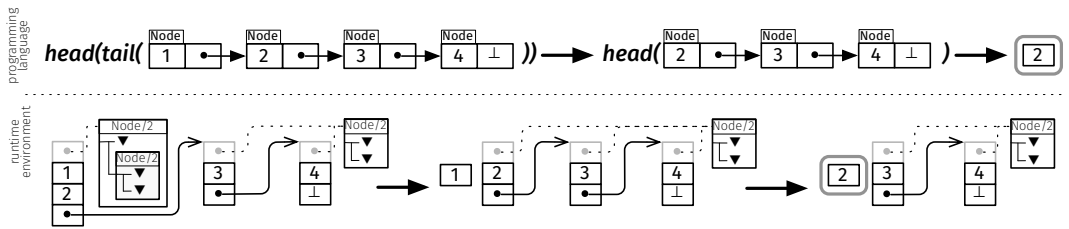


FIGURE 5: Referenced compound value reconstruction. Accessing the second item 2 of the list $l \leftarrow Node/2[1, Node/2[2, Node/2[3, Node/2[4, \perp]]]]$ by two operations $head(tail(l))$ results in two reconstructed rest lists to be created.

Configuration parameters

value. We explicitly avoid this kind of “deep” accessors and defer optimization of the access patterns to JIT compilers.

3.5 CONFIGURATION PARAMETERS

The following three parameters may influence the performance of recognition and inlining.

Maximum object size Only compound values *smaller than* this size are considered for inlining. Setting this to zero disables optimization; setting it to a very high number might result in very large inlined data structures at run-time, which might be undesirable.

Maximum shape depth The number of recursive shape occurrences per compound value is bounded by this parameter. Only objects with a shape structure whose depths is *smaller than* this value are considered for inlining. Setting this to a low value may not catch all optimizable object shapes; setting it to a very high number may lead to an excessive number of shapes at run-time should there be a lot of value objects with no fields at all.

Substitution threshold The threshold for transformation rule creation (as in section 3.2.4), when set to a zero or a very low value can lead to excessive

transformation rule creation for compound value combinations that are only rarely used. A very high number might inhibit the creation of such rules at all and practically disables our optimization.

3.6 BENEFITS

Efficient compound values

With the shape inlining approach, fewer compound values need to be created for long-lived data structures, since the references to the now-inlined compound values are elided. Combining this with compound value reconstruction and shape recognition, increasingly more memory can be saved the longer a program runs. The shapes will be tailored to fit the specific running application. That said, there may be cases where no memory can be saved, especially in programs that only work on primitive data, flat data structures, or with a high amount of sharing between data structures.

This approach can effectively increase the spatial locality of data by placing pieces of data close together when they might be accessed in temporal proximity. Furthermore, it improves spacial density, as data whose storage is dispersed prior to the application of approach can now be laid out more compactly and contiguously. This fits well with the needs and assumptions of processor architectures and operating systems which, on the one hand, operate on the granularity of cache lines and, on the other hand, pages of memory .

SUMMARY

Shapes are shared data structure descriptors for compound values. During compound value creation, shapes of constituents are recorded and transformation rules inferred. These rules are consulted and compound values are created with their constituents inlined into the referring compound value. On access, original constituent compound values are reconstructed.

4 Extended shape-based optimizations

*Any problem in computer science can be solved with
another level of indirection* (David J. Wheeler)
(Butler W. Lampson)
...except for the problem of too many levels of indirection.
(Kevlin Henney)

The presented approach to optimize the representation of compound values can already save memory and execution time, but certain use cases may warrant modifications of our approach. This chapter presents four extensions to the base approach: handling not completely immutable data in [section 4.1](#), represent simple values in an optimized fashion in [section 4.2](#), automatically omit certain invariant values in [section 4.3](#), and re-using cross-run optimization profiles in [section 4.4](#).

4.1 RESTRICTED MUTABILITY WITH CELLS

Compound values *per se* do not subscribe to the notion of mutability¹. However, using compound values as a means to represent not inherently value-like data structures for performance reasons can be an interesting trade-off (cf. [Just-in-time compilers benefit from values](#)), that makes it possible to think of mutable constituents to a compound value.

A common technique that aids the implementation of lexical scoping stemming from the Lisp/Scheme world, typically called *cell* or *handles*, can provide

¹cf. [chapter 1](#): “[V]alues are abstractions, and hence atemporal, unchangeable, and non-instantiated” [70].

a necessary indirection [102]. This basic concept has been widely applied, in interpreters and compilers alike, as well as memory managers and garbage collectors. Effectively, bindings from names to values and/or functions are represented using a one-element memory location, and any location in the program that refers to a name — such as a variable or a function — rather point to that binding instead of the value/function directly. That way, when changing data stored “behind” a name, it is not necessary to chase all locations to enact the change but rather only affect the binding. These bindings also have been called *cells* [47], which is adopted here. The Racket language even provides simple language-level cells under the name “box”.

A cell is a mere stand-in for the data it holds. All operations on a cell are actually subjected to the data it holds. Most importantly, the data a cell holds can be changed, while the cell appears to be immutable to the data structure it is put into. This is a common *one-level-of-indirection* engineering technique present in languages that support data structures with optionally mutable fields. The typical trade-off is between an increase in complexity and execution time when accessing the data behind the cell, and the expected gain of a fully immutable data structure, as shown in [chapter 3](#) and [chapter 7](#).

When applied to our approach, cells take the place of the content they refer to in the constituents of other compound values, as shown in [figure 6](#). There can be an arbitrary number of referrers to a cell and, indirectly, to an object encapsulated by it. This reflects the possibly many *vm*-level instances of compound values that could refer to the mutable object. Changes to the *content* of a cell are hence visible to all referring compound values with no need of chasing referrers to pin down necessary changes. However, cells are never inlined into other compound values.

4.1.1 Changes to the base approach

Since the default inlining step of our compound values optimization is based on the compositional nature of those values, it either expects a simple value or another composite one, the latter of which can be itself subject to the inlining step. This dichotomy no longer holds in the presence of cells, as they are, in fact possibly compound but — with regards to inlining — treated in the same way

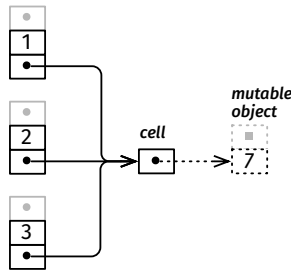


FIGURE 6: Cells connect compound values with mutable objects.

*Restricted
mutability
with cells*

as simple values and are *not* to be inlined even when their shape hits the recognition threshold. Hence, it is necessary to introduce a special kind of shape that reflects the never-to-be-inlined nature of cells. With such a `CellShape`, the algorithm can be adapted to also allow for cells, either by changing the two-way shape-switch to a three-way switch or by deferring the responsibility of what happens upon inlining to the shape itself, that is, applying plain polymorphism. Refer to [algorithm 2](#) for a possible adaption of [algorithm 1](#) for cell shapes.

Moreover, every access to a compound value's constituents has to be augmented by a check for eventual cells so that the cell's actual content can be accessed instead. That being said, the way access to inlined compound values works suggests an implementation that can easily anticipate that. Access to a compound value's constituent can be patched through its sub-shape representation and dispatching on that. The base cases, direct access and inlined compound shape, remain unaltered, and additionally, a cell shape instructs to reach into the compound value's storage and *in addition* dereference one more time.

4.1.2 Impact

Cells present a comparatively easy way to re-introduce mutability for compound values in case there are strong arguments for it and few instances of it. The changes to the base approach are limited and, depending on the way implemented, can be handled in just one more case for merging and access,

ALGORITHM 2: Adaption of shape and field determination to cells. There is no lookup of transformation rules for cells, as they must inhibit inlining of their possibly mutable content.

```

1  Input:  $s : Shape, f : [Value Object]$ 
2   $i \leftarrow 0$ 
3  while  $i < |f|$  do
4       $s_i \leftarrow f_i\{shape\}$ 
5      if  $s_i = s_{Cell}$ 
6           $s' \leftarrow s$  // do not inline cells
7      else
8           $s' \leftarrow transformations_{s_i, s_i}$  or  $s$ 
9      end
10     if  $s' \neq s$ 
11          $f \leftarrow [f_0, \dots, i-1, f_i\{storage\}, f_{i+1}, \dots, |f|]$ 
12          $s \leftarrow s'$ 
13         // restart with new storage
14          $i \leftarrow 0$ 
15     else
16          $i \leftarrow i + 1$ 
17     end
18 end
19 return  $s, f$ 

```

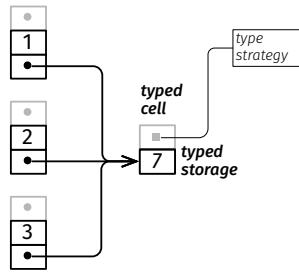


FIGURE 7: Typed cells connect compound values with mutable objects of known type in an optimized way.

*Restricted
mutability
with cells*

or merely one more `VM`-level class to defer merging and access to. If the latter infrastructure is already present — say, shapes are represented by a class hierarchy — this should be doable in rather few lines of code. By allowing restricted mutability, our approach can be used in more broad circumstances without a substantial loss of performance, if used sparingly.

4.1.3 Typed mutable cells

The fact that cells and their content cannot ever be inlined into compound values but are always shared between all referrers also means that a cell is actually the central point of information on its content. The cell “knows” what it refers to, and if and when it changes. This makes it possible to speculate on the *content of a cell*. In that regard, insights gained from, for example, storage strategies [16, 91] can be applied.

For example, when a cell refers to a number of some kind, this knowledge can be used to actually store the number *inside* the cell, saving one indirection, as in figure 7. Similar to the way storage strategies [16] work for larger objects or arrays, this automatic unboxing can save memory and decrease execution time. In principle, any type or property can be combined into the cell, be it that the cell references an integer, a float, or a certain kind of special object. Such a *typed cell* can alleviate the fact that a cell by itself inhibits information of its content to be available to its referrers, that is, other compound values.

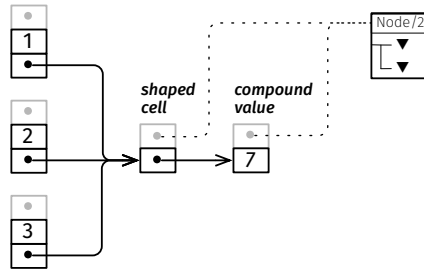


FIGURE 8: Shaped cells connect compound values with other compound values but inhibit optimization.

However, in a dynamic language, mutations to the cell's content have to take into account that the type of the object to be stored can be different from that of the cell. In this case, transition methods just like with storage strategies have to be in place, with virtually the same benefits and liabilities.

4.1.4 Shaped cells

In the special case that a cell's content is, in fact, a compound value, it is possible to specialize the cell to be aware of the content's shape. This could be beneficial in cases where making copies of rather large compound values is unwelcome and, for performance reasons, the very instance of a compound value should be shared between several other compound values. A *shaped cell* can save the indirection of directing the query for the shape of the cell's content to the pointed-to compound value and rather return the resulting shape itself, as depicted in figure 8.

In that case, the propagation of the shape could go even further, as to also introduce an indirection in the shape of the compound value that references a shaped cell. In this case, the parent's shape would refer to a *cell shape cell* that directly references the shaped cell in a manner that makes the parent compound value's shape mutable. However, as this will surely increase the complexity of handling shapes during both inlining and reconstruction, this approach is not considered further here.

4.1.5 Limitations

With cells, it is theoretically possible to introduce *cycles* and recursion in a compound value. This violates one of the basic assumption of values and may lead to surprising effects. Therefore, cells with compound value contents should be used rather sparingly.

If mutability is predominant, using cells might be ill-suited. Rather than yielding performance benefits, the additional indirections introduced by an expectedly large number of cells will likely result in increased memory usage and reduced execution speed. Self's *maps* [24] and V8's *hidden classes* [3] provide better approaches to such usage characteristics.

Although there are no substantial changes necessary to the base approach, it has to be noted that introducing cells into the shape-based inlining optimization will inevitably increase its complexity, if only ever so slightly. Good engineering practices have to be in place to handle that.

To profit from typed cells, it can be tempting to replicate optimizations from, for example, storage strategy representation mechanisms or the shape-based inlining itself. This can increase the complexity of a typed cells implementation substantially to the point that parallel optimization implementations can emerge. It is therefore advisable to use typed cells for only the most frequent cases, such as unboxed numbers.

Lastly, the presence of cells can influence how compound values are employed by developers using a programming system with cells and compound values. It might be less predictable in which cases optimization are applicable and in which not, as it is not observable from the programming language whether cells are present in certain compound values. Moreover, cells allow to use mutable elements within originally immutable structures, which could lead to an overuse of mutable elements. This in turn would render the whole approach ineffective.

*Restricted
mutability
with cells*

4.2 SHAPE-GUIDED AUTOMATIC UNBOXING

*Extended
shape-based
optimiza-
tions*

A common way to implement the semantics of simple values in a programming system is to represent them *boxed*. That is, to facilitate a uniform handling of language-level objects, objects that represent for example numbers are wrapped at the **VM**-level. This makes it possible to adjust the mapping from language-level entities to **VM**-level ones while maintaining control over its semantics. Simply put, by having a **VM**-level class `Number`, that wraps what on the language-level is a `2`, it is possible on the **VM**-level to decide how subtleties like rounding issues, widening, or type coercion should take place. This **VM**-level class is not visible to the language-level, but can partake uniformly in interactions with other entities. For example, consider another **VM**-level class `Array` that is the representation of a language-level list. It is now possible to express questions such as equality, containment, or inter-reference uniformly, given the interface of a `Number` object is compatible with that of an `Array`. Moreover, it is then unnecessary to provide specialized versions of `Array` to express both containment of entities like numbers or references, as effectively *boxing turns primitive values into references*.

This comes at a cost, as every operation on numbers now needs a dereference, and all calculations possibly result in multiple allocations of such hulls for numbers. There are different approaches to alleviate the inevitable performance impact of boxed values. The **JVM** and Java, the language, provide a primitive–object duality for certain values, especially numbers, and typed collections. That way, boxed numbers can be mixed with other objects when necessary, but primitive numbers can be used in specialized collections and especially most calculations to control performance. Smalltalk-80 and other dynamic languages “cheat” by representing values like numbers in a *tagged* fashion, that makes it easy to identify them on the language level, intermix and compare them with other objects, and maintain acceptable performance. However, tagging requires to actually check for tagged object virtually everywhere in the **VM** implementation. Therefore some language implementations, for example Python/CPython, rather represent numbers and similar primitives in a boxed fashion, and maintain a cache of often-used boxed objects. That way, at least the impact of frequent allocations it mitigated.

In environments that maintain boxed values, a typical optimization on the VM-level is to *unbox* these values wherever it is safe to do so. For example, if a JIT can infer that the addition of three numbers will, in fact, always involve only numbers, it might unbox all three values *before* carrying out the addition. In a typical, unoptimized case, that would not happen and after the first addition, the primitive result would be boxed just to be unboxed again for the second addition. Many such optimization exist, whether for individual operations, whole methods or instruction traces (for example, via escape analysis), or collections (for example, with storage strategies).

In the light of these constraints, the shape-based compound value representation can be extended to *guide* automatic unboxing of primitive values into suitable representations.

Shape-guided automatic unboxing

4.2.1 Changes to base approach

In the base approach, inlining the storage of a constituent into its referencing compound value means that all storage elements become part of the referencing compound value's storage. For automatic unboxing, a constituent that represents a boxed primitive value has to be equipped with a unique shape per primitive type. For example, all boxed integers have a special `BoxedInteger` shape instead of a *direct access* shape, boxed floats a `BoxedFloat` shape and so on. Based on these shapes, during the inlining process, the *value* of such boxed primitive is selected for merging into the referencing compound value's storage instead of a storage. Or phrased differently: boxed primitive values are implicitly treated as unary compound values with a possibly type-specific storage. Refer to [algorithm 3](#) for a possible adaption of [algorithm 1](#) for shape-guided unboxing.

4.2.2 Impact

Conveying boxing information via shapes lifts VM-level storage containers from having to introspect contents for optimization and potential unboxing. This responsibility is delegated to the inlining process. We believe that equipping shapes with information on primitive types in this way can also aid JIT

ALGORITHM 3: Adaption of shape and field determination to automatic unboxing.

```

1  Input:  $s : Shape, f : [Value Object]$ 
2   $i \leftarrow 0$ 
3  while  $i < |f|$  do
4       $s_i \leftarrow f_i\{shape\}$ 
5      // dispatch on special direct-access shapes
6      if  $s_i = s_{BoxedInteger}$ 
7           $c \leftarrow f_i\{integervalue\}$ 
8      elseif  $s_i = s_{BoxedFloat}$ 
9           $c \leftarrow f_i\{floatvalue\}$ 
10     elseif  $s_i = \dots$ 
11          $\vdots$ 
12     else
13          $c \leftarrow f_i\{storage\}$ 
14     end
15      $s' \leftarrow transformations_{s,i,s_i}$  or  $s$ 
16     if  $s' \neq s$ 
17          $f \leftarrow [f_0, \dots, i-1, c, f_{i+1}, \dots, |f|]$ 
18          $s \leftarrow s'$ 
19         // restart with new storage
20          $i \leftarrow 0$ 
21     else
22          $i \leftarrow i + 1$ 
23     end
24 end
25 return  $s, f$ 

```


compilers in applying more broad optimizations. Also, this technique serves a similar purpose as *pointer tagging*, meaning that both shapes for boxed values and tagging provide a way to support primitive and “non-primitive” objects at the same time. However, the shapes are more versatile, as they inherently allow more types to be conveyed than with tagging. Typically, pointer tagging with integers do not support more than 8 types on contemporary 64 bit systems, and NaN-tagging is usually only useful if the language has floating point numbers as its base numeric type, which is the case, for example, in Lua and JavaScript.

However, shape-guided unboxing causes a complexity increase in the whole inlining process. First, an individual unboxing step has to be devised for each supported boxed primitive type; in [algorithm 3](#), we have only given two examples for simplicity, but the list could be much longer. Second, if the `VM` implementation level does not support polymorphic storage to implement language-level data structures, line 17 of [algorithm 3](#) might be vastly more complex to achieve. In the worst case, the `VM` level has to provide n^m storage variants for a maximum length of m and n boxed types. If both are bounded reasonably, however, this optimization can be worthwhile.

*Immutable
boolean field
elision via
shapes*

4.3 IMMUTABLE BOOLEAN FIELD ELISION VIA SHAPES

Previous research on record data structures [94] has shown that several programming systems hold a notion of invariant values. For example, Scheme and Racket boolean comparisons rely on the fact that every value that is *not* the value `#f` is considered true. This has as effect that the value `#f` is treated specially in many places. Similarly, the `nil` object is special in Smalltalk. Both have in common that they are unique in the system and typically well-known to the `VM` level. This means, for example, that in Smalltalk programs, the identity of `nil` is universally known and checking for its identity heavily optimized. While this might seem contrary to the required properties of compound values, it is however possible to use this fact to the advantage of the inlining process.

We extend our approach to represent invariant values in a certain, optimized way by giving them a special shape. The fundamental idea is to not *store* the

special value inside compound values but rather indicate its *presence* through the shape of a compound value. As basis, we use our previous research on this topic [94], which applied the concept to *structures* of Racket in the Pycket implementation.

In Racket, the truth value `#f` is ubiquitous and often used as default value. When `#f` is stored in immutable fields of data structures, these are often **immutable boolean fields (IBFs)**. We augmented the existing structure metadata by an *indicator* whether a certain field is actually an **IBF**. Using this indicator, it is possible to omit the `#f` from the fields of a new structure, which saves space. (Please refer to [appendix D](#) for details of this work in context of Pycket.)

With minor modifications to the notion of shapes, certain aspects of the **immutable boolean field elision (IBFE)** in Pycket can be intuitively expressed with our approach.

4.3.1 Changes to base approach

For our approach, a specialized shape can act as an *indicator* in the sense of **immutable boolean field elision (IBFE)**?² That is, similar to unboxing, the language-special value is equipped with a special shape. During the inlining process, rather than inlining any storage or unboxing a value, nothing is stored in the referencing compound value. Refer to [algorithm 4](#) for a possible adaptation of [algorithm 1](#) for shape-guided unboxing. Note that this variant requires the *learning* step to be adapted, too. During recognition, it is now necessary to create new shapes that replace a default *direct access* shape by a kind of **IBFEShape** instead of a shape denoting a compound value. It is conceivable to support more than exactly one language-special value, for example, to support a true as well as a false value. In that case, a more complex condition is necessary in line 7 of [algorithm 4](#) and the learning step has to cope for multiple different **IBFE** shapes. Upon field access, which is also subjected to a compound value's shape, the special value is returned instead of accessing the storage, just as with the **IBFE** approach.

²Since this approach supports special values beyond booleans, “immutable singleton field” would be more apt. For consistency, we will stick with the name from our previous work.

ALGORITHM 4: Adaption of shape and field determination to immutable boolean field elision

```

1  Input:  $s : Shape, f : [Value Object]$ 
2   $i \leftarrow 0$ 
3  while  $i < |f|$  do
4       $s_i \leftarrow f_i^{\{shape\}}$ 
5       $s' \leftarrow transformations_{s,i,s_i}$  or  $s$ 
6      if  $s' \neq s$ 
7          if  $s_i = s_{IBF}$ 
8              // elide immutable boolean field
9               $f \leftarrow [f_0, \dots, i-1, f_{i+1}, \dots, |f|]$ 
10             else
11                  $f \leftarrow [f_0, \dots, i-1, f_i^{\{storage\}}, f_{i+1}, \dots, |f|]$ 
12             end
13              $s \leftarrow s'$ 
14             // restart with new storage
15              $i \leftarrow 0$ 
16         else
17              $i \leftarrow i + 1$ 
18         end
19     end
20     return  $s, f$ 

```

4.3.2 Impact

This extension to the base approach is worthwhile whenever programming systems or languages make extensive use of a very small number of special values. We have shown that for `#f` in Racket [94], and `nil` in Smalltalk [91]. In such cases, the memory consumption can be drastically reduced. However, similar to the unboxing approach, the complexity of the merging algorithm as well as the number of shapes can increase manifold; in cases where many language-special values have to be supported — for example, when considering characters —, the size and complexity of managing the metadata might outweigh their gains.

*Extended
shape-based
optimiza-
tions*

4.4 STABILITY FOR SUSTAINABLE PERFORMANCE

The shape recognition approach shares a characteristic with typical optimizing JIT compilers. They collect profiling data during un-optimized execution that serve as reasoning base for creating optimized code. This is often called the *learning phase* or *warm-up*.

4.4.1 Warm-up times

Collecting the profiling data takes time, especially because profiling happens during un-optimized execution. This *warm-up* time is more severe in our shape recognition approach, as new shapes and transformations between shapes can only happen incrementally. For example, it is not easily possible to go from an un-inlined compound value to a three-times inlined compound value in one step in the first run; there are several transformations from an un-inlined to a one-time inlined compound value necessary to actually create a two-times inlined compound value in the first place. This is, however, intentional: to not clutter the transformation table of a shape with too many transformations, these transformations are only created after a certain number of shape observations, which is recorded in the shapes history (cf. figure 3). This threshold

(cf. [section 3.5](#)) has also to be reached for every subsequent shape observation to create a new one that describes a more inlined compound value.

4.4.2 Cross-run profiling data

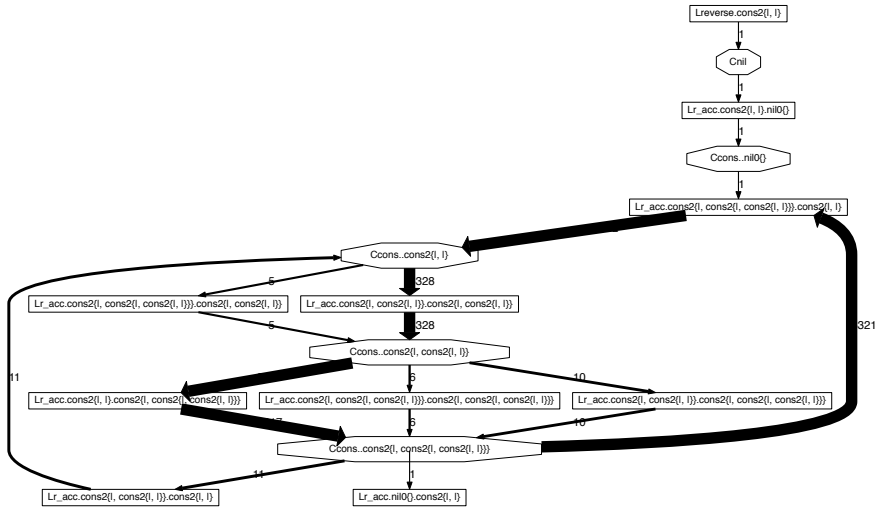
To minimize warm-up times, an offline³ cache can be used for profiling data and, moreover, the transition rules derived from them. These data are work-load-specific and therefore typically only used for a single program run. However, the structure of the profiling data and transition rules as presented are able to compensate that. If cached data from a previous execution is used for a work-load that does not fit the cached data, the worst that could happen is that no optimizations are applicable and the execution environment has to re-do the learning phase and warm-up its profiling data. That is, the execution for the unsuitable work-load would act as if no cached profiling data was present whatsoever.

*Stability for
sustainable
performance*

However, if the work-load fits that of a previous run close enough, the profiling data and, more importantly, the transformation rules can be used right away. This effect can be seen in [figure 9](#): in [figure 9a](#), where warm-up is present, intermediate shapes and transitions can be observed (thin lines, low number of observations), whereas in [figure 9b](#), using cached data and rules, the intermediates are gone. There are fewer intermediate shapes and fewer branches, indicating a higher performance.

Re-using profiling and transformation data leads to increasingly stable performance for similar work-loads, since more optimized data structures can be used directly after start-up. Note that the recognition mechanism is not, in fact, disabled when profiling and transformation data are re-used but rather is invoked less often. When a work-load is executed that does not fit the cached data, the recognition mechanism updates the profiling and transformation data accordingly. These new learned data are stored together with the already cached data at the end of the execution. That way, a comprehensive set of optimization information can be collected over time.

³that is, persistent across program executions.

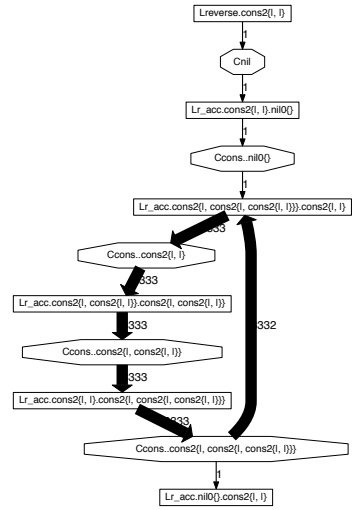


(A) With warm-up; no cached data

Notes on notation:

The stronger the arrow, the more observations. Observations of shapes in value constructions are prefixed with a “C”, followed by the name and the sub-shapes. Observations of shapes in lambda applications are prefixed with a “L”, followed by the function name (if known) and the shapes of the arguments, separated by dots. Shapes are denoted as “nameN{sub-shape...}” with N being the number of fields in a described compound value, (here only “nil” with no fields and “cons” with two fields); *direct access* shapes are denoted as “|”.

Example: **Lr_acc.nil0().cons2(1, 1)** -- function “r_acc” with a compound value of shape “nil” (no field) and no sub-shapes in the first argument and a compound value of shape “cons” (two fields) with two direct-access shapes in the second argument.



(B) No warm-up; with cached data

FIGURE 9: Transitions between observed shapes during the execution of reversing a 10 000 element list. Fewer lines is better.

4.4.3 Changes to base approach

Since the use of cross-run profile data is orthogonal to the shape inlining approach, no change to the inlining approach itself is necessary. However, the environment has to ensure that shapes are stored on disk during program termination and properly loaded at program start up.

4.4.4 Impact

Discussion

Reusing shapes and transition rules across program runs can reduce impact on the learning process. Moreover, if desired, this can be extended to the point that cross-run data is only collected once and then re-used multiple times. At the same time, the learning part of our approach could be disabled and the whole inlining process could rely solely on pre-recorded shapes and transformations. This could improve performance predictability, as warm-up is essentially eliminated.

However, the persistence and reconstruction steps can lead to a more complex startup procedure and might require special handling of abnormal program transformation, as it is unclear, whether the recorded data is in a valid state.

4.5 DISCUSSION

While the approach in [chapter 3](#) is viable on its own, the four extensions presented here can ease the integration with existing systems or enhance certain aspects of the original approach.

The inclusion of cells to support mutable parts of compound values can be a workable method integrate compound values with programming languages that cannot work without mutable state in certain areas. Moreover, the presence of cells in the approach has no performance impact when they are not used. Thus, if only a small fraction of all data structures can be expected to need modification, the inclusion of cells can be valuable. However, the benefit

of the advanced cell concepts, typed cells and shaped cells, depends more on the languages' semantics and the target implementations' architecture.

Automatic unboxing is a key performance technique, which can also be implemented using shapes. Particularly in systems that have a dichotomy between primitive and object-based data types, concepts that resemble values are introduced to control boxing and unbox of data structures. However, this can complicate the adoption of a shape-based approach as shown here. It is likely that optimizations for unboxing already exist and these are likely to interfere with the shape-based approach.

Languages and systems with a small number of special but ubiquitous values can benefit from **IBFE**. If the approach from [chapter 3](#) is employed, then using the presented shape-based approach to represent such special values can be helpful and implemented with little overhead. The key factor here is whether the value is frequently used as the default value.

The shape-based inlining approach has an inherent warm-up phase to determine transformation rules. However, when the expected execution time of programs is very short, for example, in scripts, there might not be enough time to create optimized compound values. In this case it might be desirable to reuse optimization data from previous program execution. Likewise, when it is crucial to have peak performance right from the start of a program, such cross-run profiling and optimization data can be beneficial. Since this approach is mostly orthogonal to the other parts of the shape-based optimization, this extension to our approach is often straightforward to provide.

SUMMARY

The shape based inlining can be extended with limited mutability, automatic unboxing, elision of few constant values, and cross-run optimization data.

5 Compound values in action

The ship wherein Theseus and the youth of Athens returned from Crete had thirty oars, and was preserved by the Athenians down even to the time of Demetrius Phalereus, for they took away the old planks as they decayed, putting in new and stronger timber in their places, insomuch that this ship became a standing example among the philosophers, for the logical question of things that grow; one side holding that the ship remained the same, and the other contending that it was not the same. (Plutarch, “Theseus”)

This chapter presents an implementation of the approach set forth in the preceding two chapters as feasibility prototype. The priorities of this implementation are to show

- that the algorithms and data structures presented can be implemented,
- that the implementation is achievable with comparatively low effort, and
- that the postulated performance benefits can be shown to exist.

Therefore, the prototype comprises a new programming system with a focus on these priorities. We make use of the RPython toolchain, as its meta-tracing JIT compiler provides the necessary features expected by our algorithm.

5.1 BEST-CASE PROTOTYPE SYSTEM: THESEUS

The prototype, named Theseus [90], implements a minuscule programming system with a simple execution model. It provides λ -expressions with pattern matching as the sole control structure. There is only one structured data type available, compound values — designated as “constructors” after the ML-style algebraic data types of the same name.

The programming system uses a new language that provides these very constructs. A program in this system can consist of

- values (including λ -expressions),
- applications, and
- definitions (or assignments/name-to-value-bindings).

*Compound
values in
action*

```
 $\lambda$ .  
1. Data( _, b), c  $\mapsto$  Other(c, b)  
2. arg, _  $\mapsto$  arg  
  
⟨⟨plus⟩⟩  
 $\lambda$ .  $\mapsto$  C(1, 2.0, "hi", 'you')
```

Values in Theseus support a small number of scalar data types, namely numbers and strings. The structured data type is called “constructor” and explained below. Behavior in Theseus is expressed as λ -expressions. The more traditional Church-style notation is not supported in favor to writing patterns for matching in a readable way. Each λ -expression in our system consists of a number of (pattern \mapsto expression) pairs. Upon application, the arguments are matched against a list of patterns in order of textual occurrence. Once a pattern matches, variable names in that pattern are bound to values from the arguments and the actual expression associated is evaluated. Patterns can be specified in a *structured* manner. When a λ -expression is applied to constructors, matching will be done on the structure of these constructors. Hence, when a pattern contains constructors and (free) names within such a pattern, these names can be bound to constituents of the arguments. Apart from this, matching is done on a value-equivalence basis, as types per se are not an exposed concept in our system. The pattern matching is inspired by Prolog’s unification, but decidedly does not provide its two-way matching; it does, however, include the “do-not-care” free variable for patterns ($_$). Besides λ -expressions, the language supports $\langle\langle\rangle\rangle$ -enclosed *primitives*, which can be applied similarly, but instead of being backed by rules, patterns, and expressions, their behavior is defined within the **VM**.

Constructors comprise a “tag” and constituents; the tag is used to name a group of compound values belonging together and match against during pattern matching. Constructor tags do not have to be declared, they merely exist and can be *observed* during compound value construction and pattern matching in λ -expressions. This also means that there is no dedicated functionality to access the constituents of a compound value, pattern matching

is the sole way to do that. An example to emulate access via utility functions (that is, λ -expressions bound to names) can be found in [listing 3](#).

Expressions at the top-level of a program are evaluated sequentially. There is no way to express a sequence of expressions within a λ -expression. To achieve sequential execution, a kind-of continuation passing style must be employed, with the “next” expression being passed as argument to an application. Values can be referred to by variables, both top-level and pattern-bound ones; scoping is lexical.

Application of λ -expression to values is explicit, in contrast to most popular languages, denoted by $\mu(\dots)$. The first argument to the μ denotes the λ -expression to be evaluated, the rest is passed as arguments to that λ -expression.

Definitions are only allowed at the top-level of a program. Assignable, local variables do not exist in favor of name-bindings that originate from pattern matching. References to globally defined names are evaluated eagerly, there is no dynamic lookup. This is possible since all data structures are immutable, following from compound value characteristics. However, to support recursion, definitions binding a λ -expression to a name are treated specially so that the name can also be used within the expression itself. Similarly, *forward-declarations* of such names is possible to support co-recursion. This is the sole instance of names being able to change its meaning and not considered a defining feature of the language. While it would be possible to omit this inelegant special case with Curry’s *Y combinator* [30, page 178] or similar techniques, concerns for practicability and readability — at least partially — lead to this shortcut.

Comments are provided in a way typically used in scripting languages; everything following a # character until the end of line is treated as non-existent.

The grammar for this language is given in [appendix C.1](#). An example program in Theseus with the well-known map function can be found in [listing 4](#).

Theseus

```
 $\mu(+, 1, 2)$ 
 $\mu(\lambda. x \mapsto \text{Data}(x), 17)$ 

name := ...

name :=  $\lambda.$  ... name ...
 $\lambda.$  name

# comment
```

LISTING 3: Example for *cons* cells provided with function-like λ -expressions *cons*, *head*, and *tail*.

```
1 # type Nil, Cons(,)
2 cons :=  $\lambda$ . A, B  $\rightarrow$  Cons(A, B)
3 head :=  $\lambda$ . Cons(A, B)  $\rightarrow$  A
4 tail :=  $\lambda$ . Cons(A, B)  $\rightarrow$  B
```

*Compound
values in
action*

LISTING 4: Example program for Theseus that implements the “map” function, which applies another function to all elements of a list.

```
1 # type Nil, Cons(,)
2 map :=  $\lambda$ .
3     1. fun, Nil()  $\rightarrow$  Nil()
4     2. fun, Cons(A, B)  $\rightarrow$  Cons( $\mu$ (fun, A),  $\mu$ (map, fun, B))
```

5.2 PROTOTYPE IMPLEMENTATION

Theseus is implemented as an interpreter *VM* with a direct application of the *control, environment, and continuation (CEK)*-machine [43]. We used the RPython tool chain to incorporate its meta-tracing *JIT* compiler [12], hence, the source language of our *VM* is Python. Moreover, we make use of a facility in the tool chain, a parser generator, to keep the implementation overhead of syntax elements to a minimum. While Theseus is simple and probably even easier to implement without a parser generator, early development stages required constant changes to the grammar. This was best met with the RPython-integrated tool, as it could be used incrementally, without generating code. The grammar in [appendix C.1](#) is the actual input to that generator, which in turn creates callbacks for all tokens and rules (except when enclosed in angle brackets).

5.2.1 Shape determination and inlining

The application of [algorithm 1](#) to Theseus is a rather direct mapping with few exceptions (cf. [listing 5](#)). However, since Theseus implements shapes as classes, the transformation table and the history are actually properties of a shape, and

LISTING 5: Implementation of the merging and inlining algorithm in Theseus

```
1 class CompoundShape(Shape):
2     @jit.unroll_safe
3     def merge(self, storage):
4         current_storage = storage
5         index = 0
6         shape = self
7         storage_len = shape.storage_width()
8         while index < storage_len:
9             child = current_storage[index]
10            subshape = child.shape()
11            new_shape = shape.get_transformation(index, subshape)
12            if new_shape is not shape:
13                child_storage = child.get_storage()
14                new_storage = _splice(current_storage, storage_len, index,
15                                     child_storage, subshape.storage_width())
16                current_storage = new_storage
17                shape = new_shape
18                storage_len = shape.storage_width()
19                # rewind over new storage
20                index = 0
21            else:
22                index += 1
23            return (shape, current_storage)
```

*Prototype
implementation*

the determination and inlining algorithm is implemented as a method instead of a function. Because of that the method is named “merge”: starting from a tag’s default shape, and given a compound value’s constituents, sub-objects are potentially *merged* into the running storage.

For integration with the framework, the method is decorated with the annotation `@jit.unroll_safe`, which tells the RPython `JIT` compiler to consider this method for tracing regardless of the fact that it contains a loop (line 8). The `JIT` compiler normally does not consider methods that contain loops to avoid code explosion during loop unrolling. However, since the loop here is bounded by object size, and objects are not expected to be arbitrarily large in Theseus, unrolling and tracing is safe here.

LISTING 6: Implementation of the shape integration and storage selection for Theseus constructors.

*Compound
values in
action*

```
1 class W_Constructor(W_Object):
2     _immutable_fields_ = ['_shape']
3     def __init__(self, shape):
4         self._shape = shape
5     def get_tag(self):
6         return self.shape()._tag
7     def get_children(self):
8         return self.shape().get_children(self)
9     def get_child(self, index):
10        return self.shape().get_child(self, index)
11    def get_number_of_children(self):
12        return self.shape().get_number_of_direct_children()
13    def shape(self):
14        return jit.promote(self._shape)
15    @staticmethod
16    def construct(shape, storage):
17        return W_Constructor.make(storage, shape)
```

5.2.2 Constructors

The compound values of Theseus, constructors, are implemented as essentially data holders with a shape and a certain number of fields. However, to assist RPython and the JIT compiler, we chose not to represent the storage of a constructor as simple RPython list. While this would be very easy to implement and manage, the characteristics of these lists within the RPython framework are actually too dynamic and account for mutability, which is a non-issue for compound values. Also, we expect constructors not to be very large, which is in line with experience from other languages that are in need of storage optimization [16]. We hence represent constructors as instances of VM-level classes, which are generated during the RPython process. We generate one class for each potential object size up to a certain number of fields—currently 31, so that objects on x86_64 machines do not exceed the size of 256 B—and one class with an array as storage to represent larger objects. Additionally, for very small constructors that comprise numbers, we generate a

few extra classes so that the `JIT` compiler is made aware of the types of these fields. Moreover, these classes implement automatic unboxing akin to the idea presented in [section 4.2](#), but not as generalized as given there.

Constructors defer all access to their shapes, which in turn are aware of any inlining that has taken place, and then can meaningfully access the tag or individual sub-objects of a constructor. That way, the constructor implementation is very simple yet extensible by the shape (cf. [listing 6](#)). This can make future additions of shape-guided optimizations easy to implement.

Constructors use a certain integration with the `JIT` compiler. In [line 14](#) of [listing 6](#), the shape of a structure is “promoted”. This special call instructs the `JIT` compiler to promote [[14](#), § 3.1] the given value. This ties the object identity of this value to a certain trace. As long as the value stays the same, the same trace keeps being recorded. It is a typical *hint* to give to a meta-tracing `JIT` compiler in certain locations of a program, and certainly is useful for our shapes. Using `promote` ensures that the `JIT` compiler can specialize its traces for different shapes individually. A concrete shape inherently conveys the information of what is inlined in a compound value. By promoting the shape, the `JIT` compiler can use this information to optimize for the actual presence or absence of inlined fields.

*Prototype
implementa-
tion*

5.2.3 Implementation characteristics

The `VM` consists of roughly 50 Python classes and 240 methods which are distributed among only a handful of files. The implementation has been carefully unit-tested during development to make sure that various complex substitutions and compressions work correctly.

All in all, the implementation consists of $\approx 10\,000$ source lines of code, where ≈ 1600 belong to unit tests. The grammar input to the parser generator accounts for less than 100 lines — the generated parse, however, accounts for 70 % of all source lines of code in Theseus, that is ≈ 5700 . Since the generated code can be omitted, the core of Theseus amounts to ≈ 2600 lines of Python.

5.3 INTERACTION WITH THE JIT COMPILER

*Compound
values in
action*

The implementation, following the algorithm, provides the intended memory usage reduction. However, shape recognition, shape inlining, and reconstructed reference access combined, do not yield a performance increase on their own. In fact, implementing the approach straightforwardly can yield significantly worse performance than not using the approach. This is due to the constant checking of the transformation rules every time a new value object is created. Additionally, reading inlined fields of compressed value objects results in the allocation of intermediate data structures. This is of course not the case in the naive representation. Hence, the presence of the `JIT` compiler is necessary to begin with.

To improve performance, the `JIT` compiler needs to reduce the overhead of these operations. The first step is to treat the transformation tables as constant when a function is compiled. This allows the `JIT` compiler to compile compound value creation down to a series of type checks for the types of the referenced compound values. We instruct the `JIT` compiler to treat transformation tables as constant after filling it with enough information.

Second, we have to avoid the otherwise necessary reconstruction of referenced compound values when it is being read from a compound value it has been inlined into. For that, the observation that most of these intermediate compound values are actually short-lived is crucial; most compound values are created just to be either immediately discarded or consumed in another, typically larger data structure. As a concrete example, typical linked list operations deconstruct the list they are working on. If we access the tail of a linked list node that has an inlined component (as the transition from left to middle in [figure 5](#) on [page 31](#)), we need to reconstruct the tail. However, that tail itself is usually deconstructed soon into its components (as the transition from middle to right in [the same figure](#)). This allows the tracing `JIT` compiler to optimize the reading of fields that need reconstruction. Since the compound values allocated when reconstructing a field are short-lived, the built-in escape analysis and allocation removal [13] will fully remove their allocation and thus remove the overhead of reconstruction.

5.4 DISCUSSION

The data structures and algorithms described in [chapter 3](#) are indeed feasible to implement. As shown above, the implementation presented is — compared with other VM implementations — rather small and self-contained. With less than 3000 source lines of Python code, this artifact is arguably simple; the only deviations from a most simplistic implementation are the actual inclusion of the shape indirection and hence indirections for the creation of and the access to constructors, as well as the storage selection, laid out above. As the last of the stated priorities, we show existence of a performance benefit later in [chapter 7](#).

Discussion

Reviewing the challenges introduced in [section 1.1](#), our implementation is designed to provide a uniform language view of compound values ([challenge 1](#)), regardless of the way which representation or inlining is chosen or discovered. Faithfully applying the recognition steps yields dynamic adaption of inlining for different usage patterns, which is hence again supported by design, ([challenge 2](#)). Since only compound values exist in Theseus, [challenge 3](#) does not apply here.

*
**

Theseus provides a concise and simple example of the feasibility of our proposed approach, and while with limited functionality compared with general purpose language, it can well serve as a baseline for qualitative and quantitative comparison.

SUMMARY

Theseus is a simple language implementation with compound values as its sole data type. The transfer of the inlining algorithm from the approach to Theseus was straightforward. The implementation is well-integrated with the JIT compiler of the toolchain.

6 Compound values in practice

For if, for example, that ship of Theseus, concerning the difference whereof made by continual reparation in taking out the old planks and putting in new, the sophisters of Athens were wont to dispute, were, after all the planks were changed, the same numerical ship it was at the beginning; and if some man had kept the old planks as they were taken out, and by putting them afterwards together in the same order, had again made a ship of them, this, with out doubt, had also been the same numerical ship with that which was at the beginning; and so there would have been two ships numerically the same, which is absurd. (Hobbes, *Elements of Philosophy*, Ch. 11, § 7)

In the previous chapter, we have shown that implementing our approach is indeed feasible. This chapter sets out to apply our approach to existing general-purpose language. By way of qualitative evaluation, we transfer the findings from Theseus to two other implementations, that is, Pycket, an implementation of the Racket [45] language, and RSqueak, an implementation of the Squeak/Smalltalk VM[60]. This is to answer the question whether a trade-off between valueness and performance can be observed and if so, whether it is acceptable.

6.1 CANDIDATE SYSTEMS

As first system, we chose Racket, a dynamically typed, multi-paradigm programming language, because it already has a concept of immutable data structures. Racket supports immutable-by-default lists and an implementation of design-by-contract [77]. As a member of the Scheme family of programming languages, the well-known *cons* data structure is available, but in Racket, it is

immutable. To emulate previous versions' behavior or to be compatible with other dialects, developers have to actively use a mutable version of *cons* cells or use macros to emulate them. Similarly, the Racket-specific “structures” — a heterogeneous record data type — is also immutable by default. Refer to [section 2.2.2](#) for a description of this data structure.

We chose the Pycket implementation of Racket for two reasons. First, Pycket is written using the RPython tool chain and hence transferring our learnings should not suffer from architectural and language differences. Second, Pycket's interpreter is a direct application of the CEK machine, just as Theseus. In fact, there has been a certain degree of influence from what was learned in Pycket's implementation on the development of Theseus. We expect that these similarities help assessing the applicability of our approach to a general purpose language.

The second system considered is Squeak/Smalltalk. Just like Theseus and Racket, Smalltalk is dynamically typed, has a comparatively uncomplicated execution model, and almost always runs on VMs. However, unlike the other two, Smalltalk revolves around objects with identity and mutable state; immutability is virtually absent. Moreover, the Smalltalk's *image* concept provides a clearer distinction between responsibilities of the VM and the languages running on it. For example, the language's compiler and most of the common data structures are not part of the VM but defined at the language level with a small but clear interface. This makes Squeak/Smalltalk a good candidate to assess the boundaries of how applicable our approach is.

Among several VM implementations of Squeak, we chose the RSqueak VM. Since the RSqueak VM also uses the RPython toolchain, we expect a low effort for the mere code integration. The architecture and JIT compiler integration is however sufficiently different to pose a new challenge, when compared with Pycket. Moreover, Squeak as a system in general and the RSqueak VM as software have a notion of plug-ins. This makes it possible to assess in how far our optimization can be used only optionally and disabled at the developers' discretion, and what the impact of this is.

6.2 STRUCTURES IN PYCKET

Pycket [10, 18] is a recent implementation of Racket using the RPython toolchain and its tracing JIT compiler. It already provides a wide range of Racket's functionality — such as continuations, contracts, classes, or dynamic bindings, to name a few. On a standard set of benchmarks, Pycket can compete with the reference Racket JIT compiler implementation performance-wise, in certain areas even outperforming high-performance AOT Scheme compilers.

The structure types are of special interest because, if applied carefully, they can be used like compound values. For that, a value-based comparison of structures can be enabled explicitly — the default is identity-based comparison. Moreover, structures can form hierarchies and are immutable by default with the option to make some or all fields mutable. Racket structures go beyond other structured heterogeneous data types; they support the notion of structure type properties that can influence the way structures interact with the system. For example, a special structure type property can make structure instances *callable*, so they can act like a procedure.

One of the most used data structures in Racket are *cons* lists. Changing this data structure to a proper compound value and applying our approach would potentially and unknowingly affect parts of Racket out of our scope. Rather, we chose to adapt Racket structures, which have less system-wide impact. As a convenience, we provide a Racket library (cf. listing 7) that allows us to use compound values in the form of structures while using standard Racket's *cons* interface as the surface syntax for a contained scope, such as, one application or one benchmark. *Cons* cells in Racket are immutable anyway, and in *most* applications are used without regard to their identity, therefore we can use them in a way that non-value effects are typically not noticeable.

Another candidate data structure in Racket are vectors, which are fixed-length array-like lists with index-based access. They exist in both mutable and immutable variants. However, in the packages shipped with the standard Racket distribution, only about one percent of the vector structures are immutable vectors. This is one reason we first focus on *cons* lists. Another reason is that vectors are slightly more involved when they are to be emulated using

*Structures in
Pycket*

LISTING 7: Emulation of the Racket *cons* interface by means of structures

*Compound
values in
practice*

```

1 #lang racket/base
2 (require (for-syntax racket/base) racket/performance-hint)
3 (require racket/provide)
4 (provide (filtered-out (lambda (name)
5                       (and (regexp-match? "%pycket:" name)
6                            (regexp-replace "%pycket:" name ""))))
7         (all-defined-out)))
8 (define-values
9   (struct:cons #%pycket:cons #%pycket:pair? cons-ref #%ignored:cs!)
10  (make-struct-type 'cons #f 2 0 #f '() #f #f '(0 1) #f 'cons))
11 (define #%pycket:car (make-struct-field-accessor cons-ref 0))
12 (define #%pycket:cdr (make-struct-field-accessor cons-ref 1))
13 (define (%pycket:build-list n proc)
14   (define (mk-acc m proc l)
15     (if (= 0 m)
16         l
17         (mk-acc (- m 1) (proc m) (%pycket:cons proc l))))
18   (mk-acc n proc null))
19 (define (%pycket:make-list n e)
20   (define (mk-acc m e l)
21     (if (= 0 m)
22         l
23         (mk-acc (- m 1) e (%pycket:cons e l))))
24   (mk-acc n e null))

```

structures. That being said, compound value vectors are a logical next step for our approach.

6.2.1 Basic Adaptations

Our approach is present in a modified Pycket implementation [87]. The existing structure implementation [95] already tries to optimize memory consumption and execution time. It already deals with the distinction of smaller and larger structure instances; for the former, objects with a known, small number of fields are used, for the latter, separate storage objects are created. Hence, an abstraction for field accesses already existed. We were able to take

the implementation of Theseus with little modification and use it as storage for all structure kinds.

The merging algorithm implementation as given in [listing 5](#) could be re-used without any major modification. The modification to connect Racket structures to our approach were similarly uncomplicated (cf. [listing 8](#)). Since Pycket uses a similar technique to represent entities with storage of different lengths, the expected interface of Pycket is very similar to that in Theseus and could be mapped easily. Note that, as with Theseus, all access to actual contents of a structure is “routed” through the shape to account for possibly inlined other structures. Similarly to constructors in Theseus, we “promote” the shape of a structure, but moreover, also promote the structure’s type for similar reasons. The latter is also done in the unmodified version of Pycket.

Both shapes and structure types are data structure descriptors with mutually complementary information, thus, there is no need to retain both for each structure instance. However, to ease the re-use from Theseus, we also introduced a “tag” object that does little more than providing the default shape for a group of compound values as well as the respective structure type. Thus, access to the structure type from a structure instance now flows as “structure → shape → tag → structure type”. However, since all these stay constant right from the creation of a structure instance, this does not impact performance in any noticeable way.

*Structures in
Pycket*

6.2.2 Mutability

While structures in Racket are immutable by default, developers can ask for arbitrary fields of a structure to be mutable by specifying so in the structure type definition. This means that our change to Pycket has to account for mutability and apply our extended concept as outlined in [section 4.1](#). In fact, this change is less a change to Pycket than to our approach, as Pycket already uses cells to abstract away the mutation of individual fields. The respective low-level accessors in [listing 9](#) show that, whenever a field is marked as mutable within a structure type, it is assumed that the field is occupied by a cell. Both read and write operations are then routed through the cell. Since these are low-level operations, they do not do any error recovery in case a non-cell entity is

LISTING 8: Integration of the shape-based compound value optimization into the standard Pycket structure implementation.

```
1 class W_Struct(W_RootStruct):
2     @staticmethod
3     def make(w_field_values, w_structtype):
4         tag = jit.promote(w_structtype._tag)
5         pre_shape = tag.default_shape
6         shape, w_storage = pre_shape.fusion(w_field_values)
7         return W_Struct.make_basic(w_storage, shape)
8     def __init__(self, shape):
9         assert isinstance(shape, CompoundShape)
10        self._shape = shape
11    def get_tag(self):
12        return self.shape()._tag
13
14    # shape api
15    def get_children(self):
16        return self.shape().get_children(self)
17    # pycket api
18    _get_full_list = get_children
19    # shape api
20    def get_child(self, index):
21        return self.shape().get_child(self, index)
22    # pycket api
23    _get_list = get_child
24    # shape api
25    def get_number_of_children(self):
26        return self.shape().get_number_of_direct_children()
27    # pycket api
28    _get_size_list = get_number_of_children
29
30    def shape(self):
31        return jit.promote(self._shape)
32    def struct_type(self):
33        tag = self.get_tag()
34        assert isinstance(tag, pycket.shape.StructTag)
35        return jit.promote(tag.struct_type())
```


LISTING 9: Cells for limited mutability in the shape-based variant of the Pycket structure implementation.

```
1 class W_Struct(W_RootStruct):
2     def _ref(self, i):
3         w_res = self._get_list(i)
4         immutable = self.struct_type().is_immutable_field_index(i)
5         if not immutable:
6             assert isinstance(w_res, values.W_Cell)
7             w_res = w_res.get_val()
8         return w_res
9     def _set(self, k, val):
10        w_cell = self._get_list(k)
11        assert isinstance(w_cell, values.W_Cell)
12        w_cell.set_val(val)
```

*Structures in
Pycket*

found in a structures field. In fact, the setter does not even check whether the field is actually mutable, since the construction of structure types in Racket takes care that language-level mutators or setters are *only* available when their respective field is mutable.

6.2.3 Findings

The implementation of shapes and compound values in Pycket benefited from the architectural similarities with Theseus. Re-use of code and concepts resulted in little interference between the existing Pycket architecture and our additions. By and large, only few adaptations were necessary: we added the management logic for shapes and re-routed access to fields through them. All in all, the changes amounted to less than 550 lines of code added and a handful of lines of codes removed. In addition, we transferred unit tests for shapes and compound values from Theseus to ensure their continued functionality and added more unit tests to verify that our optimization has indeed no detrimental effect on the rest of Pycket.

Pycket already provides a well-done abstraction that hides representation details of structures from the language level. It is not obvious to developers,

LISTING 10: Current logic for `eq?` in Pycket. As per current semantics of Racket, structure equality is not value based.

```
1 def eqp_logic(a, b):
2     if a is b:
3         return True
4     elif isinstance(a, values.W_Fixnum) and isinstance(b, values.W_Fixnum):
5         return a.value == b.value
6     elif isinstance(a, values.W_Flonum) and isinstance(b, values.W_Flonum):
7         return a.value == b.value
8     elif isinstance(a, values.W_Character) and isinstance(b, values.W_Character):
9         return a.value == b.value
10    return False
```

*Compound
values in
practice*

wether a structure is using a storage representation with each field corresponding to a field in the `vm`-level class or rather an element in an array in the `vm`-level class. The language view on the data structure is always maintained. The integration of our approach made use of this abstraction, hence also maintaining this view (cf. challenge 1). Since the core of our approach implementation could be re-used in its entirety, the dynamic adaptation characteristics are the same as for Theseus (cf. section 5.4 and challenge 2). However, the observability of non-value characteristics (cf. challenge 3) for structures that are used with our compound value optimization approach cannot be ruled out in its entirety. This is because structure instances are compared by identity and are not treated specially in this regard as for example numbers are (cf. listing 10). It would be too surprising for developers to change this current logic in Pycket only. However, marking structure types as `#:transparent` ensures value equality, yet not identity. If either `eq?` or `eqv?` were `EGAL` [7] we could consider challenge 3 met (see also section 2.3.2).

6.3 RSQUEAK VALUES

Squeak [60] is a programming system derived from Smalltalk-80 [51]. Its current reference `vm`, the OpenSmalltalk `vm` [75], is based on the original Squeak/

Smalltalk **VM** [60] and applies additional optimization techniques to efficiently execute Smalltalk code.

The RSqueak **VM** [17, 42] implementation of Squeak’s **VM** is written using the RPython toolchain to benefit from its meta-tracing **JIT** compiler. With that, it can provide better performance than other Squeak **VMs**, particularly in long-running benchmarks, by employing advanced optimizations, such as but not limited to storage strategies [91].

In the object model of the Squeak **virtual machine**, all objects belong to a class, and that class determines the number of fixed fields (the same for all instance of the class) plus the number of variable fields (can vary per instance); *all fields are always mutable*. Few exceptions exist, such as objects with a variable number of byte-wise accessible slots, or specially represented objects like numbers. This has the effect, that communication with regard to object storage between the Squeak image and the **VM** typically only deals with field indices—either via executing corresponding bytecodes or invoking primitive behavior. All other tooling, such as providing instance variable names, is subject to the Squeak image. Accordingly, a rich set of collection data structures is provided by the image instead of the **VM**. Most of them are expressed in terms of Arrays, which are objects with just variable fields.

*RSqueak
values*

6.3.1 Adaptations

Introducing compound values at the **VM** level simply by providing class types with immutable fixed and/or variable fields could have drastic implications for the tooling in the image. Compared to this, we use a quite simple approach in Pycket, where exactly one kind of data structure is affected and only a small amount of code is necessary to *selectively* use compound values.

Squeak has a notion of plug-ins that can provide optional functionality, which can be invoked from the image. We use this approach to implement the interface between **VM** and image for our compound values. Since RSqueak has full control over how it represents objects from the language within the **VM**, it is simple to use different representations for objects that belong to the same class. Thus, for the same class, we can provide compound value objects at the same time as “normal”, mutable objects. To aid this process, we

LISTING 11: Integration of compound value objects as a RSqueak plug-in, general parts.

*Compound
values in
practice*

```
1 class ValueMixin(object):
2     def is_value(self):
3         return True
4     def shape(self):
5         return jit.promote(in_storage_shape_instance())
6 class W_Value(W_Object):
7     _attrs_ = []
8     objectmodel.import_from_mixin(ValueMixin)
9
10 def patch_w_object():
11     """Add `W_Object.is_value` which by default returns `False`."""
12     class __extend__(W_Object):
13         def is_value(self):
14             return False
15         def shape(self):
16             # this is not supported
17             raise FatalError
```

provide a class at the RPython level that can be used to mark classes used for object representation as having value semantics (cf. listing 11). This is done by subclassing the original class and “mixing in” our template. This mixing-in is provided by the RPython toolchain; by and large, it amounts to copying methods and class properties from the “trait” class to the new subclass. As seen in the listing, we patch RSqueak’s default representation classes with convenience methods to denote that these are not compound values. Similar to Pycket, where we introduced a “tag” as indirection to access its structure type, we now use such a tag to access a compound value object’s class, which is acquired differently in the normal representation.

The default representation class we support is that for the typical object with variable or fixed fields. The `W_PointersValue` can represent a compound value, as seen in listing 12. Note that this is very similar to the integration of compound values with Pycket structures (cf. listing 8), except for the RSqueak-specific accessors and the need to provide a class. Contrary to Pycket, we have not introduced a cell indirection for mutability.

LISTING 12: Integration of the shape-based compound value optimization as a RSqueak plug-in.

```
1 class W_PointersValue(W_Value):
2     @staticmethod
3     def make(objects_w, space, w_class):
4         s_class = w_class.as_class_get_shadow(space)
5         _tag = tag(s_class, len(objects_w))
6         (shape, storage) = _tag.default_shape.fusion(objects_w)
7         return W_PointersValue.make_basic(storage, space, shape)
8     def __init__(self, space, shape):
9         W_Value.__init__(self)
10        self._shape = shape
11    def shape(self):
12        return jit.promote(self._shape)
13    def get_tag(self):
14        return self.shape()._tag
15    # shape api
16    def get_children(self):
17        return self.shape().get_children(self)
18    # rsqueak api
19    def fetch_all(self, space):
20        return self.get_children()
21    # shape api
22    def get_child(self, index):
23        return self.shape().get_child(self, index)
24    # rsqueak api
25    def fetch(self, space, n0):
26        return self.get_child(n0)
27    # squeak api
28    def at0(self, space, index0):
29        # To test, at0 = in varsize part
30    # shape api
31    def get_number_of_children(self):
32        return self.shape().get_number_of_direct_children()
33    # rsqueak api
34    size = get_number_of_children
35    def getclass(self, space):
36        return self.get_tag().w_cls()
37    def class_shadow(self, space):
38        return self.get_tag().class_shadow()
```

LISTING 13: RPython part of RSqueak primitives to integrate compound values.

```
1 @plugin.expose_primitive(unwrap_spec=[object])
2 def primitiveIsValue(interp, s_frame, w_recv):
3     if w_recv.is_value():
4         return interp.space.w_true
5     return interp.space.w_false
6
7 @plugin.expose_primitive(unwrap_spec=[object, object])
8 def primitiveValueFrom(interp, s_frame, w_cls, w_obj):
9     space = interp.space
10    instance_kind = w_cls.as_class_get_shadow(space).get_instance_kind()
11    pointers_w = w_obj.fetch_all(space)
12    try:
13        vals_w = make_sure_all_value(pointers_w)
14    except NoValueError:
15        raise PrimitiveFailedError
16    return W_PointersValue.make(vals_w, space, w_cls)
17
18 @plugin.expose_primitive(unwrap_spec=None)
19 def primitiveValueFromArgs(interp, s_frame, argcount):
20    args_w = s_frame.pop_and_return_n(argcount):]
21    w_cls = s_frame.pop()
22    space = interp.space
23    instance_kind = w_cls.as_class_get_shadow(space).get_instance_kind()
24    try:
25        vals_w = make_sure_all_value(args_w)
26    except NoValueError:
27        raise PrimitiveFailedError
28    return W_PointersValue.make(vals_w, space, w_cls)
```

LISTING 14: Basic Smalltalk part of primitives to integrate compound values

```

1  _____ Behavior _____
2  valueFrom: anObject
3      <primitive: 'primitiveValueFrom' module: 'ValuePlugin'>
4      self primitiveFailed
5
6  _____ Object _____
7  isValue
8      <primitive: 'primitivelsValue' module: 'ValuePlugin'>
9      ^ false
10
11 asValue: aClass
12     ^ aClass valueFrom: self
13
14 valueCopy
15     ^ self asValue: self class

```

*RSqueak
values*

To instantiate compound values or identify whether any given object is actually a compound value, we provide *primitive* behavior, which can be invoked from the image. The three primitives presented in listing 13 provide the value check (`primitivelsValue`), making a copy with compound value representation from an object (`primitiveValueFrom`), and creating a compound value from a given class and a list of objects that should form the compound value's constituents (`primitiveValueFromArgs`). This provides a slim interface to the image to access and create compound value objects. Since we integrate the value representation with the way RSqueak already accesses objects, no further specialized primitives are necessary.

In our adaptation of Pycket, all Racket structures simply became compound values. However, in RSqueak, obtaining compound values at the language level can be done more selectively. For that, we equip every class with the possibility to make compound value copies of its instances by invoking the just specified primitives (first method of listing 14). Furthermore, we provide the compound value check to every object as well as convenience methods to make a compound value copy of itself (rest of listing 14). This is sufficient to use compound values in RSqueak.

To more conveniently use compound values in RSqueak, we provide variants of the well-known *vector* and *cons* list data structures from the Lisp world.

LISTING 15: A Squeak compound value vector

```

1 ArrayedCollection variableSubclass: #VVector
2 ----- VVector class -----
3 new
4 <primitive: 'primitiveValueFromArgs' module: 'ValuePlugin'>
5 ^ (self basicNew: 0) initialize
6 -----
7 with: anObject
8 "Answer a new instance of me, containing only anObject."
9 <primitive: 'primitiveValueFromArgs' module: 'ValuePlugin'>
10 " Fallback for non-immutable "
11 ^ super with: anObject
12 -----
13 valueFrom: aCollection
14 " Overridden to be co-usable in mutable form "
15 <primitive: 'primitiveValueFrom' module: 'ValuePlugin'>
16 aCollection class isVariable iffFalse: [^ self primitiveFailed].
17 ^ (self basicNew: aCollection size)
18     replaceFrom: 1 to: aCollection size with: aCollection
19 -----
20 ----- VVector -----
21 = otherCollection
22 " value semantics "
23 ^ otherCollection class == self class
24 and: [otherCollection size == self size
25 and: [self size = 0
26 or: [self hasEqualElements: otherCollection]]]

```

*Compound
values in
practice*

Providing a direct equivalent of Squeak's Array would not help, as these are almost always used in a mutable way. Specifically, they have to be created with “empty” contents at the **VM** level and filled at the language level.

In listing 15 we provide a class for compound values with variable fields. The three class-side methods `new`, `with:`, and `valueFrom:` can be used to instantiate compound values with zero, one, or any number of constituents, respectively, by invoking the appropriate primitives in the **VM**. Note that these methods contain code beyond the primitive invocation. This is fallback code, which is executed in case the primitive failed to execute in some way or when the specified plug-in is absent. We use the fallback code to create normal objects instead. That way, this code can be used regardless of whether our plug-in for compound values is present in the **VM**. Moreover, this enables to use a

Squeak image with compound values across different kind of `VMs`, not just our modified `RSqueak VM`. We consider that an advantage [92].

The instance-side equality message `=` is given in a way that it respects value equality and does not use the default object identity check. All other necessary methods to use this `VVector` as a Smalltalk collection, such as `at:` for index-based access, are already provided by its superclass `ArrayedCollection` and work without modification; very few methods are needed beyond that. Mutating methods, such as `at:put:` are overwritten to raise an error.

The `Vector` shares with its non-value counterpart `Array` that it has index-based access but cannot be changed in size. The most commonly used collection in Squeak, `OrderedCollection`, can however be resized by appending elements with the message `,` (a literal comma). This is achieved by maintaining and possibly recreating an `Array` under the hood. To support this, we provide a linked-list compound value collection that resembles *cons* lists. Contrary to the vector, the class `VCons` in listing 16 does not have variable fields but two instance variables `car` and `cdr`. Since the Smalltalk-typical process of “create object → set instance variables” cannot work with compound values, we provide a constructor `car:cdr:` that uses the `VM` primitive to create the compound value from the given arguments. Note that, again, a fallback for non-value usage is provided.

When *cons* lists are used in Lisp-family languages, they are terminated by the empty list `nil`. We need such a terminator here, too, however, Squeak’s `nil` does not fit, as it is neither a compound value but rather a singleton nor can it act like a list. We provide a helper class `VNil` as in listing 17, that has no instance variables nor variable fields and hence is a niladic value. The sole constructor method `nil` creates these values, again with a non-value fallback. Since there are no constituents for this value, the equality check `=` is simply a class check. Finally, a `VNil` represents the empty list, so it reports this in `isEmpty`.

The Squeak collections work with a template method `do:` that the rest of the interface relies on. For `VCons`, the method as given in listing 16 makes use of a loop and an local variable that acts as an accumulator. While it would have been possible to use a recursive style, Squeak does not provide a tail-call optimization. In fact, most implementations of `do:` rely on an implicit loop not unlike the one we used here.

*RSqueak
values*

LISTING 16: *Cons* lists with Squeak compound values

```

1 SequenceableCollection subclass: #VCons instanceVariableNames: 'car cdr'
2 ----- VCons class -----
3 car: anObject cdr: anotherObject
4 <primitive: 'primitiveValueFromArgs' module: 'ValuePlugin'>
5 " Fallback for non-immutable "
6 ^ self basicNew
7     instVarNamed: 'car' put: anObject;
8     instVarNamed: 'cdr' put: anotherObject;
9     initialize
10 -----
11 withAll: aCollection
12     | list |
13     list := VNil nil.
14     aCollection reverseDo: [:each | list := self car: each cdr: list].
15     ^ list
16 -----
17 ----- VCons -----
18 = other
19     ^ other class = self class
20     and: [other car = self car
21         and: [other cdr = self cdr]]
22 -----
23 isEmpty
24     ^ false
25 -----
26 do: aBlock
27     | cons |
28     cons := self.
29     [cons isEmpty] whileFalse:
30         [cons isCons iffFalse: [^ self error: 'Not a proper list'].
31         aBlock value: cons car.
32         cons := cons cdr].
33 -----
34 , aCons
35     self cdr isCons iffFalse: [^ self error: 'Not a proper list'].
36     ^ self class
37         car: self car
38         cdr: self cdr, aCons

```

LISTING 17: Squeak compound value *cons* lists terminator

```

1 Object subclass: #VNil instanceVariableNames: "
2   _____ VNil class _____
3 nil
4   <primitive: 'primitiveValueFromArgs' module: 'ValuePlugin'>
5   " Fallback for non-immutable "
6   ^ self basicNew initialize
7
8   _____ VNil _____
9 = other
10  " value semantics "
11  ^ other class == self class
12  _____
13 isEmpty
14  ^ true

```

LISTING 18: Collect with Squeak compound value *cons* list.

```

1 _____ VCons _____
2 collect: aBlock
3   | cons acc |
4   cons := self.
5   acc := VNil nil.
6   [cons isEmpty] whileFalse:
7     [cons isCons iffFalse: [^ self error: 'Not a proper list'].
8     acc := self class car: (aBlock value: cons car) cdr: acc.
9     cons := cons cdr].
10  ^ acc reversed

```

LISTING 19: Map with Squeak compound value *cons* list.

```

1 _____ VCons _____
2 map: aBlock
3   self cdr isCons iffFalse: [^ self error: 'Not a proper list'].
4   ^ self class
5     car: (aBlock value: self car)
6     cdr: (self cdr map: aBlock)
7
8   _____ VNil _____
9 map: aBlock
10  ^ self

```

With these core objects and methods, we provide a convenience method to turn any sequence-able collection into a *cons* list in [listing 16](#), with `All`. Moreover, the message `,` (comma) can be used to append to the list. This is more complicated than with mutable objects as the last object in the link list cannot simply be modified, but this was expected. Also, this method uses a recursive style to be more readable than with about three loops necessary when not relying on recursion.

*Compound
values in
practice*

6.3.2 Paradigm impact

The predominant and idiomatic way to use collections in Smalltalk is with messages such as `select`: (find all matching), `collect`: (apply to each and store result), or `anySatisfy`: (does any object match) to name a few. These are always used with a block—that is, an anonymous function—which gets passed each of the collection’s content objects to work with. Explicit enumeration, for example by looping over an index, is unidiomatic and seldom. However, below this interface, there are typically loops that in some way iterate over the collection, particularly when `Arrays` or similar structures are used for implementing a collection. For collections that work like linked lists, this implies reference walking. In the case of `VCons` with compound values, this is unavoidable. An example for such a method, `collect`: can be found in [listing 18](#). It is very close to the method `do`: above with the exception of not throwing away the result of the applied block.

This is in contrast with the functional style employed in Racket and, in a way, in Theseus. Given tail-call optimization and the ubiquity of *cons* lists, most procedures or functions working on collections use a style of walking these lists using recursion. It is entirely possible to express this in idiomatic Smalltalk, too, as can be seen in [listing 19](#), but is not very common given its performance characteristics on the traditional Smalltalk `VMS`.

RSqueak implements the stack frame model of Squeak in a way that tries to at the same time benefit from RPython’s advanced stack handling with *virtualizables* as well as maintaining the *spaghetti* or *gc stack* [26] semantics of Smalltalk. This has the effect that the loop style is more efficient than the tail-call style in RSqueak, as well. We ran several small *ad hoc* benchmarks that

suggest that the impact of a tail-call style is less pronounced in RSqueak. We will stick with the first variant given.

6.3.3 Values in Squeak

The functionality presented focuses on collections and provides compound value variants for common fixed and variable length collections. These are, however, arguably not compound values in the pure sense as specified earlier in the work. There are a number of candidates in the default image or standard library of Squeak that have value characteristics and could benefit from our approach. First and foremost, certain numerical classes, including fractions, large integers, and — depending on circumstances — floating point numbers, could be represented as compound values with our optimization. Since none of these up to now support identity based comparison for the common use cases, there would be no need to take special care when implementing them using identity-less compound values. Squeak also provides abstractions for geometric points and for matrices, which both could benefit from our approach in the same way as numerical data or collections.

*RSqueak
values*

However, to assess whether our approach has an impact on the performance characteristics of the presented systems, we have to restrict ourselves to concepts common to all of them. Therefore, we have provided little more than necessary for the benchmarks used later in this work. We consider a more user-study-centered evaluation certainly necessary, provided that the results of the benchmarks show a reasonable performance gain as a foundation.

6.3.4 Findings

The changes that were necessary to bring compound values to RSqueak were slightly more elaborate compared with Pycket, due to the differences in architecture. Of the ≈ 1500 lines of source code added to RSqueak's code base, ≈ 830 were reused from Theseus without changes. The changes amount to about 6% of the Python code within the code base. In all other regards our findings match that of Pycket, including meeting of the first two challenges. For [challenge 3](#), we have not adapted the identity logic of RSqueak to take

compound values into account yet. However, contrary to Pycket, where we “hijacked” the structure data structure, compound values are a separate new concept in RSqueak with its own rules, and as with certain numbers, we can reasonably expect developers not to count on the identity of compound values. Atemporality and non-instantiatedness are as limited as in Pycket but likewise unlikely to be interfered with.

*Compound
values in
practice*

6.4 DISCUSSION

The effort necessary to provide the technical prerequisites for compound values in more general programming systems than Theseus within the RPython ecosystem is rather low. We have summarized the differences between the three systems with regard to the quality of certain aspects in [table 1](#).

a The amount of code changes necessary to bring our optimization to Pycket was comparatively low. The similarities in architecture between Theseus and Pycket helped in that. For RSqueak, the architectural differences required an alternative approach of integration, which was not as low-effort as for Pycket, but comparatively simple nonetheless.

b Pycket shows that integration with a more mature language is quite possible, especially when the architecture matches. Since both Theseus and Pycket implement a CEK machine, proper tail recursion [27] comes for free, which greatly helps with the nature of the compound value based *cons* lists that we uses in both systems. RSqueak shows that tail-call optimization is not a prerequisite for good compound values support. Even though the object model of Squeak seems to contradict certain assumptions of compound values, it is certainly possible to provide means for working with compound values efficiently. Moreover, the non-tail-call style predominant in Smalltalk seems not greatly influence the overall performance, as one could expect from the recursive nature of compound value *cons* lists.

c The ubiquitous immutability of data structures is only partially mirrored in Pycket, but using cells, this is rather simple to manage. Immutability is generally a new concept in RSqueak; however, in the restricted areas where we applied compound values, this was only a minor issue.

TABLE I: Qualitative differences of compound values in our three systems

		Theseus	Pycket	RSqueak
<i>a</i>	Ease of adaptation	n/a	✓✓	✓
<i>b</i>	Architecture fits compound values	✓✓	✓✓	✓
<i>c</i>	Anticipation of immutability	✓✓	✓	(✓)
<i>d</i>	Ease of avoidance	✗	✓	✓✓

d In Theseus it is by design not possible to have data structures without value semantics. This is naturally not the case for the other two systems. For Pycket, to avoid using our compound values, using structures must be avoided altogether. This means that in certain circumstances where our optimization is undesirable, it might be necessary to change code or even change Racket internals. For RSqueak, using its plug-in architecture and fallback mechanism means that it is possible to selectively avoid or disable the use of our compound value optimization, without any changes to source code.

Our extensions to Pycket and RSqueak are the first applications of our approach to general purpose programming systems and show that it is feasible to be implemented in these settings. They show little interference with the pre-existing systems and will be used in comparison with Theseus in the quantitative evaluation.

Discussion

SUMMARY

The findings from Theseus apply to other systems, as well. Due to matching architectures, core techniques could be reused directly for Pycket. Architectural differences hindered the adaptation for RSqueak only slightly, but its plug-in system helped the implementation.

7 Value optimization quantified

This chapter quantitatively assesses the performance of the implementations of our approach. We assess whether the shape recognition approach is preferable to manual transformation rules, determine what default parameters the recognition approaches should use, measure execution time and memory consumption in micro-benchmarks, and compare these measurements to other related programming systems. We begin by characterizing the benchmarking environment and methodologies.

7.1 BENCHMARK SETUP

To characterize the environment in which our performance evaluation takes place, we provide information on the hardware used to perform measurements, the software used support the measurements, and which programming system implementations are included in the performance evaluation. We outline our benchmarking measurements and give concrete values for parameters that influence the performance of our approach. We give a comprehensive characterization of the setup, including hardware, software, and compared implementation in [appendix B.1](#).

7.1.1 Hardware

The processor used was an Intel Xeon Gold 6148 (Skylake) at 2.4 GHz with 27.5 MB L3-cache; 1.48 TB of RAM were available. The system had four processors with 20 cores each, that is 80 in total, and 160 native threads through hyperthreading.

7.1.2 Software

We used a 64 bit Ubuntu 16.04 LTS (Xenial Xerus) as operating system. To carry out the benchmarks we used the *ReBench* framework [71]. When isolating execution to one core we used Linux *cgroups*. All benchmarks were run from a RAM disk. Swap-space was completely disabled.

The compared implementations will be given for each evaluation part individually. Details on versions and settings can be found in [appendix B.1.3](#). Our benchmarking code and infrastructure are publicly available [86].

7.2 SHAPE RECOGNITION ASSESSMENT

To assess whether our recognition approach is preferable to manually specifying shape transformation rules, we compare the execution times of both approaches. We intend to show that the overhead of our recognition approach is eventually set off against the optimization limit of the manual approach.

7.2.1 Methodology

Even in its early stages, the prototype did not support any kind of loops, but already included tail-call elimination [27]. Therefore, we ran several list operations on increasingly longer, large lists on the early prototype. That way, we could obtain a similar effect to running a benchmark for multiple iterations: after a certain list length, the shape recognition has found enough or more transformation rules to be as effective or better than the manual approach. The list length has hence been used as an iteration equivalent.

We ran four operations (reverse, append, map, filter) in three configurations: no optimization (*None*); optimization using our approach but only using ahead-of-time, manually specified transformation rules without using shape recognition (*Inlining only*); and optimization with transformation rules derived using shape recognition (*Recognition*).

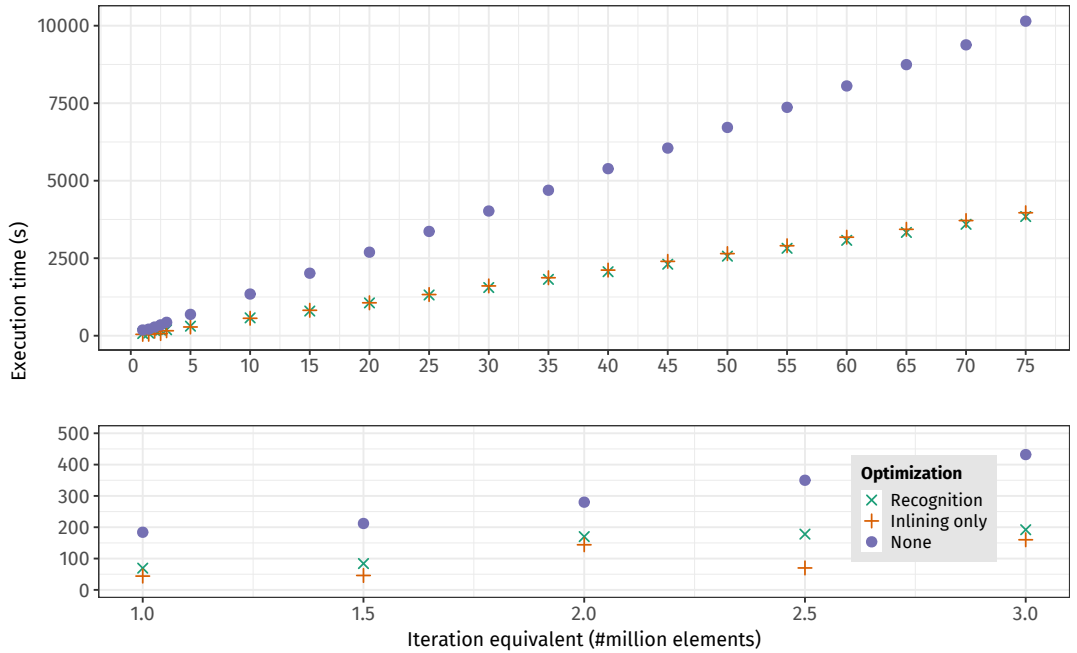


FIGURE 10: Execution times for reversing lists of different lengths. The length is used as an equivalent to iterations as typically used to assess warmup. *None*: no optimization. *Inlining only*: optimization with ahead-of-time, manually specified transformation rules; no shape recognition. *Recognition*: optimization with transformation rules derived using shape recognition.

7.2.2 Results

We provide the execution time results for reversing a long list in [figure 10](#). In this case, we found that

- both optimized versions are always faster than the unoptimized one,
- initially, the version with manually specified transformation rules is faster than the version with shape recognition, but
- for most data points, the version with shape recognition and transformation rule inference is as least as fast as the version with manually specified transformation rules.

*Value
optimization
quantified*

The results for other list operations (appending, mapping, filtering) were very similar and have therefore been omitted.

These findings validate the shape recognition approach, in the context of optimization, as preferable to specifying transformation rules manually. Accordingly, the possibility to manually specify transformation rules has been omitted from Theseus.

7.3 DERIVATION OF OPTIMIZATION CONFIGURATION

The shape recognition and inlining approach is parametrized by three configuration values as described in [section 3.5](#). These influence the measurements of our implementations. In this section, we show how we derived the default values used in the benchmarks described further in this chapter.

7.3.1 Substitution threshold

The purpose of the substitution threshold parameter is to delay the creation of transformation rules to a user-specified point in the execution. This is useful in case overspecialization is foreseeable. Moreover, developers may want to exclude certain compound values from optimization as they are known to occur a handful of times during execution but not so often as to warrant optimization. The latter is actually the case for Theseus in our benchmarks.

All data structures in Theseus are compound values and by default subject to our optimization approach. However, we use some of these data structures in a supporting manner, for example, to process command line arguments or timing results, to name a few. To avoid influencing the measurements by optimizing these data structures, we need a substitution threshold that is high enough for the supporting structures to not be reached by accident. We therefore chose the value 17, as next prime after the power-of-two 16. As laid out later, the data structures subject to benchmarking are several orders of magnitude larger, so this threshold will hardly influence our benchmarks.

*Derivation
of
optimization
configura-
tion*

SUBSTITUTION THRESHOLD We use a threshold of 17 shape occurrences.

7.3.2 Maximum object size and maximum shape depths

The maximum object size and maximum shape depth are parameters that interact with size and structure of compound values. The object size parameter guards against size explosion of optimized compound values, while the shape depth parameter guards against size explosion of derived shapes. On the one hand, this means low values can inhibit potential optimizations and impede making full use of resources. This can lead to worse-than-expected performance in terms of execution time and memory consumption. On the other hand, high values can result in over-optimization, that is compound values that take up a lot of memory and/or a high number of derived shapes. This, again, can lead to worse-than-expected performance in terms of execution time and memory consumption. It is hence crucial to find values for these parameters that avoid both bad-performance scenarios. We explain the process of deriving these parameters in the following.

Methodology To determine potential parameter values, we used a brute-force approach. We ran the five micro-benchmarks that we intended to benchmark thoroughly in a different setting (cf. [section 7.4.2](#) below).

We first determined the possible search space for the two parameters through short experiments with selected values of different size and magnitude. We use a search space with values between 2 and 23 for both parameters. Much higher

values indeed resulted in excessive execution time. The value 0 would have disabled our optimization and the value 1 would have favoured one variant of the benchmarks over the other. To rule out that one set of parameters works very well on one optimized implementation but not the others, we decided to use all three optimized implementations in the derivation process. In summary, we ran 5 benchmarks in 2 variants (numeric elements and niladic elements, see below) on 3 implementations with 22 values for the object size parameter and 22 values for the shape depths parameter, totaling in 14 520 benchmark executions.

Contrary to the main benchmark, we are not interested in definitive results here but rather a range of plausible values. The outcome of the following experiment is to ensure selection of default parameter values that do not result in undesirable performance characteristics for any of the three implementations. Therefore, we ran these benchmark executions in parallel, despite the possibility that they might slightly influence each other. For each execution, we measured execution time (*cpu*) and memory consumption (*resident set size*).

Analysis We first assessed whether a global optimum exists for our parameters. That is, is there a value for each parameter, that results in the shortest execution time and the lowest memory consumption for all combinations of benchmarks, variants, and implementations? **We found that there is no such global optimum.** Therefore, we opted to proceed with a process to eliminate unfavorable parameter values (refer to [appendix B.2](#) for all intermediate steps):

- i. For each benchmark, variant, and implementation, and for each measurement criterion (execution time, memory consumption):
 - [NORMALIZATION PROCESS]
 - a) Eliminate all parameter values that resulted in an excessive measurement, that is **five times larger** than the minimum for this set.
 - b) Normalize the measurements for the remaining values to a $[0; 1]$ range.

2. For each variant and measurement criterion:
 - a) Sum the normalized measurement values across all benchmarks and implementations.
 - b) Apply the NORMALIZATION PROCESS to the accumulated values.
3. For both variants:
 - a) Sum the normalized measurement values across both measurement criteria.
 - b) Normalize the result to a $[0; 1]$ range.
4. Multiply the normalized values of the normalized numeric elements variant **by three** and add the values from the normalized niladic elements variant. That way, we give a lower precedence to the niladic elements case, because we expect it to be far less valuable in real-world applications.
5. Normalize the result to a $[0; 1]$ range.

The result is a matrix that contains a value between 0 and 1 for every parameter value combination that was never responsible for an excessive measurement (five times larger than the minimum). Within these values, we identified the parameter values responsible for the 2% lowest values.

Results The analysis resulted in the following (shape depth, object size) pairs for the best 2% (cf. figure 11):

(9, 5); (9, 7); (9, 8); (9, 10);
 (10, 4); (10, 5); (10, 6); (10, 8).

From these, we chose the last pair.

MAXIMUM OBJECT SIZE We use a size of 8 fields.

MAXIMUM SHAPE DEPTH We use a depth of 10 shapes.

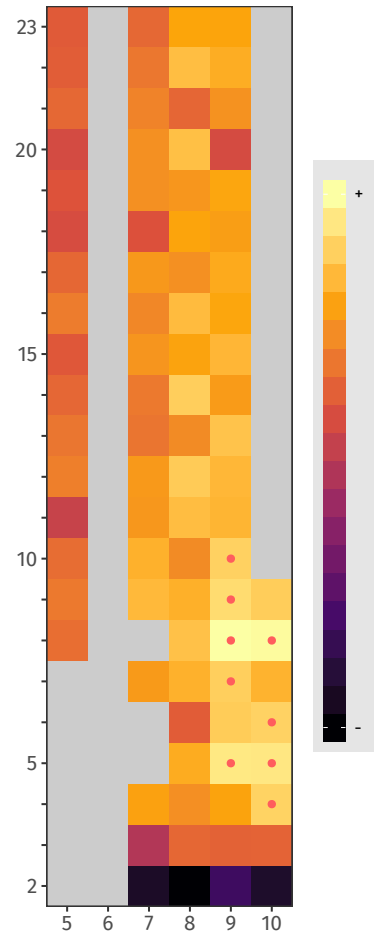








FIGURE 11: Results of elimination process. Red dots indicate best 2% within normalized measurements for given parameter values. +/Lighter is better. Parameters: x-axis: maximum shape depths; y-axis: maximum storage width.







7.4 COMPARATIVE MICRO-BENCHMARKS

We report the performance of five micro-benchmarks with their execution time and peak memory consumption to assess the quantitative improvements of our approach.

7.4.1 Compared implementations

Implementation

-  Theseus
-  Theseus (not optimized)
-  Pycket (optimized)
-  Pycket (original)
-  RSqueak (optimized)
-  RSqueak (original)

For the comparative micro-benchmarks, we included all implementations where we applied our optimization (Theseus , Pycket (optimized) , and RSqueak (optimized) ). Furthermore, to quantify the potential improvements of our approach, we included the original versions for Pycket (original)  and RSqueak (original) , and a version of Theseus (not optimized)  with our optimization disabled. This results in the set of six systems listed to the left.

Non-regression Our optimization should not influence anything except compound values. Since Theseus' sole data structure is compound values, a regression cannot occur. However, to ensure that there is no influence of our approach to Pycket and RSqueak besides compound values, we ran system-typical benchmarks for each of them.

For Pycket (optimized), we ran the *shootout* benchmarks described in the original paper on Pycket [10]. These benchmarks hardly make use of structures. On average, the execution time for these benchmarks deviates less than 6% from the original implementation. This low deviation shows that our approach has very little overhead when structs are not used.

For RSqueak (optimized), we ran the *tiny benchmarks* [60, “Performance and Optimization”] and found no statistically relevant deviation, as expected.

7.4.2 Benchmarks

The benchmarks chosen are append, filter, map, and reverse on very long linked lists and the creation and complete prefix traversal of a binary tree — inspired by the seminal GCBenchmark [40]. The benchmarks exist in two variants, each

operating on collection data structures containing either numeric elements or niladic elements (cf. [section 2.1.3](#)). As we expect different optimization characteristics for each variant, we include both — affixed with either $_n$ or $_E$.

The append and reverse benchmarks are straightforward, appending one long linked list of binary compound values to another and reversing such a list, respectively. The filter and map benchmarks use decidedly minuscule auxiliary functions to select from or transform a long linked list of binary compound values, respectively. Because we are interested in the performance of the “outer” function, not the auxiliary one, we ensure that their functionality is as small as possible. Lastly, the tree benchmark does not operate on binary compound values organized as linked list but rather a binary tree of ternary compound values (left, content, right), that is traversed in a certain fashion. This is a step to assess not only *cons* style data structures. The benchmarks are written in a style aiming to best suite both the idiomatic style and performance characteristics of each implementation. For Pycket/Racket, this means a predominantly tail-recursive style, for RSqueak/Squeak, this means object recursion where possible, and loops. Theseus has no established idiomatic style, but its architecture is close to Pycket’s and loops are not available at all, so it uses a predominantly tail-recursive style.

The relevant source code for all ten benchmarks — excluding measurement infrastructure — can be found in [appendix C.3](#)

7.4.3 Methodology

Every benchmark was run ten times uninterruptedly at highest priority, in a new process, and isolated to exactly one core.

The execution time (*cpu time*) was measured *in-system* and, hence, does not include start-up; however, warm-up was not separated, so JIT compiler execution time is included in the numbers, especially so since we cannot reasonably expect warm-up to settle in all cases [9]. We arranged, however, for the large lists to be created *before* measuring the execution time of the benchmark. The maximal memory consumption (*resident set size*) was measured *out-of-system* and may hence include set-up costs. We report the arithmetic mean of the ten runs together with the 95 % confidence interval.

Moreover, we report relative execution time and relative memory consumption, comparing the implementations using our optimization to the respective unoptimized one. In this case, we give bootstrapped [32] confidence intervals showing the 95 % confidence level, suggested as one variant for showing effect sizes rigorously [63, 64].

The length of the data structures for each benchmark was chosen so that the execution time of the *unoptimized* implementations takes more than 2 seconds but less than a minute.

7.4.4 Performance Results

Our approach leads to a reduction in execution time and memory consumption. We provide absolute measurements for both of these metrics as well as relative results that show the scale of reduction.

Absolute measurements In the top part of [figure 12](#), the execution time of all benchmarks is reported.⁴ In the unoptimized case (right part of every pair), RSqueak (original) is always slower than the other two implementations, at times more than an order of magnitude, while Theseus (not optimized) and Pycket (original) usually perform comparably — this can be attributed to their architectural similarities. However, in most cases, we see a partially substantial decrease of execution time for the optimized implementations (left part of every pair), with 90 % of the measurements in the range of 0.1 to 6 seconds. The 95 % confidence intervals are comparatively small, indicating a high representativeness of our measurements for these benchmarks.

For memory consumption (bottom part of [figure 12](#)), the unoptimized benchmark executions range from 1.4 to 12.2 GB for all but the tree benchmarks, where the range is 0.8 to 1.3 GB. As with execution time, the measurements for Theseus (not optimized) and Pycket (original) are more similar to each other than to RSqueak (original). Nevertheless, in all cases we see a reduced memory consumption in the optimized implementations, with 90 %

⁴The lighter shaded part of each execution time bar is time spent during garbage collection, included purely for informative purposes; details can be found in [appendix B.3](#)

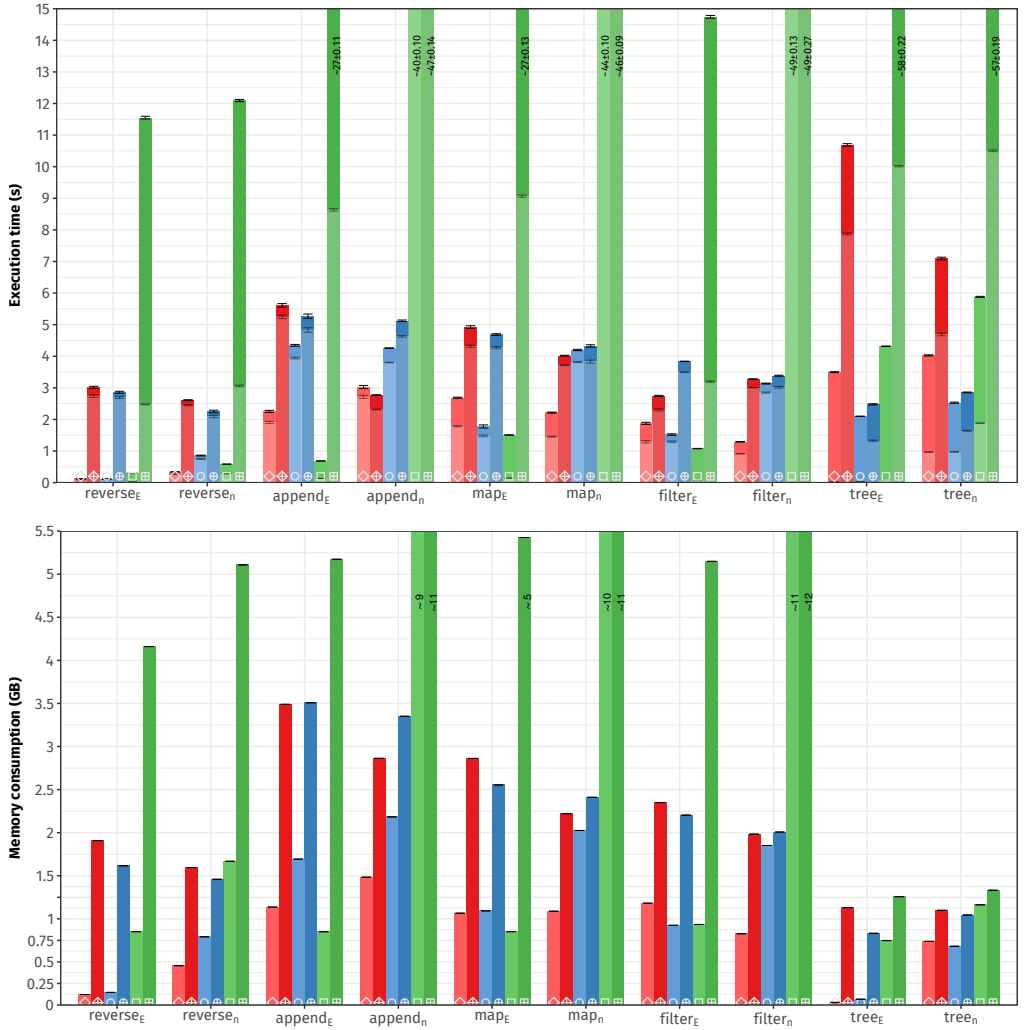


FIGURE 12: Benchmarking results. Bars show the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). We include 95 % confidence intervals. Lower is better. Raw numbers in [table 10](#) and [table 11](#).

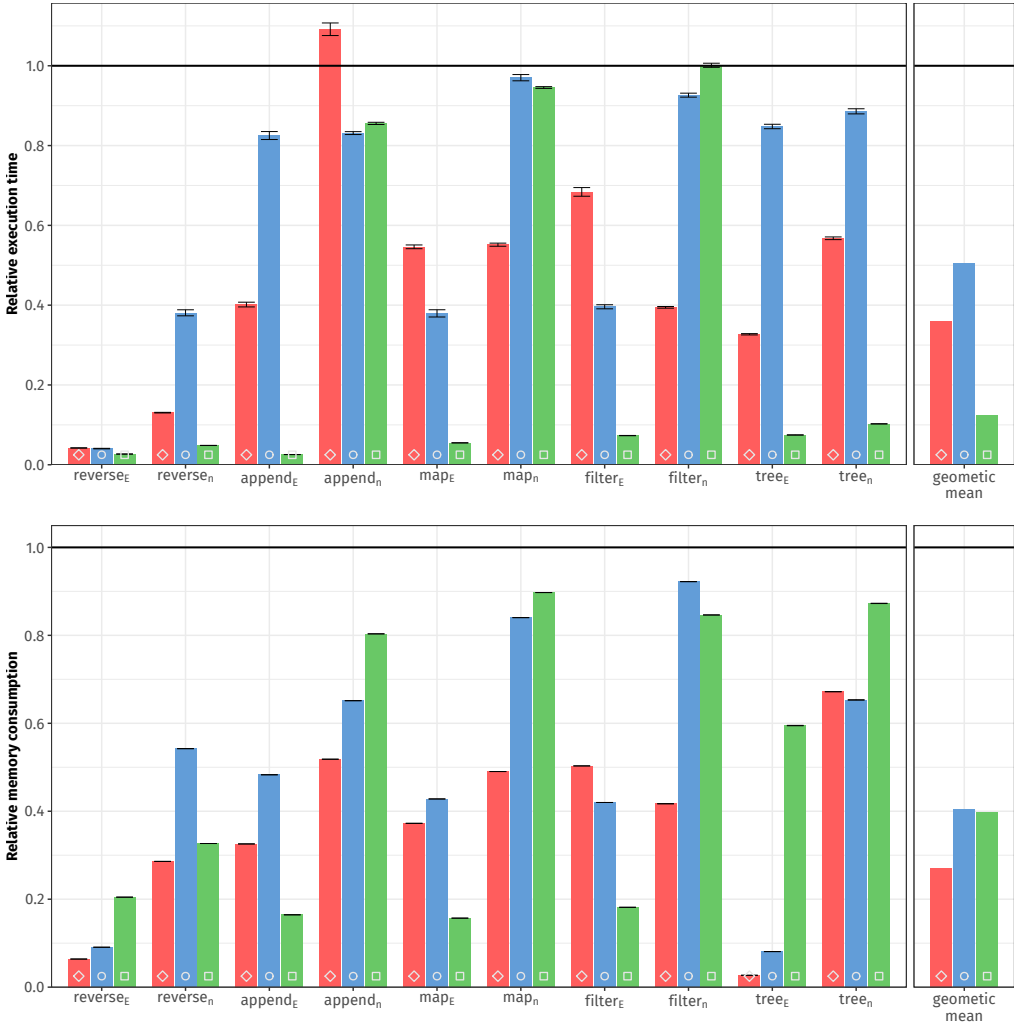


FIGURE 13: Relative benchmarking results. Bars show relative execution time (top) and memory consumption (bottom) between *each* optimized and non-optimized implementation. We include the geometric mean of all benchmarks to the right, as well as bootstrapped 95 % confidence intervals. Lower is better. Raw numbers in table 13 and table 14.

of the measurements in the range from 0.02 to 2.1 GB. In this case, the 95 % confidence intervals are minuscule, indicating a very high representativeness of our measurements for these benchmarks.

Relative results In [figure 13](#), we give *relative* results of our benchmarks. That is, we give the optimized implementations' *fraction* of execution time/memory consumption of the unoptimized implementations.

For execution time (top part of [figure 13](#)), we find a single case where one of our optimized implementations is demonstrably slower, and one case where a difference cannot be shown (but see [section 7.6](#)). In all other cases, we find often substantial speed-ups. For two thirds of the cases, these range from $\frac{1}{2}$ to 39 times faster. The bootstrapped confidence intervals allow us to trust these numbers to be representative for our benchmarks. For convenience, we include a geometric mean (cf. right of [figure 13](#)) of all relative runtimes, indicating a speed-up of about a factor of 2.7 for Theseus, 2 times for Pycket (optimized), and 8 times for RSqueak (optimized).

For memory consumption (bottom part of [figure 13](#)), we find no case where the optimized implementation consumes more memory than the unoptimized one. The memory saving is at least 8 %, and for 80 % of the measurements between 33 % and 97 % of the memory consumption can be saved. Again, the bootstrapped confidence intervals allow us to trust these numbers to be representative for our benchmarks. The geometric mean of all relative memory consumption indicate around 60 % saving for Pycket (optimized) and RSqueak (optimized), and more than 70 % for Theseus.

Analysis One key reason for our implementations' performance is the interaction between escape analysis and compressed storage. The benchmarks exhibit a certain usage pattern. In particular, the access to a list element is typically followed by inserting this element into a new list, with possibly processing it. The tracing JIT compiler and its escape analysis can infer that no reconstruction of the actual compound value is necessary and, furthermore, that a certain number of such operations occur consecutively. Operations can thus happen in blocks. For example, for a list that is inlined n levels deep, reverse can operate in chunks of n items. Given the chosen value of the maximum object size

*Comparative
micro-
benchmarks*

LISTING 20: reverse in Theseus. Tail-recursive implementation with accumulator and auxiliary function.

```
1 reverse$aux := λ.  
2   1. acc, Nil()      → acc  
3   2. acc, Cons(h, t) → μ(reverse$aux, Cons(h, acc), t)  
4 reverse := λ. l → μ(reverse$aux, Nil(), l)
```

*Value
optimization
quantified*

parameter — 8 fields, exclusive — we expect the inlining to result in chunks of 7 consecutive list elements and one next-chunk reference. This means that (a) six shape references and six next-element references can be saved per chunk, that is more than 50 %, and (b) the list operations can work on these chunks consecutively, comparable to what list unrolling [104] achieves. Moreover, the tracing JIT compiler can make assumptions about what stays constant within such a chunk. Thus it can remove almost all type checks, reduce the number of allocations to a minimum, has to follow fewer references, and reduce the overall number of operations the tracing JIT compiler processes by up to 60 %.

The comparative micro-benchmarks show that our approach can lead to significant performance improvements in execution time and memory consumption. Performance deteriorations are rare.

7.5 DISCUSSION OF reverse AS REPRESENTATIVE EXAMPLE

Since the micro-benchmarks are rather uniform, we take one representative example from the benchmarks and show in detail how the effects of our optimization approach can be observed in our implementations. The reverse benchmark is the shortest in code and execution time and will serve as example here. The reverse benchmark is simple. Given a singly-linked list, it returns the list in backwards order. The implementation used for Theseus and Pycket/Racket reflects the well-known, tail-recursive variant and can be found in

LISTING 21: reverse in Racket. Tail-recursive implementation with accumulator and auxiliary function.

```
1 (define (reverse l)
2   (define (loop a lst)
3     (if (null? lst)
4         a
5         (loop (cons (car lst) a) (cdr lst))))
6   (loop null l))
```

listing 20 for Theseus (cf. chapter 5) and in listing 21 for the Racket equivalent. Note that while the Racket code is presented here in the way it would be written idiomatically, using the library from listing 7, we can replace the standard Racket list operations with the optimized ones. The RSqueak/Squeak implementation (listing 22) is similarly typical for the language, using a loop with a temporary variable that accumulates the reverse list. Tail-recursion is not employed. While Smalltalk’s notion of stack frames is not dissimilar to that of the Lisp family and, similarly, is not bound to a stack limit, tail-recursion is absent in virtually all Smalltalk *vms*. Thus, while a close equivalent of the Theseus and Racket variant of the benchmarks is expressible in Squeak it would be neither idiomatic nor particularly fast. However, the loop found in the source is commonly encapsulated in the method `#do:`, which was essentially inlined here manually to provide a more self-contained benchmark.

For this discussion, we used a list with 200 000 000 elements and recorded the operations of reverse.

RPython JIT codes We re-use a functionality that all RPython-written *vms* provide, namely an online tracer that makes it possible to profile the RPython *JIT* compiler and examine its effects. The format of these “traces” is now introduced very briefly. The RPython *JIT* compiler represents its traces in an static single assignment (*ssa*) form. Some of its basic operations relevant here are memory operations, arithmetic operators, control flow operations, meta operations, and debug operations.

Discussion of reverse as representative example

LISTING 22: reverse in Squeak. Loop implementation with accumulator.

```
1  _____ VCons _____
2  reversed
3  | list cons |
4  list := VNil nil.
5  cons := self.
6  [cons isEmpty] whileFalse:
7    [cons isCons iffFalse: [^ self error: 'Not a proper list'].
8     list := self class car: cons car cdr: list.
9     cons := cons cdr].
10  ^ list
```

*Value
optimization
quantified*

MEMORY OPERATIONS Allocations (`new_with_vtable`, `new_array`), loads (for example `getfield_gc_r`), and stores (for example `setfield_gc`) are the relevant operations here.

ARITHMETIC OPERATIONS Only few operations are relevant to the traces here, such as subtraction (`int_sub`) or comparison (`int_eq`), to name a few.

CONTROL OPERATIONS There are mostly unconditional jumps (`jump`) and their targets (`label`), sometimes calls (`call`). As intended by tracing, branching is virtually absent in these traces.

META OPERATIONS In the absence of branching, *guards* are assertions that the state of certain values in the trace stays constant. For example, the operation `guard_class(value, class)` asserts that the value is an instance of the named class, and if not, execution control is transferred from the trace to the language interpreter, possibly with a rollback of the state.

The `force_token` operation is similar. It produces a tiny object used to manage so-called *virtualizables* of a stack frame, which facilitate an efficient mapping of stack frames directly to processor registers.

We exclude `guard_not_invalidated` from this category. It does not result in any operations being emitted and has virtually no effect in a loop. The guard serves as a placeholder so that—in the event that certain global assumptions do not hold any longer—a jump back into the interpreter can be patched into the trace. This is destructive and the trace is no longer usable after such a patch. We therefore can treat this specific guard as being absent during our analysis.

DEBUG OPERATIONS The `debug_merge_point` operation is special; it solely serves as an aid for the developer to identify which language-level operation happened when the surrounding operations were recorded by the tracer. They do *not* manifest themselves in machine code or execution at all. The same is true for the operations `enter_portal_frame` and `leave_portal_frame`, which helps profilers in locating the source stack frame of certain other operations but—again—do not appear in the final machine code. For that reason, these operations have been omitted from the trace.

Discussion of reverse as representative example

Control, arithmetic, and debug operations, and `guard_not_invalidated` will be subsumed under “other” in the remainder of this chapter.


The granularity of the trace outputs is per loop. For each loop, a trace can show several operations before the actual loop, which ranges from a label to a jump. These initial operations are actually the first iteration of the loop, which has been moved outside in an optimization called “loop peeling”. We will focus on the actual loops and omit the initial iteration.

**

The following cases are discussed:

1. Theseus, operating on a list of numeric elements.
2. Theseus, operating on a list of niladic elements.
3. Pycket, operating on a list of numeric elements.
4. Pycket, operating on a list of niladic elements.
5. RSqueak, operating on a list of numeric elements.
6. RSqueak, operating on a list of niladic elements.

The distinction between numeric elements and niladic elements as list contents is important to properly discuss the scope of our optimization. In our context, numeric elements are always represented using primitives of the `VM`. We therefore view them as natural optimization barrier, that is, these elements themselves are never subject to our optimization, but only the data structures that contain them. On the contrary, due to how shapes work, niladic elements

LISTING 23: Trace for reverse in Theseus (not optimized) , numeric elements. Loop without peeled iteration. Debug operations are omitted.

```

1 +460: label(p6, p3, p13, p14, p1, descr=TargetToken(4401759200))
2 +493: p19 = getfield_gc_r(p14, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize2.inst__storage_0 16 pure>)
3 +497: p20 = getfield_gc_r(p14, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize2.inst__storage_1 24 pure>)
4 +501: guard_nonnull_class(p19, ConstClass(W_Integer), descr=<Guard0x1065e05c0>)
5 +519: guard_nonnull_class(p20, 4392412976, descr=<Guard0x1065e0560>)
6 +538: p23 = getfield_gc_r(p20, descr=<FieldP lamb.model.W_Constructor.inst__shape 8 pure>)
7 +549: guard_value(p23, ConstPtr(ptr24), descr=<Guard0x1065e20b0>)
8 +558: p25 = new_with_vtable(descr=<SizeDescr 32>)
9 +595: setfield_gc(p25, ConstPtr(ptr26), descr=<FieldP lamb.model.W_Constructor.inst__shape 8 pure>)
10 +599: setfield_gc(p25, p6, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize2.inst__storage_0 16 pure>)
11 +603: setfield_gc(p25, p3, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize2.inst__storage_1 24 pure>)
12 +607: jump(p13, p25, p19, p20, p1, descr=TargetToken(4401759200))

```

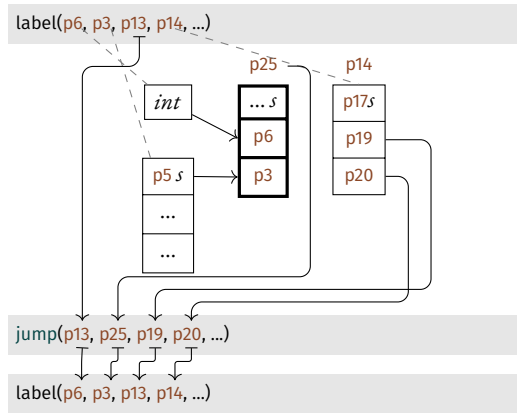
are subject to our approach and can be optimized away and only present via shape information. From the point of view of optimizability, the variants thus serve as opposing ends of the spectrum of data that can constitute a compound value in our system.

7.5.1 Theseus and numeric elements list

For the first two traces, we explain their workings in detail. The other cases, while with distinct parts, share enough structure that only the differences need mention.

Unoptimized trace We first present the “unoptimized” trace, that is, we disabled all means that make the recognition of shapes and the inlining happen. That way, the shapes are mere data structure descriptors and all VM-level storage elements directly match their language-level counterparts.


The main loop (listing 23) has 19 operations, but 7 of these are debug operations. This leaves 12 operations to consider. These operations represent the effect of reverse\$aux (cf. listing 20), and there the second case. This is a recursive case, so it is safe to assume that the variable configuration always has a constructor with a two-field storage in p14 (the tail), the current element in p6, and the accumulator list in p3, the element of the next iteration in p13. Right after the entry label, p14 is deconstructed in its two constituents, p19 for



Discussion of reverse as representative example

FIGURE 14: Unoptimized trace for the prototype. Variable configuration flow (\mapsto) and allocations (bold frame) show that one element is processed per iteration.

the head and `p20` for the tail. The following guards ensure that the element is a language-level integer (cf. line 4) and the tail has a known class, namely the same as `p14` (cf. line 5). Then the shape of the tail is accessed and checked to be the expected one. Note that this is a *value guard*, that is, the JIT compiler expects a certain object to be found, not just that it is of a certain (VM-level) class. This means that the JIT compiler is aware that closely related compound values share the same shape and that the JIT compiler does indeed speculate on that. Finally, the allocation of the new constructor happens (`p25`), it is filled with its shape and `p6` as element and `p3` as tail. Note that these two references were *not* obtained in the current loop iteration but in the previous one or the peeled loop on the first iteration. The loop closes with a jump to the loop label in line 12, and the variable configuration now consists of the previous next-iteration element `p3`, the new accumulator `p25`, the new next-iteration element `p19` and the new tail `p20` — both obtained from `p14`. This flow of data is visualized in figure 14. An individual element flows from `p14`'s first element in two iterations via `p19` \mapsto `p13` \mapsto `p6` into the storage of a new value `p25`.

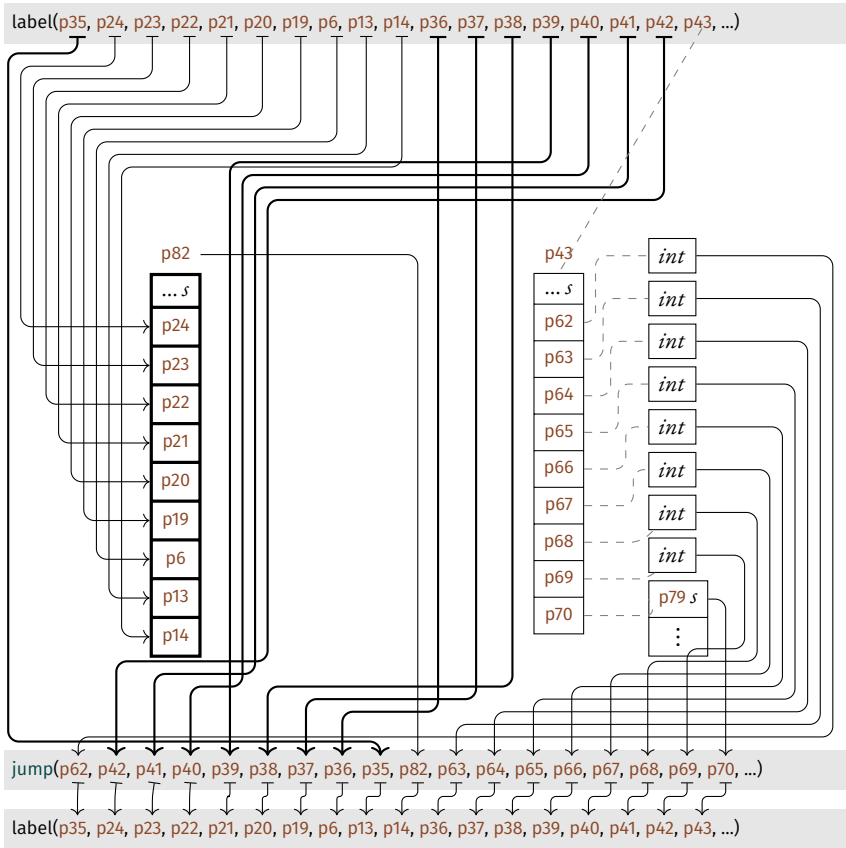
LISTING 24: Trace for reverse in Theseus , numeric elements. Loop without peeled iteration. Debug operations are omitted.

```

1 +1098: label(p35, p24, p23, p22, p21, p20, p19, p6, p13, p14, p36, p37, p38, p39, p40, p41, p42, p43, p1, descr=TargetToken(6697599888))
2 +1117: guard_class(p35, ConstClass(W_Integer), descr=<Guard0x10841e800>)
3 +1129: guard_class(p38, ConstClass(W_Integer), descr=<Guard0x18f362608>)
4 +1142: guard_class(p39, ConstClass(W_Integer), descr=<Guard0x18f3625c0>)
5 +1154: guard_class(p40, ConstClass(W_Integer), descr=<Guard0x18f362578>)
6 +1167: guard_class(p41, ConstClass(W_Integer), descr=<Guard0x18f362530>)
7 +1180: guard_class(p42, ConstClass(W_Integer), descr=<Guard0x18f3624e8>)
8 +1194: guard_nonnull_class(p37, ConstClass(W_Integer), descr=<Guard0x10841e860>)
9 +1212: p62 = getfield_gc_r(p43, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_0_16 pure>)
10 +1216: p63 = getfield_gc_r(p43, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_1_24 pure>)
11 +1220: p64 = getfield_gc_r(p43, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_2_32 pure>)
12 +1231: p65 = getfield_gc_r(p43, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_3_40 pure>)
13 +1242: p66 = getfield_gc_r(p43, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_4_48 pure>)
14 +1253: p67 = getfield_gc_r(p43, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_5_56 pure>)
15 +1264: p68 = getfield_gc_r(p43, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_6_64 pure>)
16 +1275: p69 = getfield_gc_r(p43, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_7_72 pure>)
17 +1286: p70 = getfield_gc_r(p43, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_8_80 pure>)
18 +1297: guard_class(p63, ConstClass(W_Integer), descr=<Guard0x18f3624a0>)
19 +1309: guard_class(p64, ConstClass(W_Integer), descr=<Guard0x18f362458>)
20 +1321: guard_class(p65, ConstClass(W_Integer), descr=<Guard0x18f362410>)
21 +1333: guard_class(p66, ConstClass(W_Integer), descr=<Guard0x18f3623c8>)
22 +1346: guard_class(p67, ConstClass(W_Integer), descr=<Guard0x18f362380>)
23 +1358: guard_class(p68, ConstClass(W_Integer), descr=<Guard0x18f362338>)
24 +1371: guard_class(p69, ConstClass(W_Integer), descr=<Guard0x18f3622f0>)
25 +1384: guard_class(p70, 4423682536, descr=<Guard0x18f3622a8>)
26 +1398: p79 = getfield_gc_r(p70, descr=<FieldP theseus.model.W_Constructor.inst__shape_8 pure>)
27 +1424: guard_value(p79, ConstPtr(ptr80), descr=<Guard0x18f362260>)
28 +1433: guard_nonnull_class(p62, ConstClass(W_Integer), descr=<Guard0x10841e7a0>)
29 +1451: p82 = new_with_vtable(descr=<SizeDescr 88>)
30 +1488: setfield_gc(p82, ConstPtr(ptr83), descr=<FieldP theseus.model.W_Constructor.inst__shape_8 pure>)
31 +1492: setfield_gc(p82, p24, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_0_16 pure>)
32 +1496: setfield_gc(p82, p23, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_1_24 pure>)
33 +1500: setfield_gc(p82, p22, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_2_32 pure>)
34 +1504: setfield_gc(p82, p21, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_3_40 pure>)
35 +1515: setfield_gc(p82, p20, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_4_48 pure>)
36 +1526: setfield_gc(p82, p19, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_5_56 pure>)
37 +1537: setfield_gc(p82, p6, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_6_64 pure>)
38 +1548: setfield_gc(p82, p13, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_7_72 pure>)
39 +1559: setfield_gc(p82, p14, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize9.inst__storage_8_80 pure>)
40 +1570: jump(p62, p42, p41, p40, p39, p38, p37, p36, p35, p82, p63, p64, p65, p66, p67, p68, p69, p70, p1, descr=TargetToken(6697599888))

```

Per iteration, there is one allocation and three stores into the newly allocated storage. Three accesses/loads obtain the next iterations' elements and a shape to assert its identity. The left part of table 2 summarizes these numbers. We exclude debug and meta operations, because their effective machine code is empty or very lightweight. In this specific loop arithmetic operations and control operations have only very little impact. The memory operations dominate. Combined with the iteration count, about 2 billion relevant operations are



Discussion of reverse as representative example

FIGURE 15: Optimized trace for Theseus. Variable configuration flow (\leftrightarrow) and allocations (bold frame) show that 8 elements are processed per iteration.

executed for the whole loop, which matches our expectations. This amount forms the baseline to compare the other traces against.

Optimized trace The second trace was obtained by running Theseus with the parameters found in section 7.3. We expect to see inlining happen and VM-level storage to be larger than the language-level counterparts.

The main loop of the second trace (listing 24) comprises 89 operations, 49 of which are debug information, which leaves 40 operations to consider (see figure 15 for a graphical depiction). While the data processed and the program traced are the same as in the unoptimized trace, however, the loop-local variable configuration is considerably larger. We still have the “tail” in the second-to last position (p43) in the label description, but all other positions before that are actually elements obtained through inlining. Accordingly, after asserting that the last iterations’ contents are of the expected class (lines 2 to 8), the contents of the tail are extracted into p62 through p70 (lines 9 to 17) and it is checked that seven elements are boxed integers, as expected (the eights is checked in the next iteration). Then, the new tail in p70 is checked for its class and, as in the unoptimized trace, its shape is extracted and checked (lines 25 to 27). Similar to the unoptimized trace, a new compound value is created and filled with the shape, yet, its storage is filled *in reverse order* with the contents of the last iteration (lines 29 to 39). We see a similar flow in the variables — for example, p63 \mapsto p36 \mapsto p6 into the storage of a new value p82 — but now for eight elements instead of just one. This means that the actual reversal happens neither during the loads from the object at hand (p43) nor during the creation and filling of the new object (p82). Rather, as shown by the thicker arrows in figure 15, during the passthrough of the variables containing last iterations’ read object contents (p35, and p36 through p42) to the next iterations’ written object contents (p24 through p19, p6, p13, and p14), the order of the variables changes from increasing in the label to decreasing in the *jump*. Thus, the tracing JIT compiler has practically created a pipeline. Contents that are read in one iteration are reversed in the next iteration and stored in the next but one iteration.

All in all, there are one allocation and ten stores into the newly allocated storage per iteration. Likewise ten accesses or loads happen to obtain the next iterations’ elements as well as one other shape. The right part of table 2 summarizes these numbers. The memory operations still dominate, but less so. About one billion relevant operations in total are executed.

Trace relation Comparing the two traces table 2 reveals that, while the number of operations per loop iteration has increased in nearly all categories, the

TABLE 2: Operation counts and relation for optimized and unoptimized reverse in Theseus, numeric elements. Reduction factors are for run total.


Loop iterations	Listing 23 199 998 931			Listing 24 24 998 901			Reduction Factor 8.00
	per loop	once	run total	per loop	once	run total	
Allocations	1	0	199 998 931	1	0	24 998 901	8.00
Stores	3	0	599 996 793	10	0	249 989 010	2.40
Loads	3	8	599 996 801	10	26	249 989 036	2.40
Meta	3	9	599 996 802	17	29	424 981 346	1.41
Other	9	8	1 799 990 387	51	51	1 299 942 852	1.39
Operations	19	25	3 799 979 714	89	106	2 224 902 295	1.71
– Other	10	17	1 999 989 327	38	55	949 958 293	2.11

Discussion of reverse as representative example

total number of loop operations has fallen drastically, by a factor of 8. Even more so, the number of allocations per iteration remained the same, with the overall effect that total allocations also are 8 times fewer. For the operation categories where the numbers increased, that is actually set off by the lower iteration count, with reductions from 1.39 to 2.4. Overall, we see around half the operations in the optimized traces. However, the number of memory operations has been reduced much more, and since these disproportionately contribute to execution time and memory consumption, we see speed ups and memory savings much higher than a factor two. The inlining of up to seven elements is nicely visible in the trace and the iteration count.

Warm-up The first trace executes one application of reverse to the head of the list. Therefore, the 199 998 931 loop iterations plus one peeled were all spent on one element of the list each. After the first 1068 elements, the JIT compiler apparently reached a steady state.

The second trace is inlined, and we see 10 stores. The first is for the shape and the last for the reference to the next chunk of the list. This leaves 8 elements to store in the trace, which means that $(24\,998\,901 + 1) \cdot 8$ elements were processed using the second trace. This leaves 8784 elements processed before using this trace, showing that the warm-up phase increased eightfold.

LISTING 25: Trace for reverse in Pycket (original) , numeric elements. Loop without peeled iteration. Debug operations are omitted.


```
1 +624: label(p10, p19, p24, p26, p1, p14, p8, p21, descr=TargetToken(4585693296))
2 +669: p28 = getfield_gc_r(p19, descr=<FieldP pycket.values_struct.W_Struct.inst__type 8 pure>)
3 +680: guard_value(p28, ConstPtr(ptr29), descr=<Guard0x1114ae0e0>)
4 +689: p30 = getfield_gc_r(p19, descr=<FieldP rpython.tool.pairtype.W_StructSize2.inst__storage_1_24 pure>)
5 +693: guard_class(p30, 4540421576, descr=<Guard0x111569730>)
6 +705: p32 = getfield_gc_r(p19, descr=<FieldP rpython.tool.pairtype.W_StructSize2.inst__storage_0_16 pure>)
7 +709: guard_class(p32, 4540600168, descr=<Guard0x1115696e8>)
8 +722: guard_not_invalidated(descr=<Guard0x111569610>)
9 +722: p34 = new_with_vtable(descr=<SizeDescr 32>)
10 +759: setfield_gc(p34, ConstPtr(ptr35), descr=<FieldP pycket.values_struct.W_Struct.inst__type 8 pure>)
11 +763: setfield_gc(p34, p26, descr=<FieldP rpython.tool.pairtype.W_StructSize2.inst__storage_1_24 pure>)
12 +767: setfield_gc(p34, p24, descr=<FieldP rpython.tool.pairtype.W_StructSize2.inst__storage_0_16 pure>)
13 +771: jump(p10, p30, p32, p34, p1, p14, p8, p21, descr=TargetToken(4585693296))
```

7.5.2 Pycket and numeric elements list

The main loop of reverse in Pycket (original) is very similar to that of Theseus (not optimized). The differences in the trace (listing 25) are one additional `guard_not_invalidated` and the order of certain guards and loads. The one additional guard results in no machine code and is negligible. Therefore, we can treat these loops as identical, especially with respect to memory operations. This can also be seen in the left part of table 3. However, the number of loop iterations is higher for Pycket, which indicates less warm-up; only 133 elements are not processed by the loop.

Regarding the trace (listing 26) of Pycket (optimized), we can conclude that it is virtually the same as that of Theseus. Indeed, if the lightweight guards are removed, the traces are identical. The numbers of operations per loop as in table 3 are also very similar. We see that $(24\,999\,810 + 1) \cdot 8$ elements are processed by the loop, leaving 1512 elements to be processed during warm-up, which is also considerably less than for Theseus. However, compared with the unoptimized variant, warm-up has increased more than tenfold.

The architectural similarities between Theseus and Pycket seem to alleviate the differences in complexity. For this benchmark, the approach as applied to Theseus is equivalent to the approach as applied to Pycket.

LISTING 26: Trace for reverse in Pycket (optimized) , numeric elements. Loop without peeled iteration. Debug operations are omitted.

```

1  +1434: label(p7, p55, p56, p57, p58, p59, p60, p61, p62, p54, p34, p33, p32, p31, p30, p29, p28, p48, p49, p1, p17, p22, p15, descr=TargetToken(4530350464))
2  +1453: guard_class(p56, 4484119496, descr=<Guard0x10e0483e0>)
3  +1465: guard_class(p57, 4484119496, descr=<Guard0x10e07dbb0>)
4  +1478: guard_class(p58, 4484119496, descr=<Guard0x10e07db68>)
5  +1490: guard_class(p59, 4484119496, descr=<Guard0x10e07db20>)
6  +1504: guard_class(p60, 4484119496, descr=<Guard0x10e07dad8>)
7  +1518: guard_class(p61, 4484119496, descr=<Guard0x10e07da90>)
8  +1531: guard_class(p55, 4484119496, descr=<Guard0x10e07da48>)
9  +1543: guard_not_invalidated(descr=<Guard0x10e07da00>)
10 +1543: guard_class(p54, 4484119496, descr=<Guard0x10e07d9b8>)
11 +1555: p82 = getfield_gc_r(p62, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_0 16 pure>)
12 +1559: p83 = getfield_gc_r(p62, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_1 24 pure>)
13 +1563: p84 = getfield_gc_r(p62, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_2 32 pure>)
14 +1574: p85 = getfield_gc_r(p62, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_3 40 pure>)
15 +1585: p86 = getfield_gc_r(p62, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_4 48 pure>)
16 +1596: p87 = getfield_gc_r(p62, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_5 56 pure>)
17 +1607: p88 = getfield_gc_r(p62, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_6 64 pure>)
18 +1618: p89 = getfield_gc_r(p62, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_7 72 pure>)
19 +1629: p90 = getfield_gc_r(p62, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_8 80 pure>)
20 +1640: guard_class(p83, 4484119496, descr=<Guard0x10e07de80>)
21 +1653: guard_class(p84, 4484119496, descr=<Guard0x10e07de38>)
22 +1665: guard_class(p85, 4484119496, descr=<Guard0x10e07ddf0>)
23 +1677: guard_class(p86, 4484119496, descr=<Guard0x10e07dda8>)
24 +1689: guard_class(p87, 4484119496, descr=<Guard0x10e07dd60>)
25 +1702: guard_class(p88, 4484119496, descr=<Guard0x10e07dd18>)
26 +1714: guard_class(p89, 4484119496, descr=<Guard0x10e07dcd0>)
27 +1728: guard_class(p90, 4484103704, descr=<Guard0x10e07dc88>)
28 +1742: p99 = getfield_gc_r(p90, descr=<FieldP pycket.values_struct.W_Struct.inst__shape 8 pure>)
29 +1767: guard_value(p99, ConstPtr(ptr100), descr=<Guard0x10e07dc40>)
30 +1776: guard_class(p82, 4484119496, descr=<Guard0x10e07dbf8>)
31 +1788: p102 = new_with_ytable(descr=<SizeDescr 88>)
32 +1835: setfield_gc(p102, ConstPtr(ptr103), descr=<FieldP pycket.values_struct.W_Struct.inst__shape 8 pure>)
33 +1839: setfield_gc(p102, p34, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_0 16 pure>)
34 +1843: setfield_gc(p102, p33, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_1 24 pure>)
35 +1847: setfield_gc(p102, p32, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_2 32 pure>)
36 +1858: setfield_gc(p102, p31, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_3 40 pure>)
37 +1869: setfield_gc(p102, p30, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_4 48 pure>)
38 +1880: setfield_gc(p102, p29, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_5 56 pure>)
39 +1891: setfield_gc(p102, p28, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_6 64 pure>)
40 +1902: setfield_gc(p102, p48, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_7 72 pure>)
41 +1913: setfield_gc(p102, p49, descr=<FieldP rpython.tool.pairtype.W_StructSize9.inst__storage_8 80 pure>)
42 +1924: jump(p7, p83, p84, p85, p86, p87, p88, p89, p90, p82, p61, p60, p59, p58, p57, p56, p55, p54, p102, p1, p17, p22, p15, descr=TargetToken(4530350464))

```

TABLE 3: Operation counts and relation for optimized and unoptimized reverse in Pycket, numeric elements. Reduction factors are for run total.

Loop iterations	Listing 25 199 999 866			Listing 26 24 999 810			Factor 8.00
	per loop	once	run total	per loop	once	run total	
Allocations	1	0	199 999 866	1	0	24 999 810	8.00
Stores	3	0	599 999 598	10	0	249 998 100	2.40
Loads	3	12	599 999 610	10	34	249 998 134	2.40
Meta	3	13	599 999 611	18	37	449 996 617	1.33
Other	11	11	2 199 998 537	60	60	1 499 988 660	1.47
Operations	21	35	4 199 997 221	99	131	2 474 981 321	1.70
– Other	10	23	1 999 998 683	39	71	974 992 661	2.06

*Value
optimization
quantified*

7.5.3 RSqueak and numeric elements list

The trace of the main loop for RSqueak (original) is different than the two other unoptimized traces (cf. [listing 27](#)). First, it is much longer, and second, it contains more diverse operations. This is mainly due to the different execution model of Squeak on the one hand and the implementation approach taken by RSqueak on the other hand. Stack frames are modeled much more directly, partly as their content has to be accessible for developers via reflection. Using a RPython technique called *virtualizables*, these stack frame contents are mapped directly to processors wherever possible, which is reflected by `force_token` operations and certain operations relating to “portal frames” — these have however been omitted from [listing 27](#) as they are essential debug operations.

Moreover, RSqueak uses a specific approach to represent its objects called storage strategies, which focuses on specializing mutable object contents to homogeneous arrays [91]. The effect is that an additional array is necessary for a common RSqueak object. This is reflected in the trace by the fact that we see three allocations. One for the actual binary compound value ([line 23](#)), one for its contents array ([line 24](#)), and one for the integer (`i61`) that is part of the compound value ([line 25](#)). The latter is necessary, as the JIT compiler tried to reduce boxing and unpacked this integer in the previous iteration:

TABLE 4: Operation counts and relation for optimized and unoptimized reverse in RSqueak, numeric elements. Reduction factors are for run total.


Loop iterations	Listing 27 199 991 649			Listing 28 24 991 583			Factor 8.00
	per loop	once	run total	per loop	once	run total	
Allocations	3	0	599 974 947	1	0	24 991 583	24.01
Stores	7	1	4 199 824 630	18	8	449 848 502	9.34
Loads	6	36	3 599 849 718	10	60	249 915 890	14.40
Meta	12	21	7 199 699 385	18	47	449 848 541	16.00
Other	45	50	26 998 872 665	156	157	3 898 687 105	6.93
Operations	73	108	43 798 171 239	203	272	5 073 291 621	8.63
– Other	28	50	16 799 298 574	47	115	1 174 604 516	14.30

Discussion of reverse as representative example

p88 is our compound value, its storage array p100 is extracted in line 5 and following, from where the integer i104 is extracted in line 8. This is passed via jump and label to the next iteration, where it is boxed again. Additionally, a call in line 20, which is practically a bounds check, complicates this trace, as leaving and returning to a trace can impede execution time.

An architectural peculiarity of RSqueak is that the actually interactive and user-oriented system design requires means for interrupts. The original Squeak VM and its descendants check possible interrupt sources by polling after the execution of certain bytecodes, which is undesirable in the architecture of RSqueak. The VM therefore maintains an *interrupt check counter*, which is checked and decremented after execution said bytecodes. Only when this counter hits zero, interrupt sources are polled, and the counter is re-primed, commonly to 10 000. We can see this check in lines 19 to 22.


The boxing and checking results in more memory operations than compared with Theseus and Pycket, as can be seen in the left of table 4. The number of loads for the first peeled operation is notably high, as the first thing that is done when entering the whole trace is mapping the content of the heap-allocated stack frame object to local variables — and ultimately processor registers. The number of debug operations is also higher. All in all, the RSqueak version takes about eight times the operations Theseus takes.

LISTING 27: Trace for reverse in RSqueak (original) , numeric elements. Loop without peeled iteration. Debug operations are omitted.

```
1 +1104: label(p0, p2, p3, p4, i7, p8, p9, i61, p12, p88, p22, p24, p26, p28, p30, p32, p34, p36, p38, p40, p42, i93, descr=TargetToken(5123964960))
2 +1133: guard_not_invalidated(descr=<Guard0x12e972ec0>)
3 +1133: p98 = getfield_gc_r(p88, descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
4 +1144: guard_value(p98, ConstPtr(ptr99), descr=<Guard0x12e972f20>)
5 +1153: p100 = getfield_gc_r(p88, descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_storage 16>)
6 +1157: p102 = getarrayitem_gc_r(p100, 0, descr=<ArrayP 8>)
7 +1161: guard_nonnull_class(p102, 4562587528, descr=<Guard0x12e973040>)
8 +1179: i104 = getfield_gc_i(i102, descr=<FieldS rsqueakvm.model.numeric.W_SmallInteger.inst_value 8 pure>)
9 +1183: p105 = force_token()
10 +1186: p108 = force_token()
11 +1186: i112 = int_ne(i104, 9223372036854775807)
12 +1199: guard_true(i112, descr=<Guard0x12e1922f0>)
13 +1205: p114 = force_token()
14 +1215: i119 = call_i(ConstClass(ll_fixed_length_arrayPtr), ConstPtr(ptr118), descr=<Calli 8 r EF=2>)
15 +1256: i121 = int_ge(0, i119)
16 +1260: guard_false(i121, descr=<Guard0x12e9730a0>)
17 +1266: p125 = getarrayitem_gc_r(p100, 1, descr=<ArrayP 8>)
18 +1270: guard_nonnull_class(p125, ConstClass(W_PointersObject), descr=<Guard0x12e973100>)
19 +1288: i128 = int_sub(i93, 1)
20 +1299: setfield_gc(ConstPtr(ptr129), i128, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
21 +1303: i131 = int_le(i128, 0)
22 +1307: guard_false(i131, descr=<Guard0x12e973160>)
23 +1313: p132 = new_with_vtable(descr=<SizeDescr 32>)
24 +1350: p134 = new_array_clear(2, descr=<ArrayP 8>)
25 +1369: p135 = new_with_vtable(descr=<SizeDescr 16>)
26 +1380: setfield_gc(p135, i61, descr=<FieldS rsqueakvm.model.numeric.W_SmallInteger.inst_value 8 pure>)
27 +1384: setarrayitem_gc(p134, 0, p135, descr=<ArrayP 8>)
28 +1388: setarrayitem_gc(p134, 1, p12, descr=<ArrayP 8>)
29 +1399: setfield_gc(p132, ConstPtr(ptr138), descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
30 +1411: p139 = getfield_gc_r(p3, descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
31 +1415: setfield_gc(p132, p134, descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_storage 16>)
32 +1419: setfield_gc(p132, 0, descr=<FieldS rsqueakvm.model.base.W_AbstractObjectWithIdentityHash.inst_hash 8>)
33 +1427: guard_value(p139, ConstPtr(ptr156), descr=<Guard0x12e1a2080>)
34 +1443: p141 = getfield_gc_r(ConstPtr(ptr153), descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
35 +1454: guard_value(p141, ConstPtr(ptr155), descr=<Guard0x12e1a20e0>)
36 +1470: p142 = getfield_gc_r(ConstPtr(ptr163), descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
37 +1481: guard_value(p142, ConstPtr(ptr165), descr=<Guard0x12e1a2140>)
38 +1490: jump(p0, p2, p3, p4, i7, p8, p9, i104, p132, p125, p22, p24, p26, p28, p30, p32, p34, p36, p38, p40, p42, i128, descr=TargetToken(5123964960))
```

The higher complexity of RSqueak also results in longer warm-up. About 8350 elements of the list are not processed within the given loop.

The trace of RSqueak (optimized) is seemingly complex (listing 28). However, the inlining and block-by-block operation is nicely visible. Lines 28 to 47 extract and check the content of an seven times inlined compound value and lines 52 to 62 creates and fills a new compound value in reverse order. Reviewing the variable configuration of the label and jump shows that the same pipelining happens as with Theseus. The different representation of compound values also pays off. All array operations have been removed and the unboxing–boxing roundtrip also disappeared, leaving the trace much more

LISTING 28: Trace for reverse in RSqueak (optimized) , numeric elements. Loop without peeled iteration. Debug operations are omitted.

```

1 +2048: label(p0, p2, p3, p4, i7, p8, p9, p142, p93, p92, p91, p90, p89, p88, p55, p82, p83, p144, p145, p146, p147, p148, p149, p150, p151, p22, p24, p26, p28, p30, p32, p34, p36, p38,
   p40, p42, p69, i163, descr=TargetToken(6017035024))
2 +2109: guard_not_invalidated(descr=<Guard0x166a48da0>)
3 +2109: i168 = int_sub(163, 1)
4 +2134: i173 = int_sub(168, 1)
5 +2138: setfield_gc(ConstPtr(ptr174), i173, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
6 +2142: i176 = int_le(i173, 0)
7 +2146: guard_false(i176, descr=<Guard0x166a48c20>)
8 +2152: i178 = int_sub(i173, 1)
9 +2156: setfield_gc(ConstPtr(ptr179), i178, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
10 +2160: i181 = int_le(i178, 0)
11 +2164: guard_false(i181, descr=<Guard0x166a48b60>)
12 +2170: i183 = int_sub(i178, 1)
13 +2174: setfield_gc(ConstPtr(ptr184), i183, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
14 +2178: i186 = int_le(i183, 0)
15 +2182: guard_false(i186, descr=<Guard0x166a48e00>)
16 +2188: i188 = int_sub(i183, 1)
17 +2192: setfield_gc(ConstPtr(ptr189), i188, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
18 +2196: i191 = int_le(i188, 0)
19 +2200: guard_false(i191, descr=<Guard0x166a48ec0>)
20 +2206: i193 = int_sub(i188, 1)
21 +2210: setfield_gc(ConstPtr(ptr194), i193, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
22 +2214: i196 = int_le(i193, 0)
23 +2218: guard_false(i196, descr=<Guard0x166a48f20>)
24 +2224: i198 = int_sub(i193, 1)
25 +2228: setfield_gc(ConstPtr(ptr199), i198, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
26 +2232: i201 = int_le(i198, 0)
27 +2236: guard_false(i201, descr=<Guard0x166a48f80>)
28 +2242: p202 = getfield_gc_r(p151, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_0_16 pure>)
29 +2246: guard_nonnull_class(p202, 4544843936, descr=<Guard0x166a48fe0>)
30 +2266: p204 = getfield_gc_r(p151, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_1_24 pure>)
31 +2270: p205 = getfield_gc_r(p151, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_2_32 pure>)
32 +2274: p206 = getfield_gc_r(p151, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_3_40 pure>)
33 +2278: p207 = getfield_gc_r(p151, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_4_48 pure>)
34 +2282: p208 = getfield_gc_r(p151, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_5_56 pure>)
35 +2286: p209 = getfield_gc_r(p151, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_6_64 pure>)
36 +2290: p210 = getfield_gc_r(p151, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_7_72 pure>)
37 +2294: p211 = getfield_gc_r(p151, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_8_80 pure>)
38 +2305: guard_class(p204, 4544843936, descr=<Guard0x166a4eec0>)
39 +2317: guard_class(p205, 4544843936, descr=<Guard0x166a4ef08>)
40 +2331: guard_class(p206, 4544843936, descr=<Guard0x166a4ef50>)
41 +2343: guard_class(p207, 4544843936, descr=<Guard0x166a4ef98>)
42 +2356: guard_class(p208, 4544843936, descr=<Guard0x166a4efe0>)
43 +2369: guard_class(p209, 4544843936, descr=<Guard0x166a4f028>)
44 +2382: guard_class(p210, 4544843936, descr=<Guard0x166a4f070>)
45 +2395: guard_class(p211, 4544836568, descr=<Guard0x166a4f0b8>)
46 +2409: p220 = getfield_gc_r(p211, descr=<FieldP rsqueakvm.plugins.value.pointers.W_PointersValue.inst__shape_8 pure>)
47 +2434: guard_value(p220, ConstPtr(ptr221), descr=<Guard0x166a4f100>)
48 +2443: i223 = int_sub(198, 1)
49 +2447: setfield_gc(ConstPtr(ptr224), i223, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
50 +2451: i226 = int_le(i223, 0)
51 +2455: guard_false(i226, descr=<Guard0x166a49040>)
52 +2461: p227 = new_with_ytable(descr=<SizeDescr 88>)
53 +2508: setfield_gc(p227, ConstPtr(ptr228), descr=<FieldP rsqueakvm.plugins.value.pointers.W_PointersValue.inst__shape_8 pure>)
54 +2512: setfield_gc(p227, p93, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_0_16 pure>)
55 +2516: setfield_gc(p227, p92, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_1_24 pure>)
56 +2527: setfield_gc(p227, p91, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_2_32 pure>)
57 +2538: setfield_gc(p227, p90, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_3_40 pure>)
58 +2549: setfield_gc(p227, p89, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_4_48 pure>)
59 +2560: setfield_gc(p227, p88, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_5_56 pure>)
60 +2571: setfield_gc(p227, p55, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_6_64 pure>)
61 +2582: setfield_gc(p227, p82, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_7_72 pure>)
62 +2593: setfield_gc(p227, p83, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize9.inst__raw_storage_8_80 pure>)
63 +2604: jump(p0, p2, p3, p4, i7, p8, p9, p202, p150, p149, p148, p147, p146, p145, p144, p142, p227, p204, p205, p206, p207, p208, p209, p210, p211, p22, p24, p26, p28, p30, p32, p34
   , p36, p38, p40, p42, p69, i223, descr=TargetToken(6017035024))

```

TABLE 5: Operation counts and relation for optimized and unoptimized reverse in Theseus, niladic elements. Reduction factors are for run total.

Loop iterations	Listing 29 199 998 956			Listing 30 22 221 114			Reduction Factor 9.00
	per loop	once	run total	per loop	once	run total	
Allocations	1	0	199 998 956	1	0	22 221 114	9.00
Stores	3	0	599 996 868	2	0	44 442 228	13.50
Loads	4	10	799 995 834	2	12	44 442 240	18.00
Meta	4	11	799 995 835	2	13	44 442 241	18.00
Other	9	8	1 799 990 612	57	56	1 266 603 554	1.42
Operations	21	29	4 199 978 105	64	81	1 422 151 377	2.95
– Other	12	21	2 399 987 493	7	25	155 547 823	15.43

*Value
optimization
quantified*

similar to the optimized traces of Theseus and Pycket. Owing to the architecture, we see multiple interrupt check patterns just like the one found in the unoptimized trace; six at the beginning of the loop and one in between content extraction and object creation. Since this check counter decrement is very regular and it is unlikely that the null-check branch is taken, current processors will very likely coalesce these operations. The overall number of relevant operations (cf. table 4, to the right) is in the same range as for Theseus and Pycket. However, with $(24\,991\,583 + 1) \cdot 8$ elements processed by the loop, the remaining 67 326 elements show a long warm-up. Compared with the non-optimized version, the warm-up increased eightfold, which is again similar to the other implementations.

The shape-based optimization seems to have aligned the execution for all three implementations; when their traces looked different in the unoptimized variant, they were much more similar in the optimized variant.

7.5.4 Theseus and niladic elements list

Turing to the niladic elements variants, we see that for Theseus (not optimized) the trace listing 29 is virtually identical to listing 23, except that the contents of the processed compound value are not checked to be an integer, but rather a compound value with a certain shape (lines 3, 7 and 8). As a result,

there is one more load and one more guard, but the essence is the same, as can also be seen in the left part of [table 5](#). However, the optimized trace in [listing 30](#) is much shorter and completely dissimilar to [listing 23](#). Compared with the unoptimized version, there are quite a few loads and stores less; the allocated object seems to only have one field beyond its shape. As seen in [line 6](#), the allocation `new_with_vtable(descr=<SizeDescr 24>)` requests space for three 64 bit-sized elements, which accounts for the RPython-specific *vtable*, the shape and one field. Nevertheless, the iteration count, as seen in the right part of [table 5](#), is even lower than in the optimized version of the numeric elements variant. Together with the low amount of memory operations, we see an overall reduction of relevant operations by a factor of 15.


Discussion of reverse as representative example

The inlining is not as visible, although present. Since a niladic value is eligible for inlining, the shape recognition algorithm will introduce new shapes when encountering many of those values. However, there is nothing to inline, as their storage is essentially empty. The “splicing” algorithm used will not allocate any memory for empty lists that are to be incorporated into a new storage. That way, an inlined niladic value is only observable in the structure of a shape. In our case, this means that the inlining has happened but ended up with a nine-times inlined shape and no storage for these. The store into the storage we see in [line 8](#) is the reference to the next, equally inlined compound value in the list. Accordingly, the number of iterations is 9 times lower, which nicely corresponds to the (exclusive) maximum shape depth limit of 10 shapes deep.

Considering the iteration counts in [table 5](#), the unoptimized version had comparatively short warm-up, only 1043 elements were not processed by the loop. On the other hand, the optimized version’s loop processed $(22\ 221\ 114 + 1) \cdot 9$ elements, leaving 9965 elements. This amounts to a nearly tenfold increase in warm-up.

7.5.5 Pycket and niladic elements list


Regarding Pycket (original) with niladic elements, there is no difference between [listing 31](#) and the numeric elements variant in [listing 25](#), save for the order of one operation. Likewise, the numbers in the left part of [table 6](#) are

LISTING 29: Trace for reverse in Theseus (not optimized) , niladic elements. Loop without peeled iteration. Debug operations are omitted.

```

1 +526: label(p6, p3, p15, p16, p1, descr=TargetToken(4551549680))
2 +557: p23 = getfield_gc_r(p16, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize2.inst__storage_0 16 pure>)
3 +561: p24 = getfield_gc_r(p16, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize2.inst__storage_1 24 pure>)
4 +565: guard_nonnull_class(p23, 4545613824, descr=<Guard0x10f4ba3e0>)
5 +584: p26 = getfield_gc_r(p23, descr=<FieldP lamb.model.W_Constructor.inst__shape 8 pure>)
6 +588: guard_nonnull_class(p24, 4545607472, descr=<Guard0x10f4ba440>)
7 +607: p28 = getfield_gc_r(p24, descr=<FieldP lamb.model.W_Constructor.inst__shape 8 pure>)
8 +618: guard_value(p26, ConstPtr(ptr29), descr=<Guard0x10f4bc260>)
9 +634: guard_value(p28, ConstPtr(ptr30), descr=<Guard0x10f4bc2a8>)
10 +643: p31 = new_with_vtable(descr=<SizeDescr 32>)
11 +680: setfield_gc(p31, ConstPtr(ptr32), descr=<FieldP lamb.model.W_Constructor.inst__shape 8 pure>)
12 +684: setfield_gc(p31, p6, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize2.inst__storage_0 16 pure>)
13 +688: setfield_gc(p31, p3, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize2.inst__storage_1 24 pure>)
14 +692: jump(p15, p31, p23, p24, p1, descr=TargetToken(4551549680))

```

LISTING 30: Trace for reverse in Theseus , niladic elements. Loop without peeled iteration. Debug operations are omitted.


```

1 +573: label(p15, p23, p1, descr=TargetToken(4372137328))
2 +589: p27 = getfield_gc_r(p23, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize1.inst__storage_0 16 pure>)
3 +593: guard_class(p27, 4362515880, descr=<Guard0x1049ba2c0>)
4 +605: p29 = getfield_gc_r(p27, descr=<FieldP theseus.model.W_Constructor.inst__shape 8 pure>)
5 +616: guard_value(p29, ConstPtr(ptr30), descr=<Guard0x1049b2020>)
6 +625: p31 = new_with_vtable(descr=<SizeDescr 24>)
7 +662: setfield_gc(p31, ConstPtr(ptr32), descr=<FieldP theseus.model.W_Constructor.inst__shape 8 pure>)
8 +666: setfield_gc(p31, p15, descr=<FieldP rpython.tool.pairtype.W_ConstructorSize1.inst__storage_0 16 pure>)
9 +670: jump(p31, p27, p1, descr=TargetToken(4372137328))


```

TABLE 6: Operation counts and relation for optimized and unoptimized reverse in Pycket, niladic elements. Reduction factors are for run total.

Loop iterations	Listing 31			Listing 32			Reduction Factor
	199 998 956			22 222 023			
	per loop	once	run total	per loop	once	run total	
Allocations	1	0	199 999 866	1	0	22 222 023	9.00
Stores	3	0	599 999 598	2	0	44 444 046	13.50
Loads	3	10	599 999 608	2	18	44 444 064	13.50
Meta	3	13	599 999 611	2	21	44 444 067	13.50
Other	11	12	2 199 998 538	67	67	1 488 875 608	1.48
Operations	21	35	4 199 997 221	74	106	1 644 429 808	2.55
– Other	10	23	1 999 998 683	7	39	155 554 200	12.86

LISTING 31: Trace for reverse in Pycket (original) , niladic elements. Loop without peeled iteration. Debug operations are omitted.

```
1 +595: label(p18, p12, p14, p16, p1, p8, p20, descr=TargetToken(4606214256))
2 +637: guard_not_invalidated(descr=<Guard0x11252c140>)
3 +637: p26 = getfield_gc_r(p12, descr=<FieldP pycket.values_struct.W_Struct.inst__type 8 pure>)
4 +648: guard_value(p26, ConstPtr(ptr27), descr=<Guard0x1129256a0>)
5 +657: p28 = getfield_gc_r(p12, descr=<FieldP rpython.tool.pairtype.W_StructSize2.inst__storage_1 24 pure>)
6 +661: guard_class(p28, 4561331656, descr=<Guard0x112925658>)
7 +673: p30 = getfield_gc_r(p12, descr=<FieldP rpython.tool.pairtype.W_StructSize2.inst__storage_0 16 pure>)
8 +677: guard_class(p30, 4561510248, descr=<Guard0x112925610>)
9 +689: p32 = new_with_vtable(descr=<SizeDescr 32>)
10 +726: setfield_gc(p32, ConstPtr(ptr33), descr=<FieldP pycket.values_struct.W_Struct.inst__type 8 pure>)
11 +730: setfield_gc(p32, p16, descr=<FieldP rpython.tool.pairtype.W_StructSize2.inst__storage_1 24 pure>)
12 +734: setfield_gc(p32, p14, descr=<FieldP rpython.tool.pairtype.W_StructSize2.inst__storage_0 16 pure>)
13 +738: jump(p18, p28, p30, p32, p1, p8, p20, descr=TargetToken(4606214256))
```

LISTING 32: Trace for reverse in Pycket (optimized) , niladic elements. Loop without peeled iteration. Debug operations are omitted.

```
1 +824: label(p7, p38, p34, p1, p17, p15, p22, descr=TargetToken(4426204288))
2 +877: guard_not_invalidated(descr=<Guard0x10775a020>)
3 +877: p42 = getfield_gc_r(p38, descr=<FieldP rpython.tool.pairtype.W_StructSize1.inst__storage_0 16 pure>)
4 +881: guard_class(p42, 4380496024, descr=<Guard0x107dc01d0>)
5 +893: p44 = getfield_gc_r(p42, descr=<FieldP pycket.values_struct.W_Struct.inst__shape 8 pure>)
6 +904: guard_value(p44, ConstPtr(ptr45), descr=<Guard0x107dc00f8>)
7 +913: p46 = new_with_vtable(descr=<SizeDescr 24>)
8 +950: setfield_gc(p46, ConstPtr(ptr47), descr=<FieldP pycket.values_struct.W_Struct.inst__shape 8 pure>)
9 +954: setfield_gc(p46, p34, descr=<FieldP rpython.tool.pairtype.W_StructSize1.inst__storage_0 16 pure>)
10 +958: jump(p7, p42, p46, p1, p17, p15, p22, descr=TargetToken(4426204288))
```

virtually identical to those from the numeric elements variant. The similarity to the respective Theseus trace remain. Warm-up is the same.

For Pycket (optimized), the trace [listing 32](#) is, again, very dissimilar to its numeric elements counterpart but matches nicely the Theseus variant in [listing 30](#) with but one more operation, which is also reflected in the very similar numbers in the right part of [table 6](#). Inlining “into” the shape has happened here, too.

The warm-up has slightly increased to about thirteen times. Considering the iteration counts in [table 6](#), only 133 elements were not processed by the unoptimized versions’ loop, whereas, the optimized version’s one left about 1784 elements. This is but slightly more than for the numeric elements variant.

TABLE 7: Operation counts and relation for optimized and unoptimized reverse in RSqueak, niladic elements. Reduction factors are for run total.

Loop iterations	Listing 33 199 991 683			Listing 34 22 213 825			Reduction Factor 9.00
	per loop	once	run total	per loop	once	run total	
Allocations	2	0	399 983 366	1	0	22 213 825	18.01
Stores	6	1	1 199 950 099	11	9	244 352 084	4.91
Loads	7	35	1 399 941 816	2	44	44 427 694	31.51
Meta	11	19	2 199 908 532	11	32	244 352 107	9.00
Other	48	47	9 599 600 831	175	176	3 887 419 551	2.47
Operations	74	102	14 799 384 644	200	261	4 442 765 261	3.33
– Other	26	55	5 199 783 813	25	85	555 345 710	9.36

*Value
optimization
quantified*

7.5.6 RSqueak and niladic elements list

In the trace for RSqueak (original) with niladic elements, a deviation between [listing 33](#) and the numeric elements variant [listing 27](#) can be found, in that the former variant does not have to box and unbox the current element, but can rather reuse the niladic value. This is supported by the numbers to the left in [table 7](#). Interrupt check counter and frame accounting remain the same.

The trace for RSqueak (optimized) in [listing 34](#) shows similarities to the unoptimized counterpart, but with important changes. First, like for Theseus and Pycket, we see the allocation of a one-filed object in [line 43](#). Second, the pattern for the interrupt check counter decrement now happens nine times, and at different points in the trace. However, leaving the interrupt check aside, the trace is very similar to the optimized ones from the other implementations. This again repeats the alignment of how the traces look that can be observed in the numeric elements variant. The nine times smaller iteration count, together with the significantly reduced number of allocations and loads, shows a reduction of almost one order of magnitude in the total number of relevant operations executed. This can be seen to the right in [table 7](#).

The iteration counts in [table 7](#) shows that 8316 elements were not processed by the unoptimized versions' loop. In the optimized variant, 75 566 accounted


TABLE 8: Reduction factors for operations across optimizations and implementations; increase factor for warm-up

	numeric elements			niladic elements		
	Theseus table 2	Pycket table 3	RSqueak table 4	Theseus table 5	Pycket table 6	RSqueak table 7
Iterations	8.00	8.00	8.00	9.00	9.00	9.00
Allocations	8.00	8.00	24.01	9.00	9.00	18.01
Stores	2.40	2.40	9.34	13.50	13.50	4.91
Loads	2.40	2.40	14.40	18.00	13.50	31.51
All relevant	2.11	2.06	14.30	15.43	12.86	9.36
Increase factor for warm-up						
	Theseus	Pycket	RSqueak	Theseus	Pycket	RSqueak
Warm-up	8.23	11.37	8.06	9.55	13.41	9.09

for warm-up, the highest number in this analysis, but compared with the unoptimized version, the warm-up has been reduced to about nine times.

7.5.7 Findings

The detailed look at the twelve traces has revealed that the inlining process works as expected. For numeric elements, we generally see a reduction in loop iterations proportional to the size increase of the objects allocated within the loop. Since overhead for objects and next-pointers is avoided, the overall number of executed, relevant operations decreased in all cases. In table 8, for each category of operations, the overall reduction factors between unoptimized and optimized versions is summarized. That is, for any n in the table, the optimized version executes n times *less* operations compared to the unoptimized version. The factors for the iteration count match nicely the algorithm's parameters. The maximum object size of 8 is reflected directly, as the number of processed elements per iteration rose from 1 to 8. Likewise, the maximum shape depth is also reflected. Since for the niladic elements variant, the element has a shape itself that has to be considered for the overall shape depth during

LISTING 33: Trace for reverse in RSqueak (original) , niladic elements. Loop without peeled iteration. Debug operations are omitted.

```


1 +*994: label(p0, p2, p3, p4, i7, p8, p9, p59, p12, p80, p22, p24, p26, p28, p30, p32, p34, p36, p38, p40, p42, i85, descr=TargetToken(5059610608))
2 +*1021: guard_not_invalidated(descr=<Guard0x12a4e1040>)
3 +*1021: p89 = getfield_gc_r(p80, descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
4 +*1032: guard_value(p89, ConstPtr(ptr90), descr=<Guard0x12a4e10a0>)
5 +*1041: p91 = getfield_gc_r(p80, descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_storage 16>)
6 +*1045: p93 = getarrayitem_gc_r(p91, 0, descr=<ArrayP 8>)
7 +*1049: guard_nonnull_class(p93, ConstClass(W_PointersObject), descr=<Guard0x12a4e1100>)
8 +*1068: p95 = force_token()
9 +*1068: p98 = force_token()
10 +*1075: i102 = instance_ptr_eq(p93, ConstPtr(ptr101))
11 +*1078: guard_false(i102, descr=<Guard0x12a48e578>)
12 +*1084: p104 = force_token()
13 +*1084: p110 = getarrayitem_gc_r(p91, 1, descr=<ArrayP 8>)
14 +*1088: guard_nonnull_class(p110, ConstClass(W_PointersObject), descr=<Guard0x12a4e1160>)
15 +*1107: i113 = int_sub(i85, 1)
16 +*1118: setfield_gc(ConstPtr(ptr114), i113, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
17 +*1122: i116 = int_le(i113, 0)
18 +*1126: guard_false(i116, descr=<Guard0x12a49e080>)
19 +*1132: p117 = new_with_vtable(descr=<SizeDescr 32>)
20 +*1172: p119 = new_array_clear(2, descr=<ArrayP 8>)
21 +*1191: setarrayitem_gc(p119, 0, p59, descr=<ArrayP 8>)
22 +*1195: setarrayitem_gc(p119, 1, p12, descr=<ArrayP 8>)
23 +*1199: setfield_gc(p117, ConstPtr(ptr122), descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
24 +*1211: p123 = getfield_gc_r(p3, descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
25 +*1215: setfield_gc(p117, p119, descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_storage 16>)
26 +*1219: setfield_gc(p117, 0, descr=<FieldS rsqueakvm.model.base.W_AbstractObjectWithIdentityHash.inst_hash 8>)
27 +*1227: guard_value(p123, ConstPtr(ptr56), descr=<Guard0x12a49e0e0>)
28 +*1243: p126 = getfield_gc_r(ConstPtr(ptr53), descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
29 +*1254: guard_value(p126, ConstPtr(ptr55), descr=<Guard0x12a49e140>)
30 +*1270: p127 = getfield_gc_r(ConstPtr(ptr62), descr=<FieldP rsqueakvm.model.pointers.W_PointersObject.inst_strategy 24>)
31 +*1281: guard_value(p127, ConstPtr(ptr64), descr=<Guard0x12a49e1a0>)
32 +*1290: jump(p0, p2, p3, p4, i7, p8, p9, p93, p117, p110, p22, p24, p26, p28, p30, p32, p34, p36, p38, p40, p42, i113, descr=TargetToken(5059610608))

```

inlining, we can inline at most to a depth of 9, which is reflected in all niladic elements variants. The reduction factors range from 2.06 to 24.01.

The warm-up for the optimized version is always higher, as can be seen in the bottom part table 8, ranging from 8.06 to 13.41 time higher. The reduced number of allocations and the omission of object overhead and next-references in the optimized versions can explain the reduction in memory usage. The reduced number of actually executed operations in the optimized versions can explain the reduction in execution time.

Eventually, the traces for all optimized versions seem to align for all implementations where the traces for the unoptimized versions align only between Theseus and Pycket. Save for trace components specific to the language or architecture, all traces showed the inlining predicted. We can conclude that our approach is applicable beyond just our prototype Theseus.

LISTING 34: Trace for reverse in RSqueak (optimized) , niladic elements. Loop without peeled iteration. Debug operations are omitted.

```

1 +1526: label(p0, p2, p3, p4, i7, p8, p9, p81, p106, p22, p24, p26, p28, p30, p32, p34, p36, p38, p40, p42, p68, i136, descr=TargetToken(4899243536))
2 +1565: guard_not_invalidated(descr=<Guard0x11a9075e0>)
3 +1565: i141 = int_sub(i136, 1)
4 +1576: setfield_gc(ConstPtr(ptr142), i141, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
5 +1580: i144 = int_le(i141, 0)
6 +1584: guard_false(i144, descr=<Guard0x11a907520>)
7 +1590: i146 = int_sub(i141, 1)
8 +1594: setfield_gc(ConstPtr(ptr147), i146, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
9 +1598: i149 = int_le(i146, 0)
10 +1602: guard_false(i149, descr=<Guard0x11a9073a0>)
11 +1608: i151 = int_sub(i146, 1)
12 +1612: setfield_gc(ConstPtr(ptr152), i151, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
13 +1616: i154 = int_le(i151, 0)
14 +1620: guard_false(i154, descr=<Guard0x11a907340>)
15 +1626: p155 = getfield_gc_r(p106, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize1.inst___raw_storage_0 16 pure>)
16 +1631: guard_class(p155, 4368275792, descr=<Guard0x11ad233d0>)
17 +1643: p157 = getfield_gc_r(p155, descr=<FieldP rsqueakvm.plugins.value.pointers.W_PointersValue.inst__shape 8 pure>)
18 +1654: guard_value(p157, ConstPtr(ptr158), descr=<Guard0x11ad23388>)
19 +1663: i160 = int_sub(i151, 1)
20 +1667: setfield_gc(ConstPtr(ptr161), i160, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
21 +1671: i163 = int_le(i160, 0)
22 +1675: guard_false(i163, descr=<Guard0x11a907280>)
23 +1681: i165 = int_sub(i160, 1)
24 +1685: setfield_gc(ConstPtr(ptr166), i165, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
25 +1689: i168 = int_le(i165, 0)
26 +1693: guard_false(i168, descr=<Guard0x11a9071c0>)
27 +1699: i170 = int_sub(i165, 1)
28 +1703: setfield_gc(ConstPtr(ptr171), i170, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
29 +1707: i173 = int_le(i170, 0)
30 +1711: guard_false(i173, descr=<Guard0x11a907100>)
31 +1717: i175 = int_sub(i170, 1)
32 +1721: setfield_gc(ConstPtr(ptr176), i175, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
33 +1725: i178 = int_le(i175, 0)
34 +1729: guard_false(i178, descr=<Guard0x11a907040>)
35 +1735: i180 = int_sub(i175, 1)
36 +1739: setfield_gc(ConstPtr(ptr181), i180, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
37 +1743: i183 = int_le(i180, 0)
38 +1747: guard_false(i183, descr=<Guard0x11a906f80>)
39 +1753: i185 = int_sub(i180, 1)
40 +1757: setfield_gc(ConstPtr(ptr186), i185, descr=<FieldS rsqueakvm.interpreter.Interpreter.inst_interrupt_check_counter 24>)
41 +1761: i188 = int_le(i185, 0)
42 +1765: guard_false(i188, descr=<Guard0x11a906ec0>)
43 +1771: p189 = new_with_vtable(descr=<SizeDescr 24>)
44 +1808: setfield_gc(p189, ConstPtr(ptr190), descr=<FieldP rsqueakvm.plugins.value.pointers.W_PointersValue.inst__shape 8 pure>)
45 +1812: setfield_gc(p189, p81, descr=<FieldP rpython.tool.pairtype.W_PointersValueSize1.inst___raw_storage_0 16 pure>)
46 +1816: jump(p0, p2, p3, p4, i7, p8, p9, p189, p155, p22, p24, p26, p28, p30, p32, p34, p36, p38, p40, p42, p68, i185, descr=TargetToken(4899243536))

```

7.6 LIMITATIONS

We are aware that this quantitative evaluation is limited. As pointed out earlier, the benchmark set used is not representative for real-world applications. Nevertheless, assuming basic meaningfulness of the measurements, the desired effect could be shown. Moreover, while the measurements include warm-up

in the sense that we did not do separate executions before starting measurements, we nevertheless separated the preparation of the rather long lists from the operations on them. This might have had an impact on the JIT compiler.

There are deviations in style and architecture for the benchmarks between Squeak and the other two systems. We explained our rationale to that earlier, however, it might still be a source of inaccuracies. For example, the filter benchmark for Theseus and Racket uses a tail-call-modulo-cons architecture to build up the result list. In the loop-style of Squeak, it is not possible to express this efficiently without also introducing means similar to tail-call-elimination. Therefore, the algorithm here builds up the resulting list in reverse and eventually restores order via reverse. A similar approach was taken for append. This might have influenced the results as shown in figures 12 and 13.

While care was taken to shield the benchmark execution from environmental influence as far as possible, it cannot be ruled out that certain influences remained. The hardware-governed speed control of the processor, the granularity of the clock used, and operating system kernel context switches, to name a few, are potential sources of noise that we could not mitigate.

During the statistical processing of the benchmark results — particularly with regards to bootstrapped confidence intervals — we had to make assumptions about the underlying distribution of the time and memory measurements. Since our sample per measurement was too small to verify a certain distribution, we used *Student's t-distribution*. A larger sample size certainly can help to identify the true distribution; however, the confidence intervals for all measurements were rather narrow. Therefore, a higher confidence in the kind of distribution would not necessarily improve our results with respect to how close they reflect reality.

SUMMARY

In five micro-benchmarks across two variants, there is always reduction in memory consumption, often significantly, and almost always reduction in execution time, sometimes very significantly.

8 Related work

Data structure optimization is well documented in literature and industry. We want to put our approach to compound value optimization into this context.

8.1 RELATED CONCEPTS

The concept of compound values is related to other notions of data structures. In particular, *persistent data structures* and *algebraic data structures* have key elements in common with compound values. Moreover, our inlining approach is related to *object inlining* and *cdr-coding*.

8.1.1 Persistent data structures

When a data structure cannot be modified *in-place*, but rather new “versions” of a data structure must be created to result in modification, these data structures can be called *persistent* [35]. This is to be contrasted with *ephemeral* data structures, that can change in-place. Several sub-types of persistent data structures exist, relating to the capability of making new versions of any previous version or not. Literature typically calls for “support[] [to] access [...] multiple versions” [35]. Some variants, such as the “fat node” [35] approach, explicitly record all modification locally and hence provide a way to recover previous version; others employ “path copying” to clone and modify all affected structures— this entails updating all incoming references. Most research in that area is however concerned with turning ephemeral data structures into persistent ones, preferably automatically.

Looking from the perspective of compound values, we essentially provide persistent data structures, as long as object identity is not concerned. That is,

updating data structures always results in new versions. Previous versions of data structures are implicitly retained, but not necessarily in the same place in memory or under the same object id, should the respective programming system have this concept.

Related work

Clojure Clojure [55] is a Lisp-family programming language that strongly advocates the use of persistent data structures, mainly for reasons of functional purity and simplicity of sharing in multithreaded settings. The reference implementation runs on top of regular JVMs. Clojure supports lists, vectors, maps, sets, and other collections in a persistent fashion. Ephemeral data structures are called *transient* and are available as counterpart to most persistent data structures, but are discouraged to use except for very few use cases.

The creators of Clojure regard the value-nature of their persistent data structures as one of their important features [56]. That is to say, a sharp distinction between persistent data structures and compound values is not always drawn. In that regard, compound values as presented and optimized herein can be used quite similarly to Clojure’s persistent data structures.

As of writing, the reference implementation of Clojure’s persistent data structures do not optimize layout, storage, or access in any special way, leaving potential optimizations to the underlying JVM (see also section 8.7 below).

8.1.2 Algebraic data types

From a data structure optimization point of view, compound values are similar to *algebraic data types* as found in languages in the ML family [31, 73]. Hence, optimizations done to this category of data structures are relevant to compound values, too [68, 69].

8.1.3 Object inlining

Wimmer has proposed object inlining [114] as a general data structure optimization for structured objects in Java. This approach shares many similarities with ours: it also inlines objects into their referring objects, saving space and pointer indirections. It has a number of advantages over our approach: the

approach guarantees to never need more memory than without the optimization. Also, it does not need any complex run-time support, since it relies on a static, global analysis to identify classes for which the inlining is possible. However, for that reason, it cannot be applied so easily to dynamic languages and in situations where reflection or class loading is used. Additionally, the inlining decision is done per class, while in our approach several different shapes and thus inlining patterns can be created for a value class.

*Related
concepts*

8.1.4 Cdr coding

Cdr coding [113, § 5.4] is an optimization for *cons* lists provided in *hardware*. It can be applied to both mutable and immutable structures. *Cons* lists that are created with a certain interface are not completely stored as linked lists, but compressed like arrays. The intermediate references are omitted and all relevant list operations, in particular the ones that mutate a list, take special care to utilize this optimized storage. Specifically, the `rplacd` function, which can be used to alter the *next* pointer in a linked list, checks whether the optimized storage can be retained or whether actual *cons* cells have to be allocated and connected to the list.

When our approach is applied to *cons* cells, the effective memory layout will match that of cdr coding. Nevertheless, cdr coding is limited to *cons* lists and also does not work on arbitrary *cons* cells, such as improper lists. It is, however, the only hardware-supported optimization of this kind known to us.

*

**

One language-level implementation of compound values (named complex values) [103] in Squeak/Smalltalk is similar to our image-side Smalltalk code for RSqueak. However, the complex values project cannot rely on support by the `vm` and hence provides all means necessary by code generation and tooling. Due to the way the `vm` works, immutability cannot be guaranteed and identity-based comparison cannot be completely avoided.

8.2 LANGUAGE-LEVEL OPTIMIZATION

Improving data structures to gain execution speed has been proposed for operations on linked lists in functional languages, for example by unrolling [104]. Typically, those optimizations are restricted to linked lists of *cons* cells.

Related work

One of the key effects in our optimization is avoiding to allocate intermediate data structures. In that respect, *hash consing* [41, 44, 52], as used in functional languages for a long time, is related to this work. However, hash consing typically works at the language level using libraries, coding conventions, or source-to-source transformations. It is not adaptable at run-time.

Deforestation [50, 108, 112] has the aim to eliminate intermediate data structures and is in this respect related to our approach. However, deforestation deliberately works through program transformation and does not incorporate dynamic usage information. It is typically only available to statically typed functional languages, such as ML.

8.3 JUST-IN-TIME COMPILERS

Compiling to native code at run-time, that is **JIT** compilation, is a prevalent and extensively studied technique, found in several different, but chiefly object-oriented, dynamically-typed languages [4]. Prominent examples include the Smalltalk-80 bytecode to native code compiler by Deutsch and Schiffman [34], and the optimizing **JIT** compiler of Self, with type specialization and speculative inlining [24]. These concepts were later used in the HotSpot **JIT** compiler [85] for Java.

The prevalence of web browsers has made **JIT** compilation an important topic for JavaScript implementations, for example the int V8 JavaScript implementation [59]. The map transitions for hidden classes used in V8 [3] and inspired by Self [24], are in principle similar to our notion of transformation rules. As well as objects in V8 start with a default hidden class and follow map transitions to their most optimal hidden class, the transformation rules in our approach change the shape of a compound value from its default shape to its most optimized one during the compound value's creation.

An important difference between the hidden classes of V8 to the shapes of our approach is that V8 needs to deal with the objects being mutated after their construction. Indeed, while the hidden classes of V8 (and similarly of Higgs [25]) can encode the type of the fields of the objects, they do that only for primitive values like int, float etc. They cannot recursively express that a field is itself an object with a specific hidden class, which is what we do with shapes in the current paper. The reason this is impossible (or at least significantly harder) in the JavaScript setting is the fact that the inner object can be mutated later, which might cause its hidden class to change.

Tracing JIT compilers as introduced by Mitchell [76] have seen implementations for Java [49], JavaScript [48], or Lua⁵ to name a few. In the context of a JavaScript implementation, the SPUR project [11] provided a tracing JIT compiler for Microsoft's Common Intermediate Language (CIL). Tracing an *interpreter* that runs a program instead of tracing the program itself is the core idea of meta-tracing JIT compilers, pioneered in the DynamoRIO project [106]. PyPy [15, 99] is a meta-circular Python implementation that uses a meta-tracing JIT compiler. Provided through the RPython tool chain, other language implementations can benefit from a meta-tracing, Haskell [109], PHP⁶, or R⁷. The meta-tracing JIT used in this work is provided by RPython, as well.

8.4 DATA STRUCTURE IMPLEMENTATION

Many systems, particularly dynamically typed systems, have a need to distinguish language-level structured data, such as objects, from primitive data, such as integers. A common approach to represent such a distinction is *tagging*, that is, using a bit in a machine word to distinguish numbers from objects/handles/pointers/etc., which would otherwise be potentially identical. This technique can also be applied to structured data, for example *cons* cells and

⁵<http://luajit.org> (last accessed March 17, 2021).

⁶<http://hippyvm.baroquesoftware.com/> (last accessed March 17, 2021).

⁷https://bitbucket.org/roy_andrew/rapydo (last accessed March 17, 2021).

Related work

has often been applied for Lisp, both in interpreters and Lisp machines [46]. Since then, tagging has been applied in numerous environments, including but not limited to Smalltalk [51], or even OCaml [69]. Gudeman gives a comprehensive overview of the common alternatives [53]. It is still applicable today, despite more recent variants for newer processor architectures. Some processor architectures besides the Lisp machines, such as SPARC, support tagged representations directly. That is, dedicated instructions exist for arithmetics with integers that are represented with a tag in the least significant bit. Yet, this is typically limited to arithmetics of integers alone.

Late Data Layout is a lightweight annotations mechanism [111] to eliminate limitations of coercions between internal data representations. Boxing and unboxing operations are not inserted eagerly by a compiler but only at execution time, with checks that ensure the consistency of the data representation. The checks are based on multi-phase *type-driven* data representation transformations and local type inference. Hence, unnecessary transformation operations can be omitted and data-type representations are added optimally.

The Object Storage Model [115] of Truffle [116] creates every object as an instance of a *storage class*, which works as a container for the instance data. This class references a *shape* that describes the object's data format and behavior. Shapes and all their accessible data are immutable, but the reference to a shape from the storage class themselves can vary over time. Thus, any change of the object's shape results in a new shape. The proposed approach is suitable for sufficiently efficient compilation with further optimizations, such as *polymorphic inline caches (PICs)* for efficient object's property lookup.

A more specialized approach to increase performance of data structures in *vms* is *storage strategies* [16, 91] for collections of homogeneously typed elements. If possible, they are stored unboxed and their type is stored separately and only once with a special object called *strategy*. For example, appending an integer to an empty collection enables the *integer* strategy for this collection and this integer and all subsequent integers will be saved unboxed. However, appending anything other than an integer, such as a string, causes a transition to a generic strategy, because the collection is now heterogeneous. It is assumed that such transformations are unlikely, which is shown by the authors. A similar approach is used for structures with mutable cells in this work. Every

cell has its strategy and its values are saved unboxed when not using a generic strategy.

While pointer tagging and strategies reduce memory consumption by unboxing values, it is also possible to reduce the size of the structure itself, when a substantial amount of structures is allocated. *Structure vectors* group structures of the same type, allowing to store the header and the type descriptor only once [23]. This optimization is most beneficial when large amounts of structures are used, achieving a speed-up of up to 15%. Yet, while allocation becomes faster, field access and especially type descriptor access become up to three to four times slower [23]. However, the allocation of a large number of structures is not very common in Racket. [94]

An effective run-time representation exists for R⁶RS Scheme records [65] where each record has an associated run-time representation, **record-type descriptor (RTD)**, determining its memory layout. When an RTD is created, the compiler calculates record sizes and field offsets for this record type similar to the way presented here. They have flat representation with inlined fields, quite similar to structures in Pycket. A special interface allows to store raw integers, untagged floating point numbers, and raw machine pointers, in addition to ordinary Scheme data types.

The representation of structures in Racket's implementation is related to our work, too. However, we deliberately chose to not investigate the implementation but rather base our approach solely on the extensive documentation and the static and dynamic analyses. A comparison of our implementation to Racket's is part of future work.

8.5 VALUES IN SYSTEMS AND LANGUAGES

There have been several attempts at providing compound values in programming systems, languages, and architectures. We present a few of them and how they relate to our approach.

*Values in
systems and
languages*

8.5.1 CORBA/OMG Object-by-Value

The CORBA extension “Object by Value” [82] provides a notion of compound values for the CORBA architecture. The main benefit for the CORBA system is that with this idea of values, there is no need to track references, which can be a major concern in CORBA implementations, especially with respect to deployment boundaries. CORBA values are in fact always “local”, and, on the occasion of crossing system boundaries, always copied in its entirety; the value exists as independent entity on either side of the system boundary. In fact, this copy semantics (“copy-on-pass”) is a main driving force for this extension and “provide semantics that bridge between CORBA structs and CORBA interfaces” [82, § 5.1]. In that regard, CORBA values are different from compound values as used here, as they allow cyclic relationships. Therefore, the optimizations presented here are not applicable to CORBA values.

Related work

8.5.2 Newspeak Values

Newspeak [21] specifies values as “deeply immutable”. Also, “[a compound value is] globally unique, in the sense that no other object is equal to it.” [20]. Newspeak’s requirements for values essentially include flat values, value-based equality, and recursive immutability, hence specifying compound values as used in this work. However, the specification makes concessions for *mirror* access to compound values. That is, reflective capabilities of the language and execution environment might be able to alter the contents of compound values, given they are located within the same address space (and not, for example, copied across system boundaries). In that regard, immutability of compound values in Newspeak is limited.

At the time of writing, the specification also acknowledges that the reference implementation of Newspeak does not support values *per se*: “At the moment, Newspeak still relies on Squeak Smalltalk for some of its libraries. There is no class value yet.” [20, § 3.1.1] That being said, the SOMns [72] implementation of the Newspeak specification explicitly includes values, but makes

no attempt to optimize values beyond communicating to the underlying JVM that a compound value's constituents are constant.

8.5.3 BETA value types

The BETA system [67] provided a rich set of abstractions for different use cases. With direct influences from SIMULA and DELTA, value types can be explicitly denoted alongside classes and record types, to name a few. In that, these abstraction mechanisms are distinct and the resulting “realizations” (that is, values, objects, record, and so on) cannot cross-reference freely: “A variable in BETA either holds a value or a reference to an object.” [67] That being said, values are characterized as *subpatterns* and not instances, arguing that “a value, like four, is an abstraction over all collections of *four* objects” [67]. This differs from the way even flat values are conceptualized in most other languages, where, for example in Smalltalk, the number four would be a mere instance of a class, value or not. It is not obvious from literature whether BETA's values were specially represented or optimized.

*Values in
systems and
languages*

8.5.4 Fortress compound values and traits

The Fortress language [105] distinguishes between referenced objects and values, calling the latter similar to primitives in other languages. In fact, data types such as Float are expressed in terms of compound values. The penultimate specification [1] gives detailed information on how values in Fortress are to work. Values in Fortress can be compound and are immutable. Although a settable annotation for a value's constituents can be given, it is made explicit that such a “setter” is merely convenience and “an abbreviation for constructing a new compound value with a different value for one field” [1, § 10.3]. The rules for object equivalence [1, § 10.4] make it clear that comparisons for values are based on their respective constituents. Due to the nature of how abstractions are provided in Fortress, the modifier “value” can also apply to traits [36], which comprise behavior for objects to extend on. A respective object will automatically be a value if it extends upon a value trait.

The last available version of Fortress relies on the `JVM` for execution and hence cannot rely on value support from the underlying `VM`. It is not known whether the language-level compound values were represented in a specially optimized fashion.

Related work

8.6 NON-VALUES IN SYSTEMS AND LANGUAGES

Some systems and languages use the term “value” to refer to data structures in a way that is influenced by the *call-by-value/call-by-reference* evaluation strategies. In these environments, “value” typically means that memory contents are copied when used as parameters or in assignments. Immutability, whether transitive or not, and identity in these systems are typically not required to be compatible with our notion of compound values.

8.6.1 C# and CIL value types

The C# language inherits its notion of values from a C/C++ point of view. There, values are merely a denotation of copy-on-assignment-*semantics*: “Assignment to a variable of a value type creates a *copy* of the value being assigned.” [37, emphasis in original] Since this includes C# structs, composites that are called “value” exists, but there is no statement regarding immutability, atemporality, or similar indications that would make it possible to say values of C# are compound values in the sense used here. This matches the notion of value types as found in the CIL [38, § I.8.2.1].

8.6.2 Scala value classes

Scala [83] provides a notion of value classes, that is said to be “similar to value types in .NET” [84]. However, these value classes are explicitly only unary and primarily intended as a kind of object-oriented shell around primitive types better fitting Scala’s type system. They are thus more wrappers to facilitate inlining than values in the sense used here. Further, there is no communication to the underlying `VM` — typically the `JVM` — that there are values present.

8.7 JAVA AND THE JVM

Neither the language Java nor its most predominantly used `VM` support compound values at the time of writing. That being said, support for compound values has been a long-standing research topic for Java and derivative languages. For the precise reason of more fine-grained performance control, Kava [6] and project Valhalla’s JEP 169 [100] propose variants for compound value support for both the language and the `VM`. The latter of which would also benefit non-Java languages on the `JVM`, for example, Scala, Clojure, or SOMns.

*Java and the
JVM*

Kava Building on the Jikes [5] and Jalapeño [22] `VMs`, Kava went very far in its application of compound values. All primitive types available through the `VM` in Kava are expressed in terms of compound values, deferring the actual representation of these to the `VM`. This work is in line with this research and—to a certain degree—are an instance of the optimizations on the `VM`-level that are necessary to run Kava-esque programs with good performance.

Kava makes a compelling case for reifying “primitive” data types of the underlying `VM` as compound values:

In Kava, `int` is part of the package `kava.lang.primitive`, and is defined as a value object containing an array of 32 enumeration objects of type `bit`, which can have the value zero or one. There is nothing the programmer can do to observe that `int` is not a fully general object. Yet it can be implemented as a 32-bit register value.

[6, §1]

Using shapes to represent ints, however, would not yield the same, optimized representation as in Kava. Rather, our optimization would elide zero and one values into the shape structure, as they are niladic compound values. This would eventually result in one unique shape for every actually used int. Arithmetic on numbers represented in this way is expensive. Thus we do not support Kava’s notion of primitive values.

Project Valhalla Value objects in Valhalla [100] explore the benefits of compound values in the context of both Java the language and the `VM`. One core

Related work

focus is to provide a notion of value objects aimed for performance and integrated with the Java compiler. Value classes (or *inline classes*, for that matter) exist primarily to be inlined into their surrounding data structures at compile time. This may imply compile-time program transformation to ensure effective data access works, analogous to object inlining (cf. [section 8.1.3](#)), but in contrast to its automatic nature, value classes require developers to deliberately chose which classes to inline. Value classes of this kind are required to be “final”, in Java terminology, that is they must not be subclassed; a restriction that is not found in most other compound value concepts. Although intent and effect are different, the general technique is related to hash consing and deforestation (cf. [section 8.2](#)).

SUMMARY

The optimization approach builds on well-known related ideas in data structure representation, hardware optimization, and [JIT](#) compiler compilers. There are diverse notions of what compound values are amongst different systems.

9 Conclusion

This chapter gives future directions to built upon and summarizes our approach and our findings in applying it to three **VM** implementations.

9.1 FUTURE WORK

While our prototypes give promising results on micro-benchmarks, they allow only limited reasoning about more general programs. The applicability of our approach to more general, particularly **OOB** languages and programs for real-world problems remains to be assessed. This includes broadening the benchmarks as well as applying the presented approach to other languages and their notions of data structures and compound values. For example, Racket supports more data types that may be subject to our approach, such as (immutable) vectors and *cons* cells themselves. Likewise, there are more data structures in Squeak that can benefit from our approach; in that case, the effects that potential interactions between objects and compound values have for developers warrant further investigation.

9.2 SUMMARY

While compound values are not wide-spread in **VM**-based programming systems and languages, existing systems that provide compound values show that their adoption can be desirable for several reasons. The efficient representation of compound values in **VMs** presented here adds to these reasons.

The introduction of shared shapes to describe the contents of compound values recursively and the representation of frequent compound values using

a compressed storage layout are fundamental to our approach. Both concepts contribute to a reduction in management overhead for compound values in `VMs`.

Conclusion

The approach is capable of being extended beyond the initial optimization of compound values. In that regard, using the cell indirection in the Pycket implementation has proven worthwhile to support mutability. Further, a more general look at the optimization approach allows expressing other existing data structure optimizations in terms of a shape-guided optimization. Shape-guided unboxing provides a more general way of incorporating flat values into the whole optimization process. The introduced extension can facilitate the integration of our approach with existing `VMs` and provide adjustments for specific situations.

Our approach to optimizing compound values in `VMs` provides very good results both for execution time and memory consumption for a small prototype implementation on selected micro-benchmarks. Applied to more general systems, it still shows significant performance improvements when compound values are used. There is no virtual performance degradation when compound values are not used. With only two instances with no improvements in execution time, execution was in many cases more than twice as fast as without our optimization. Our approach saved memory in all cases and, on average, we see less than half the memory consumption of the original implementations.

**

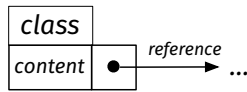
The shape-based optimization of compound values in `VMs` shows good performance characteristics and can provide a useful tool to support efficient compound values in a broader range of programming systems.

Appendix

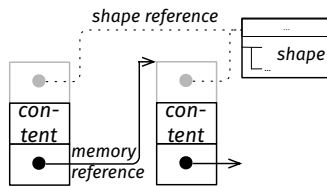
Appendix A

Key to visual language

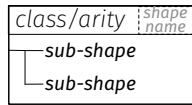
data structures



memory cells



shapes



▼ *direct access shape*

Appendix B

Comprehensive benchmark results

For the sake of completeness, this chapter provides a detailed description of the benchmarking environment, the process of determining the algorithm’s default parameters, and the comprehensive measurement results.

B.1 ENVIRONMENT

We provide information on the hardware of the host computer used for benchmarking, as well as the software used. This include system software, benchmarking software and the software that was subject to benchmarking. Note that the benchmarks themselves can be found in [appendix C.3](#) in brief and online [86] in detail.

B.1.1 Host

All benchmarks were run on a HP DL560 Gen 10 machine. It is equipped with $4\times$ Intel Xeon Gold 6148 (Skylake) CPUs, with 20 cores each, totalling at 80 cores. Hyperthreading was active, resulting in a maximum of 160 native threads. The default core frequency is 2.40 GHz with a turbo boost frequency of 3.70 GHz, governed in hardware by Intel PStates. The cache organization is presented in [table 9](#). The machine has 24×64 GB of DDR4 load-reduced multi-bit-ECC RAM bars, totaling in 1.48 TB of usable memory.

Due to local circumstances it was not possible to completely deactivate hyperthreading. We tried to manually mitigate the Intel PState governor by instructing it to use the baseline frequency as turbo frequency.

TABLE 9: Cache organization of the host machine

Level	Amount	
L1	1.25 MiB	
	20 × 32 KiB	8-way set associative
	20 × 32 KiB	8-way set associative; write-back
L2	20 × 1 MiB	16-way set associative; write-back
L3	20 × 1.375 MiB	11-way set associative; write-back

Comprehensive
benchmark
results

B.1.2 System software

We used Ubuntu 16.04 LTS (Xenial Xerus) as operating system. The Linux kernel used identifies itself as “x86_64 Linux 4.4.0 (unmodified 4.4.0-148-generic)”.

To facilitate reliable measurements, we isolated the actual benchmarks to exactly one core by means of Linux *cgroups*. Note that this was not done for the parameter determination, to the contrary, as many as 80 to 100 exploration benchmarks were executed in parallel.

All benchmarks were run from a RAM-disk so as to minimize input/output-influence. Swap-space was completely disabled, given the amount of usable memory.

We used the *ReBench* benchmarking framework as of version 1.0RC2 [71] to carry out the benchmarks.

B.1.3 Implementations

For the comprehensive results, we include several other implementations beyond those in section 7.4.1 for illustration. These additional implementations are related by the used JIT compiler (PyPy), the availability of algebraic data structures (ML family), or language equivalence (CPython, Racket, Squeak).

Most VMs can be tuned using command line flags or environment variables. For reference, we provide the tunables that were used during benchmarking.

PyPy/RPython-based Several implementations make use of the RPython toolchain, including PyPy itself. We used version 7.1.1 of PyPy and the RPython toolchain. We tuned the garbage collector—for all RPython-based vms—to benefit from the available memory as follows:

```
1 export PYPY_GC_MIN=1GB
2 export PYPY_GC_GROWTH=2.5
```

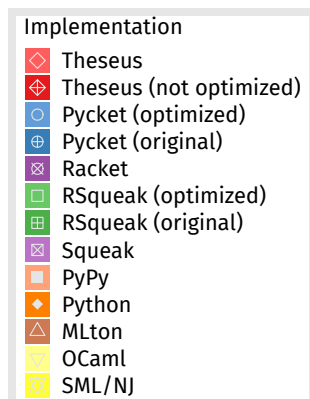
Environment

Theseus The version uses for benchmarking is `c07e4830b05d6fd261384303d2b9dfe8b763c6bb` and tagged as 2019 [90]. The unoptimized variant was obtained by setting the algorithm parameters maximum object size to 0 and substitution threshold to 99 999 999. Theseus is RPython-based, the tunables above apply.

Pycket The optimized version used for benchmarking is `c44ae8a26e9c5e4f6ec2272a6b805a0b5673f00b` and tagged as 2019 [87]. The baseline version is `981fae4a828f42d641c8c0484ed4bdd7d7348660` of the master branch.⁸ Pycket is RPython-based, the tunables above apply.

RSqueak The optimized version used for benchmarking is `b564ba8d2f9a743c750ffb63923aacb10c1ae472` and tagged as 2019 [93]. The baseline version is `7d9b6649efeb1d1fff9f2ecf4084d3dd18082022` of the master branch.⁹ RSqueak is RPython-based, the tunables above apply.

Racket For benchmarking we used version 7.3 of Racket [45].¹⁰ We did not change any tunables.



⁸<https://github.com/pycket/pycket/tree/master> (last accessed March 17, 2021).

⁹<https://github.com/hpi-swa/RSqueak/tree/master> (last accessed March 17, 2021).

¹⁰<https://racket-lang.org/> (last accessed March 17, 2021).

Squeak We used the OpenSmalltalk VM as of version *201810190412* and the default Squeak 5.2 image, obtained as a bundle.¹¹ Note that we slightly adapted the benchmarking code to completely circumvent the primitives that create compound values. While not strictly necessary, this gives a slight advantage to Squeak over our implementation.

We used the following tunables (defaults in braces, our values after the name):

*Comprehensive
benchmark
results*

- `minBackwardJumpCountForCompile (40) → 8`
- `desiredCogCodeSize (1.4 MB) → 5.6 MB` *Note: chosen to be $\leq L2$ cache*
- `desiredNumStackPages (50) → 200`
- `desiredEdenBytes (4/7 of heap) → 32 MB`
- `useMmap (1 GB) → 30 GB`

We also disabled all unnecessary input/output mechanisms, resulting in the following command line flags:

```
1 -noevents -nohandlers -nodisplay -nosound -mmap 30720m -stackpages 200
2 -cogminjumps 8 -codesize 5600k -eden 32m
```

Python We used the stock CPython *2.7.15+* of Ubuntu *16.04.6*.

MLton We used the stock MLton *20100608* of Ubuntu *16.04.6*.

OCaml We used the stock OCaml *4.02.3* of Ubuntu *16.04.6*. We used the following command line flags for compilation:

```
1 -unsafe -noassert -nodynlink
```

SML/NJ We used the stock Standard ML of New Jersey *v110.78* of Ubuntu *16.04.6*.

¹¹<https://files.squeak.org/5.2/Squeak5.2-18231-64bit/Squeak5.2-18231-64bit-201810190412-Linux.zip> (last accessed March 17, 2021).

B.2 PARAMETER DETERMINATION

We give all results for our parameter determination. In the following, we show execution time and memory usage for running all benchmarks on all three optimized implementations. We manually aborted runs that took over 3 to 10 min.

The accumulation results (appendices B.2 to B.2) add up normalized (range 0 to 1) results of the respective benchmark and measurement criterion (execution time or memory consumption), for each variant (niladic elements, numeric elements), and give the respective accumulative range. These four are again accumulated, for niladic elements and numeric elements (appendix B.2). Finally, these two results are accumulated into the final result (figure 49), however, the results of the numeric-elements–based benchmarks are favored 3 : 1 over niladic-elements–based benchmarks, because we think that niladic-element–based programs should be rarer in actual applications.

*Parameter
determina-
tion*


**

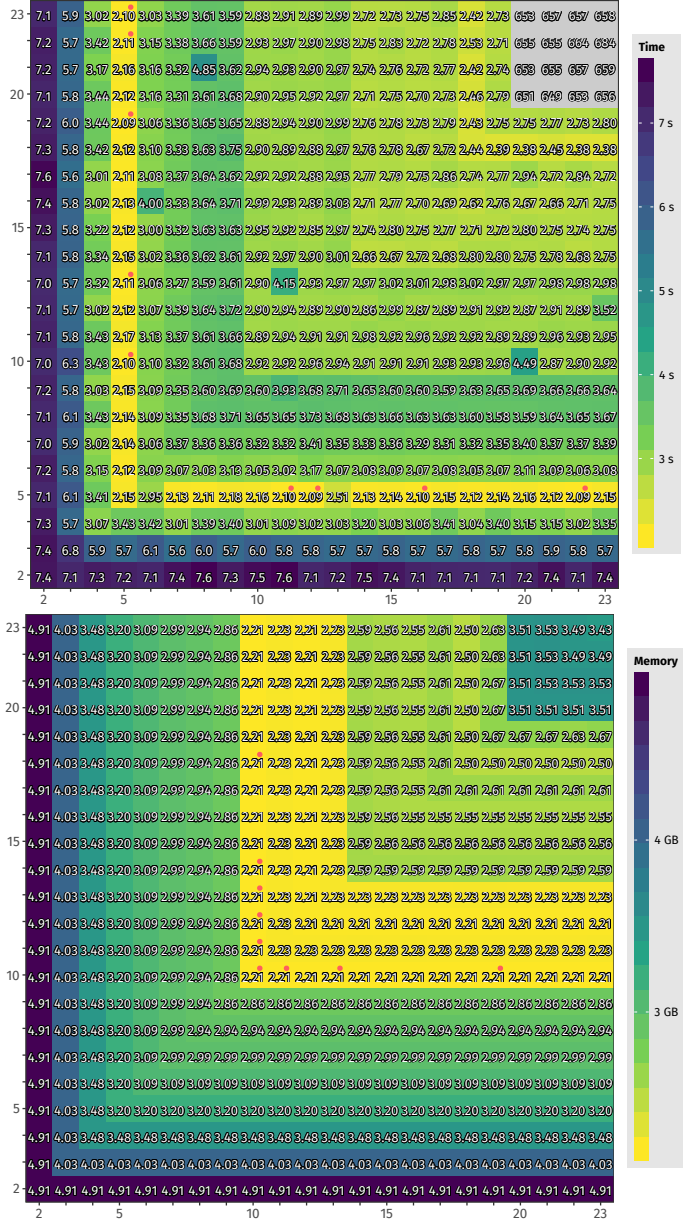
Notation For the following figures, the parameters are presented as follows:

- maximum shape depths on x-axis,
- maximum storage width on y-axis.

Red dots indicate the best 2 % within the measurement table.

Grey areas indicate runs that were aborted or outside the desired range (five times above the minimum).

FIGURE 16: Parameter exploration for reverse on Theseus , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



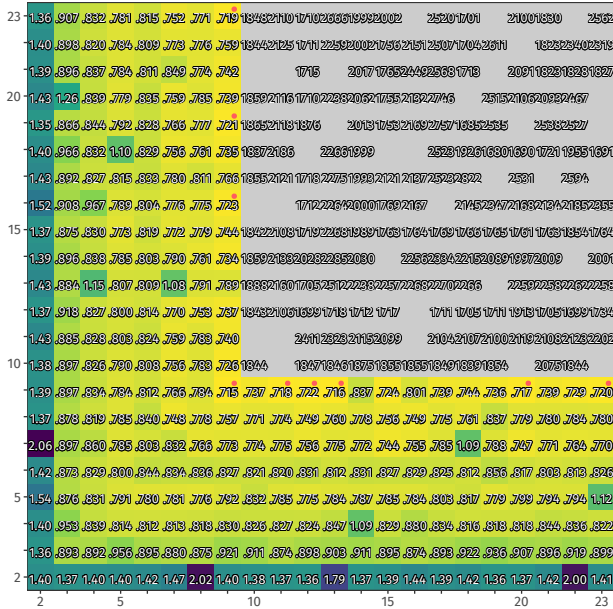


FIGURE 17: Parameter exploration for reverse on Pycket (optimized) \square , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.

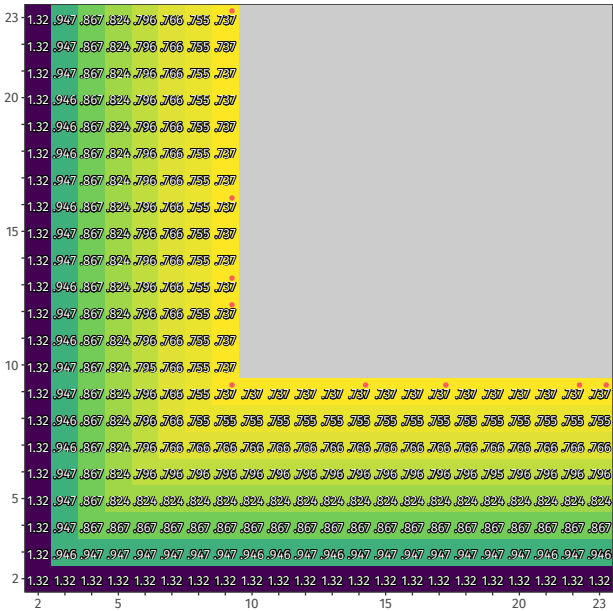
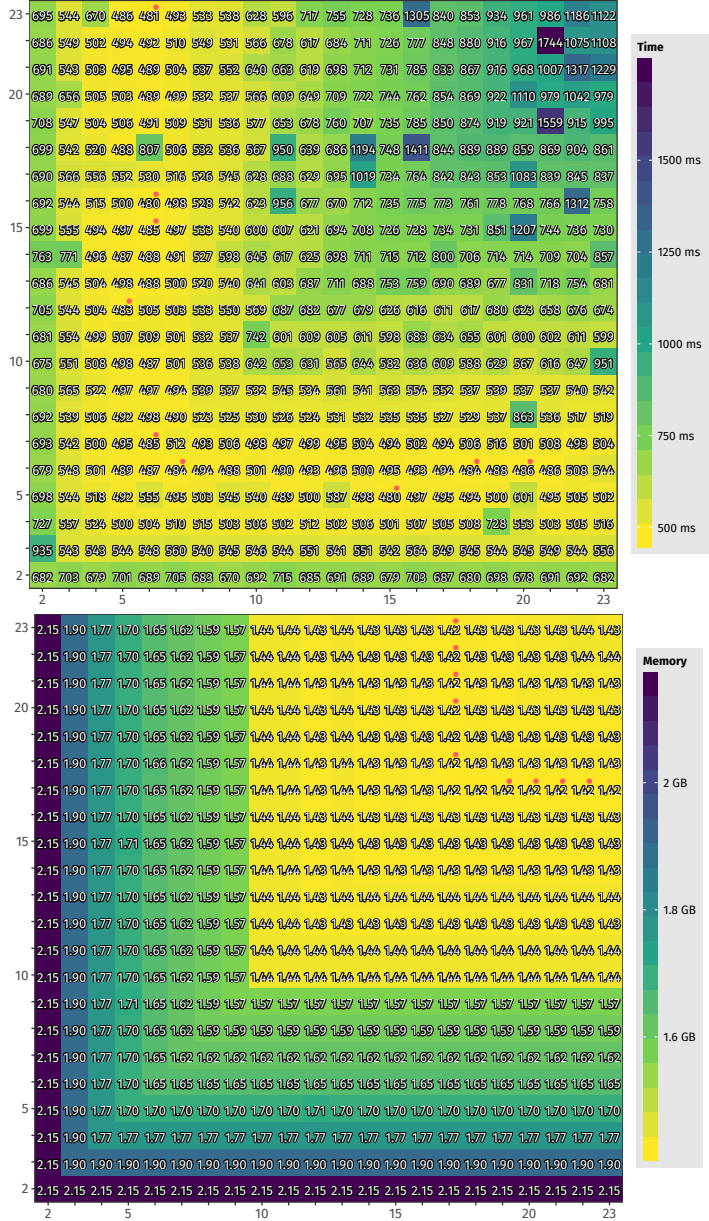


FIGURE 18: Parameter exploration for reverse on RSqueak (optimized) \square , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



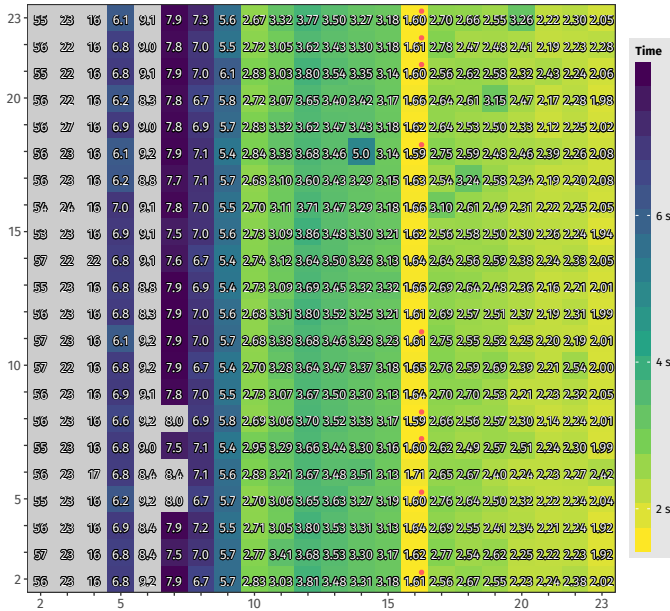


FIGURE 19: Parameter exploration for reverse on Theseus, niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.

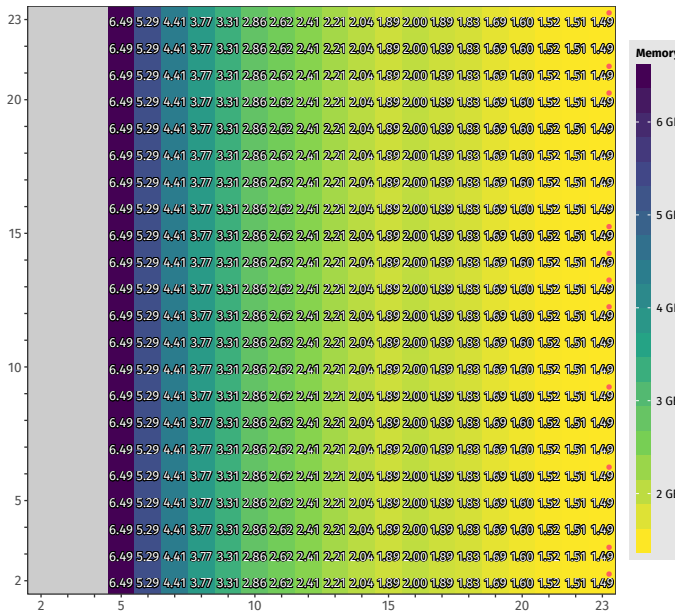

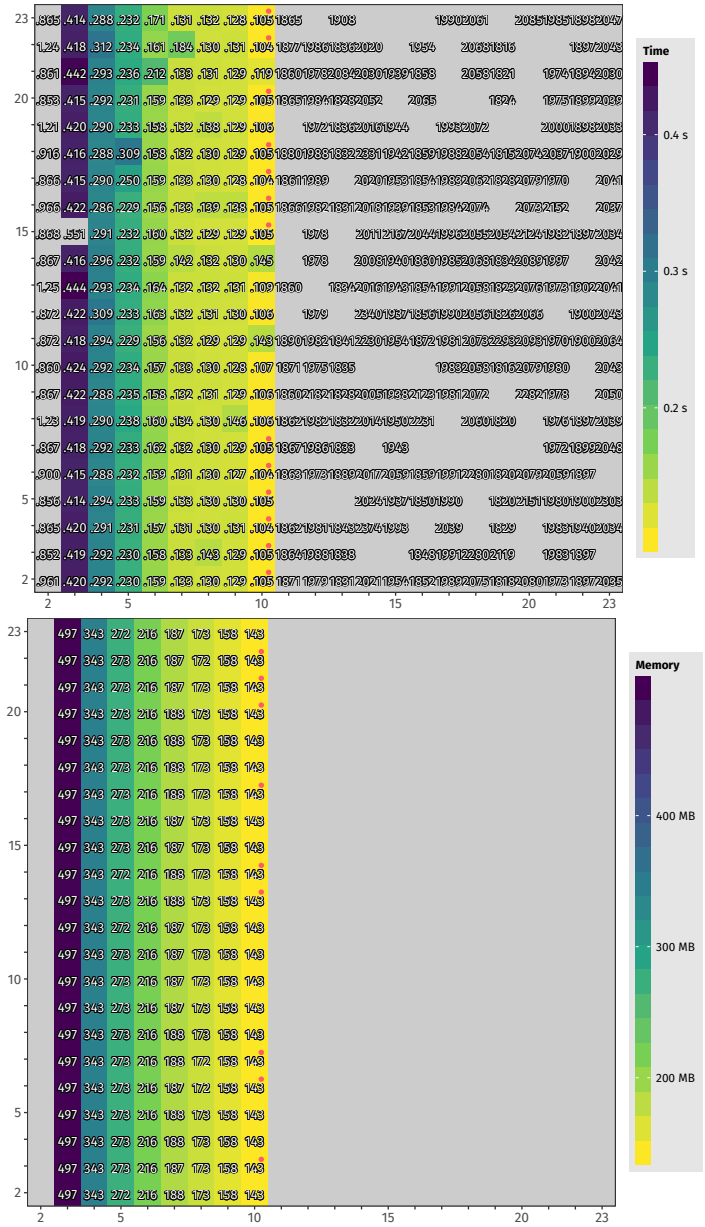


FIGURE 20: Parameter exploration for reverse on Pycket (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.



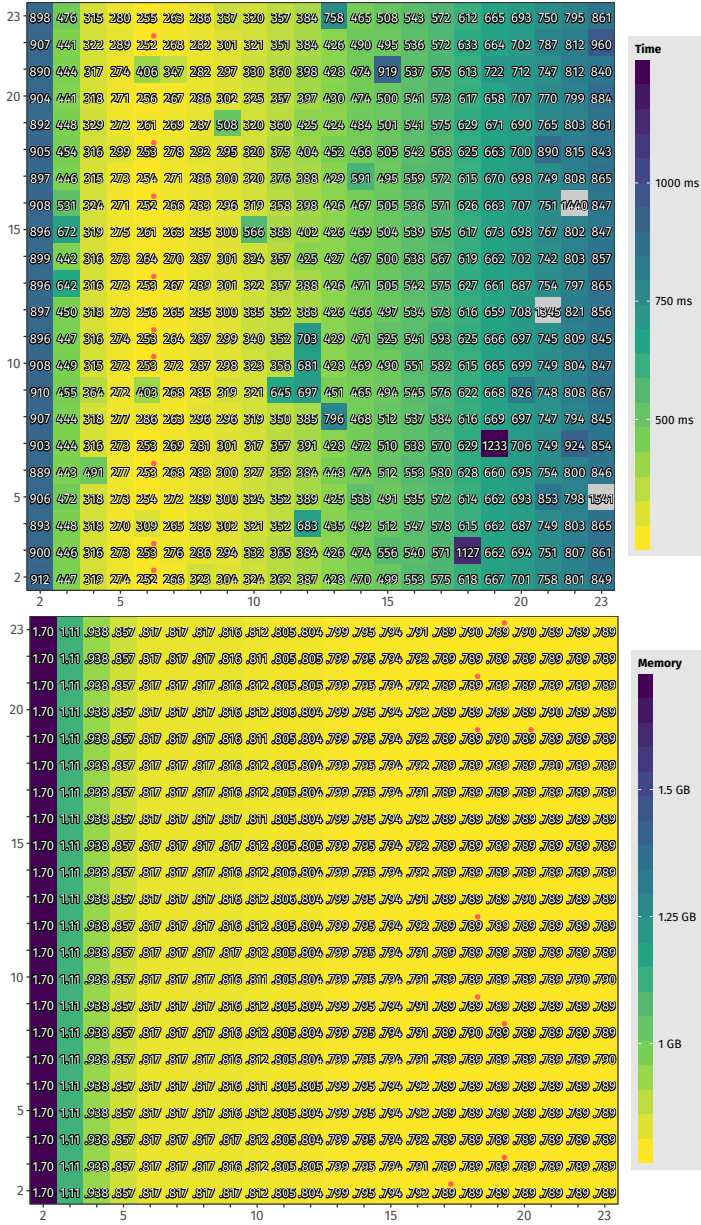




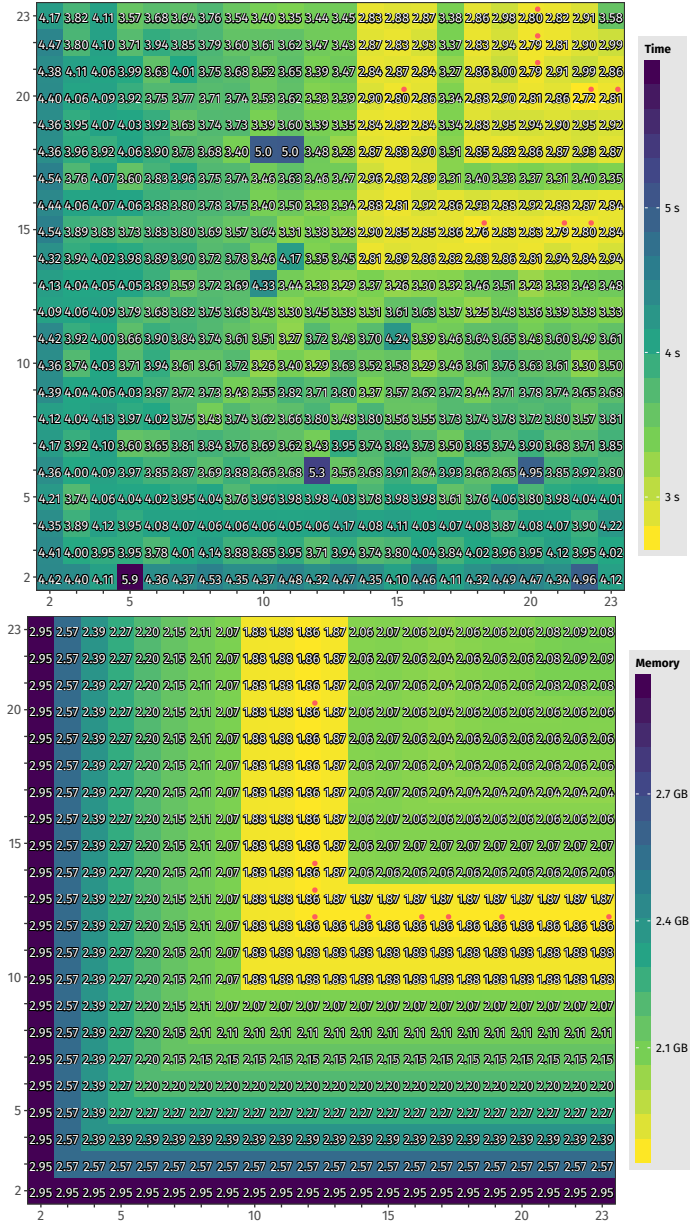
FIGURE 21: Parameter exploration for reverse on RSqueak (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.

FIGURE 22: Parameter exploration for append on Theseus , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



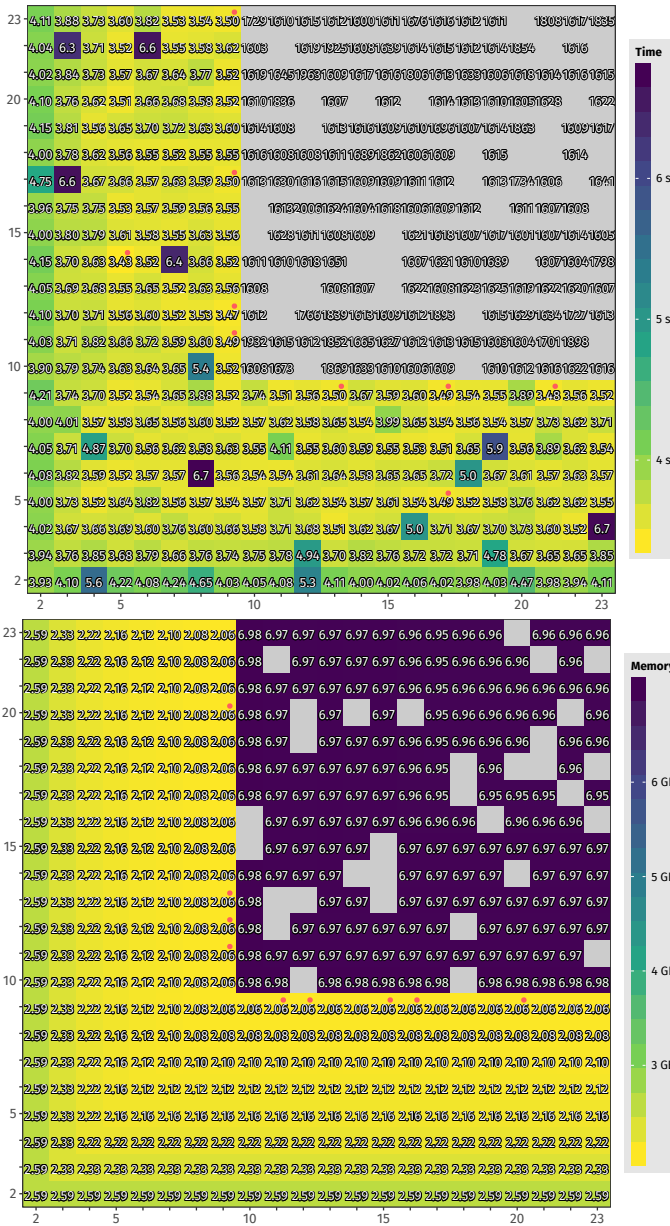



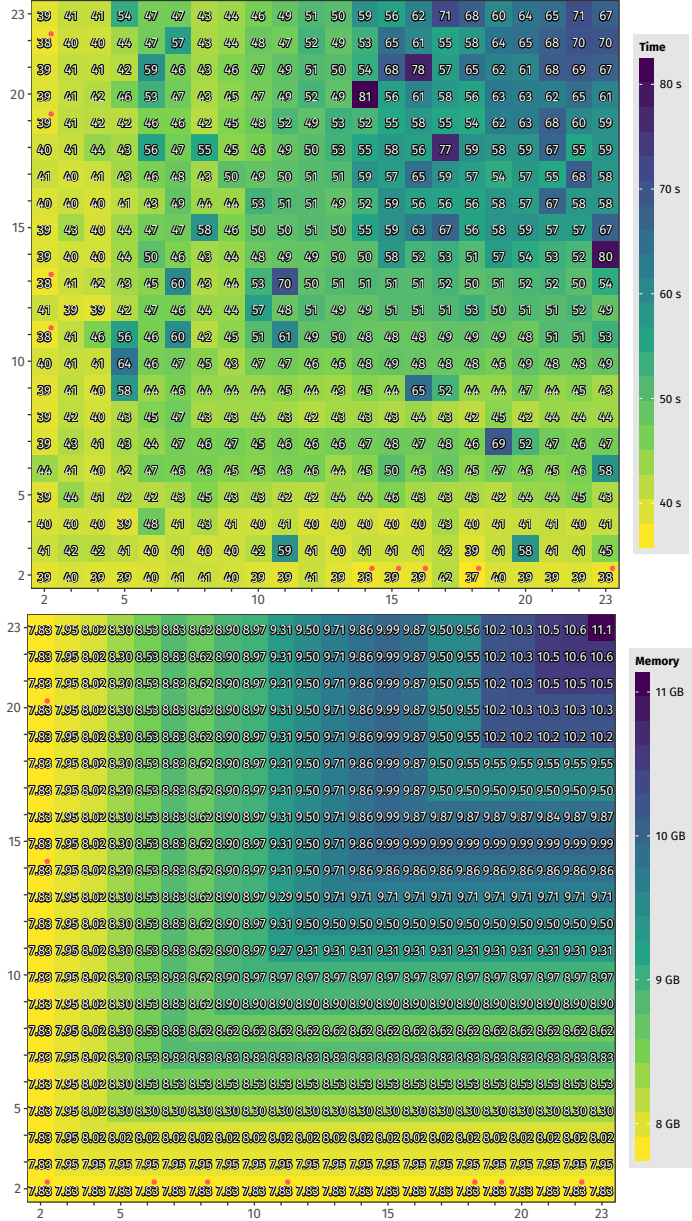
FIGURE 23: Parameter exploration for append on Pycket (optimized) , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.

FIGURE 24: Parameter exploration for append on RSqueak (optimized) \square , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



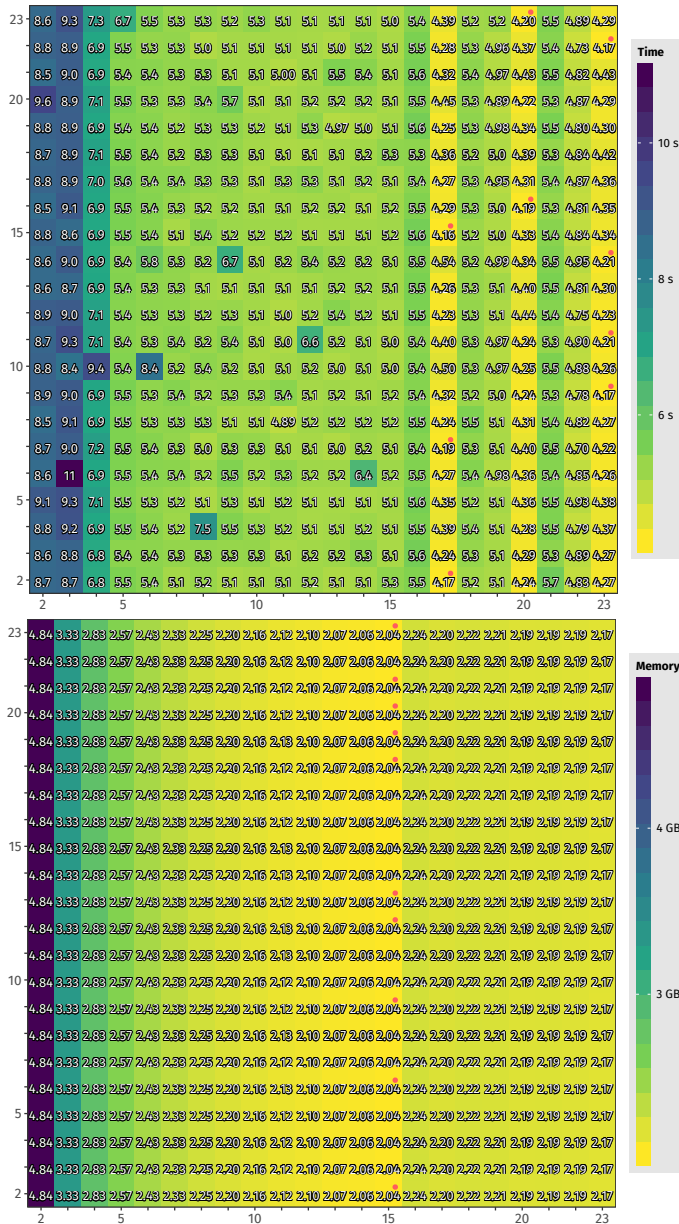




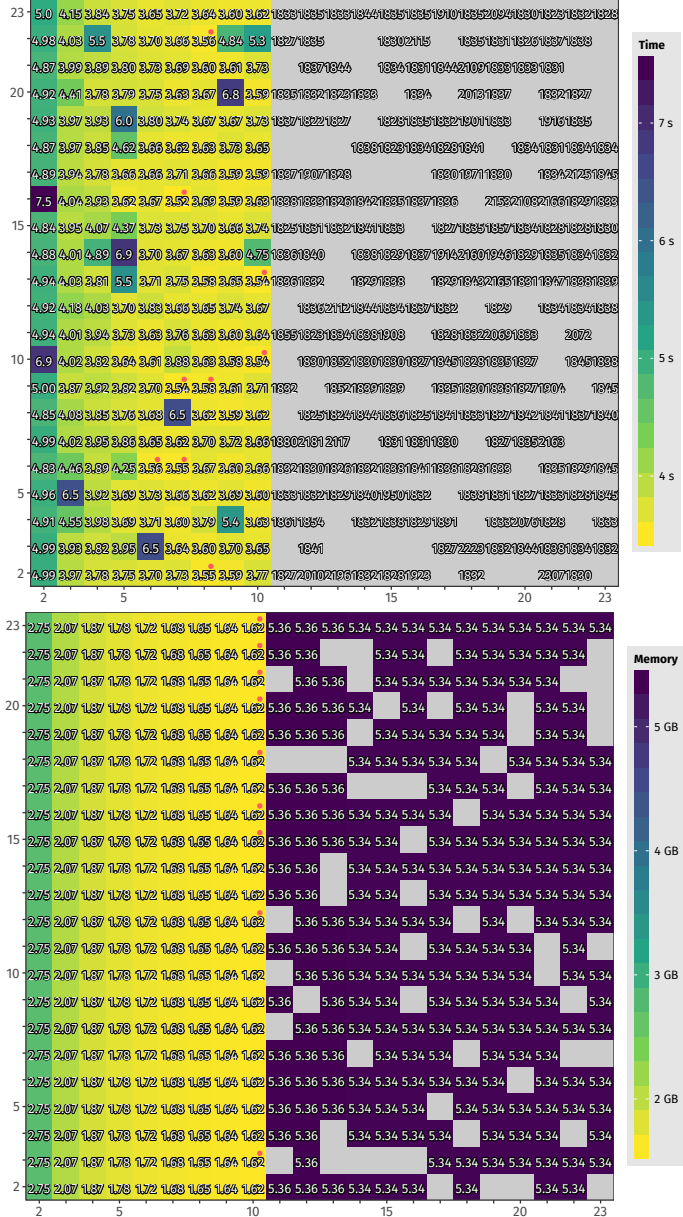
FIGURE 25: Parameter exploration for append on Theseus , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.

FIGURE 26: Parameter exploration for append on Pycket (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.



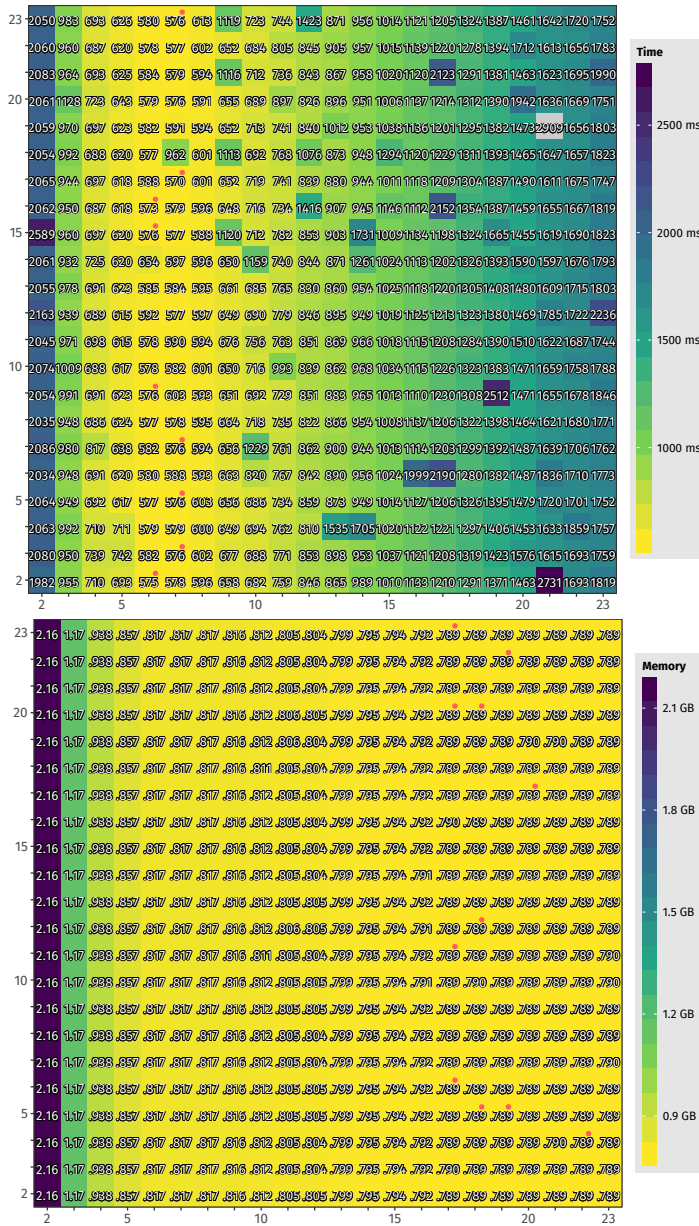

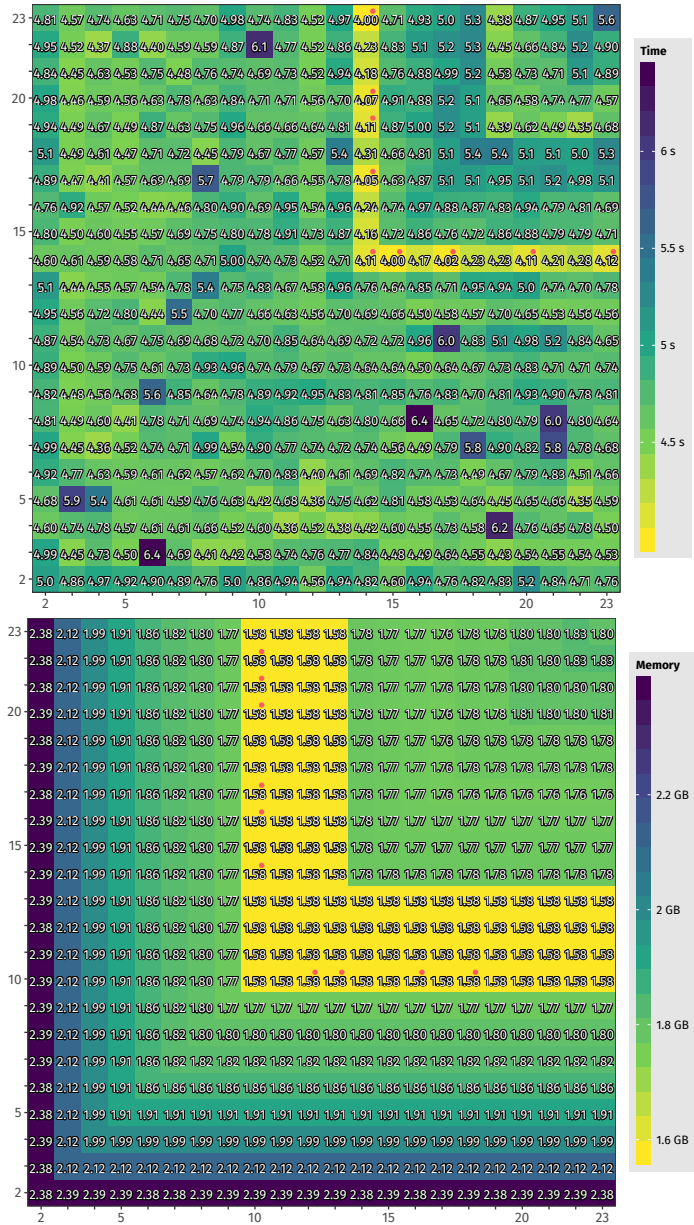


FIGURE 27: Parameter exploration for append on RSqueak (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.

FIGURE 28: Parameter exploration for map on Theseus , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



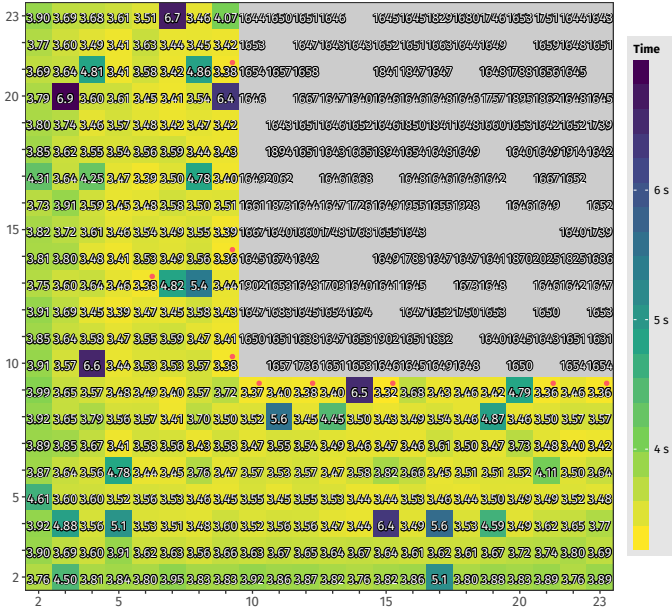

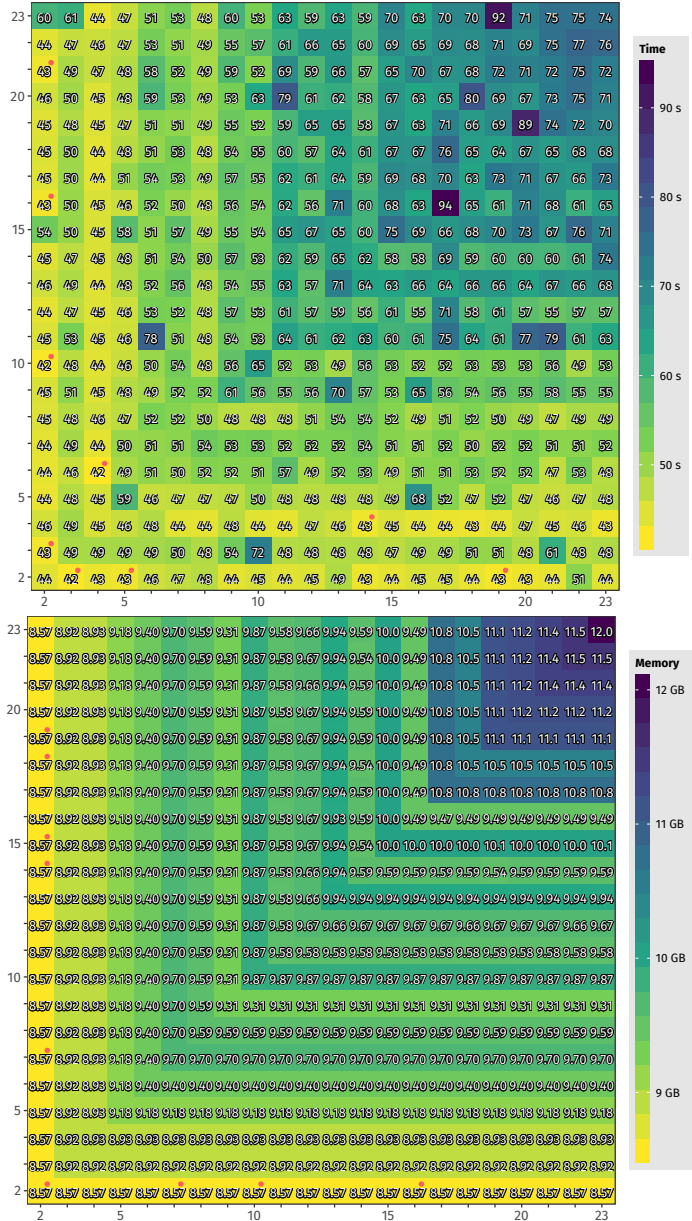


FIGURE 29: Parameter exploration for map on Pycket (optimized) , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



FIGURE 30: Parameter exploration for map on RSqueak (optimized) , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



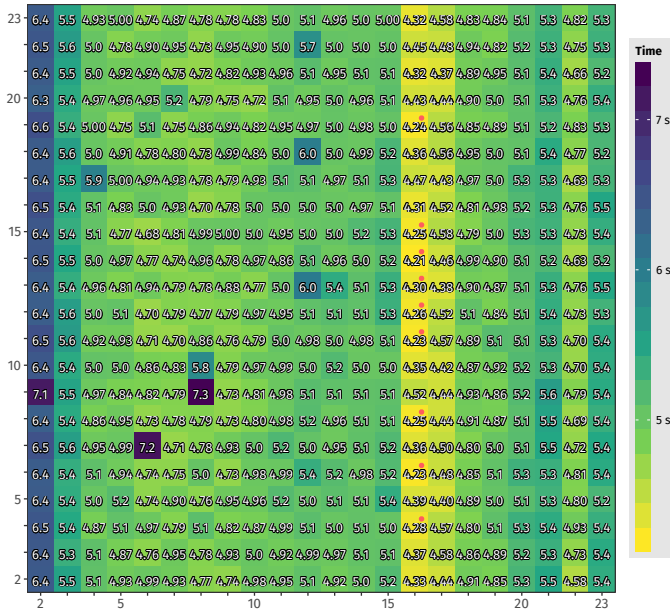



FIGURE 31: Parameter exploration for map on Theseus , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.

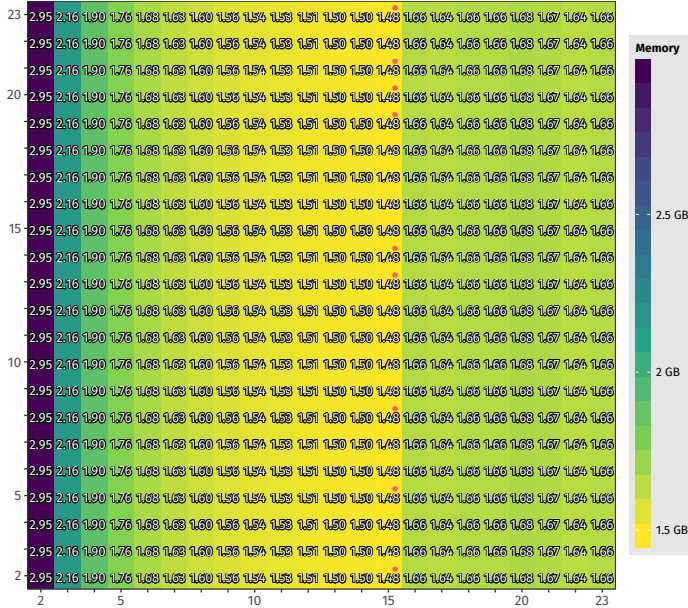

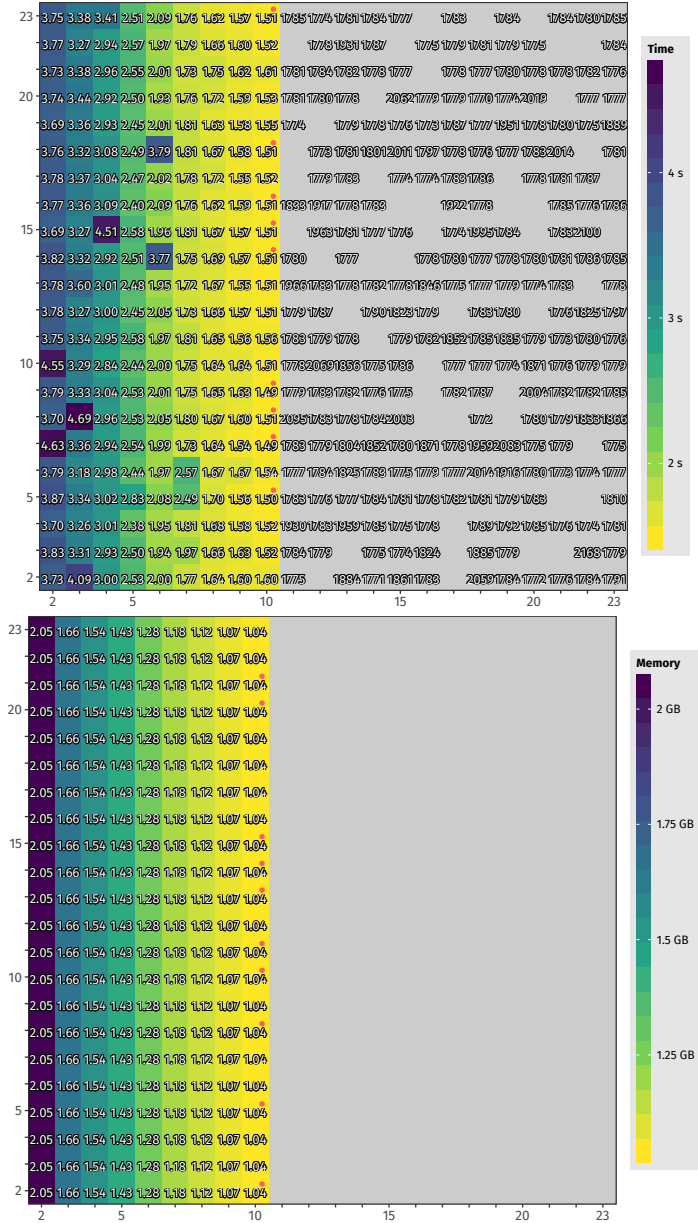


FIGURE 32: Parameter exploration for map on Pycket (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.



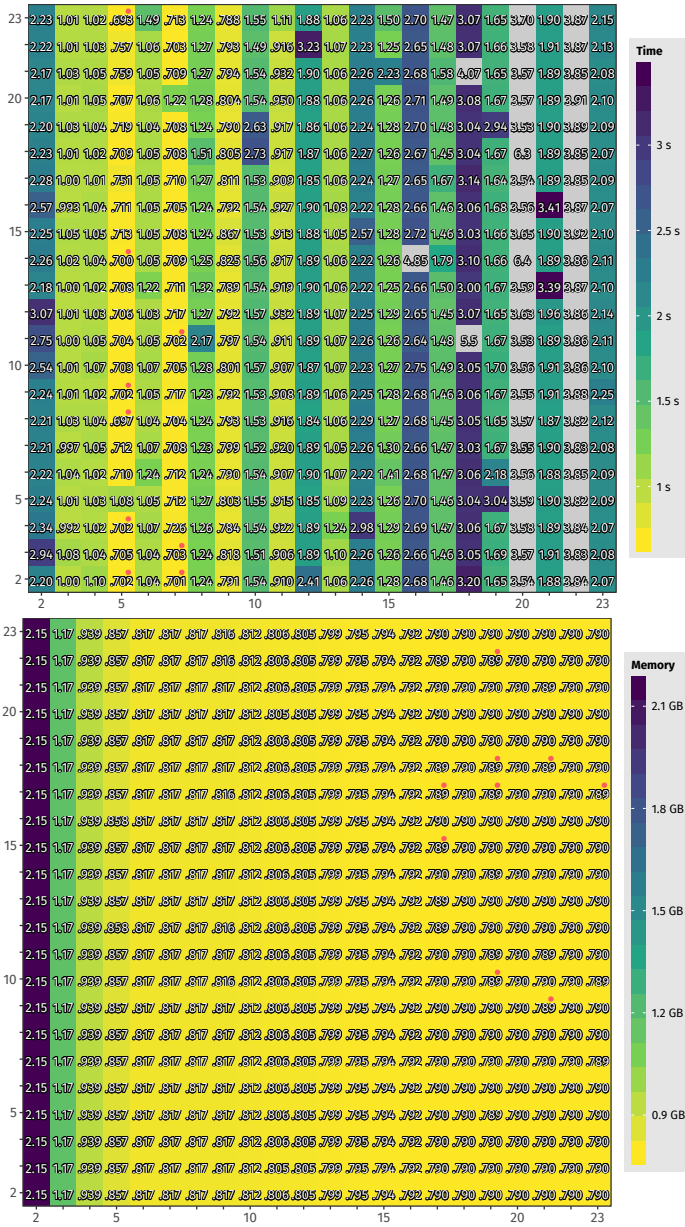




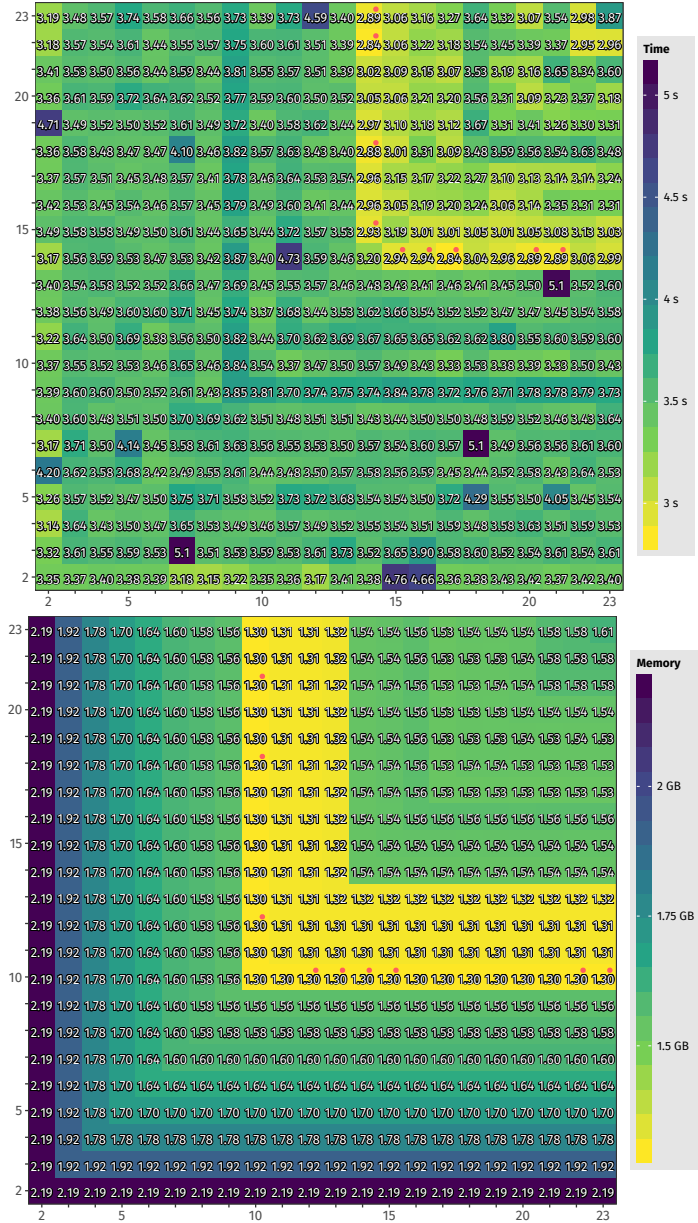
FIGURE 33: Parameter exploration for map on RSqueak (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.

FIGURE 34: Parameter exploration for filter on Theseus , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



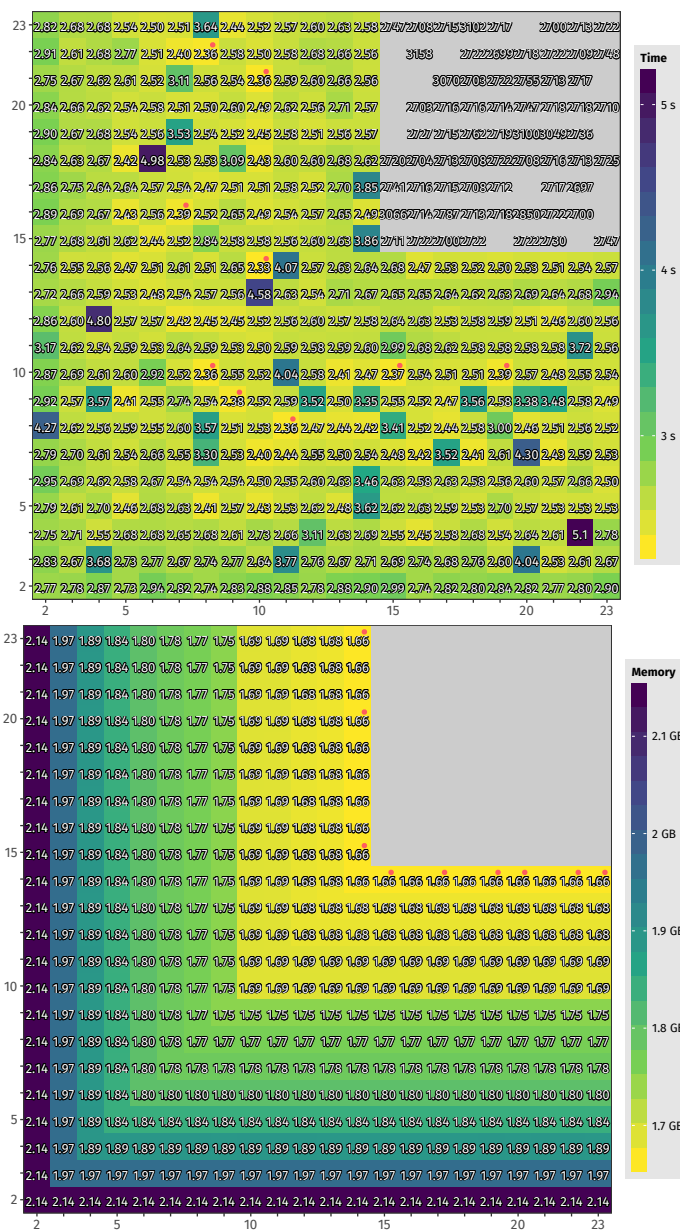




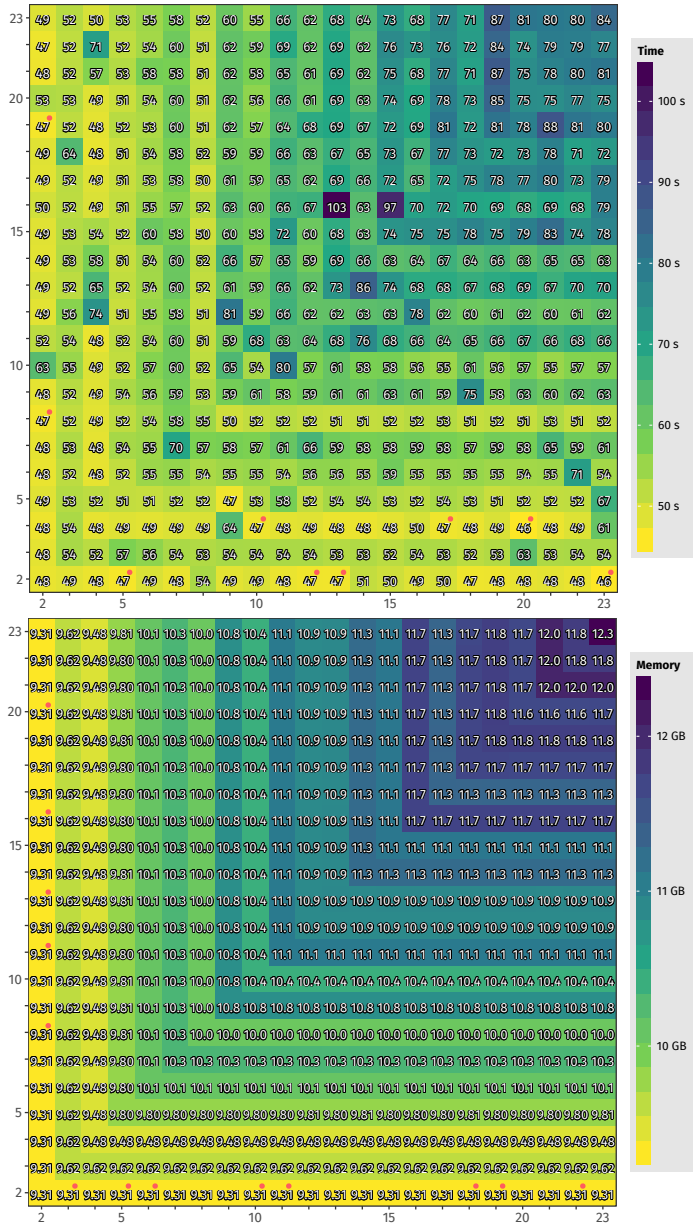
FIGURE 35: Parameter exploration for filter on Pycket (optimized) , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.

FIGURE 36: Parameter exploration for filter on RSqueak (optimized) , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.



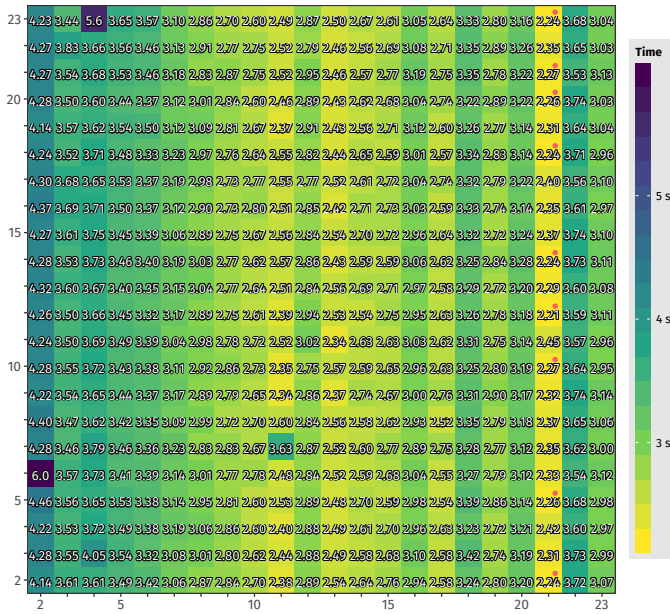



FIGURE 37: Parameter exploration for filter on Theseus , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.

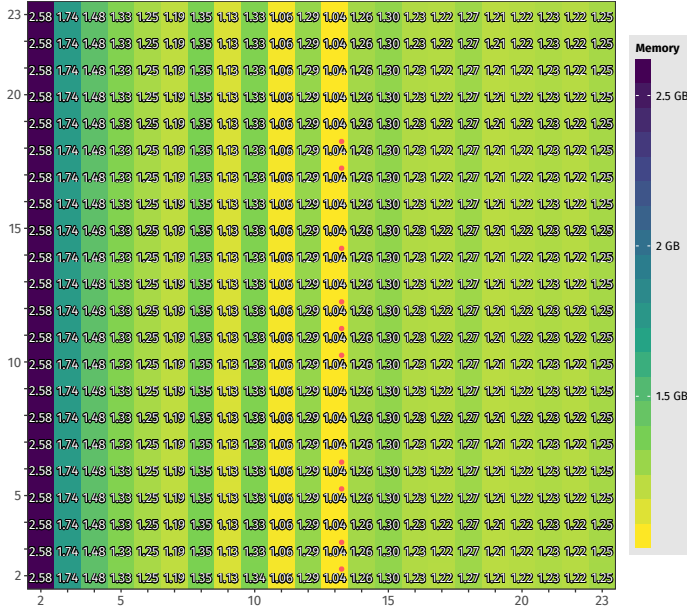

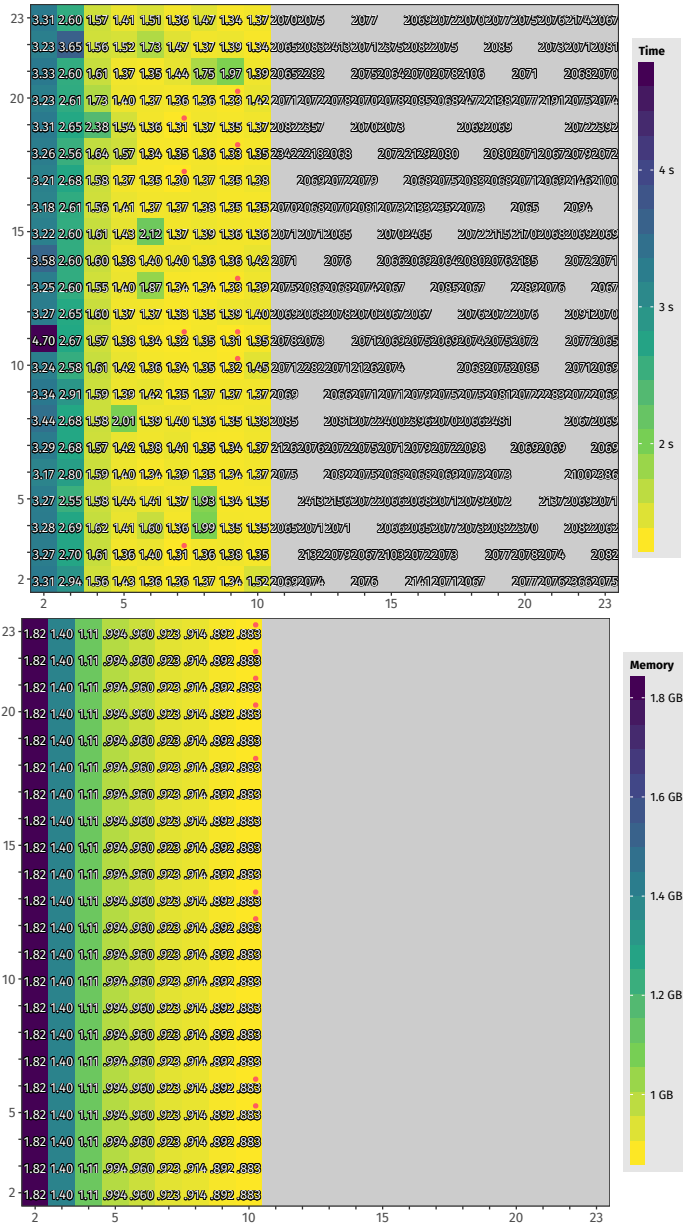


FIGURE 38: Parameter exploration for filter on Pycket (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



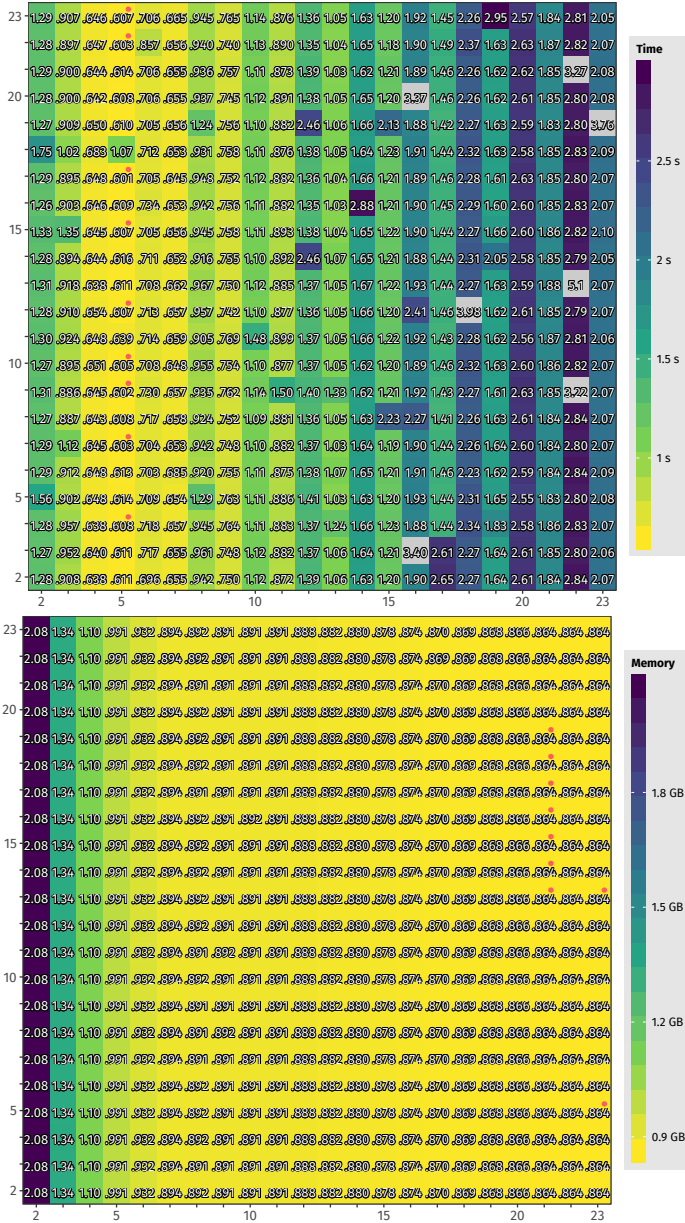




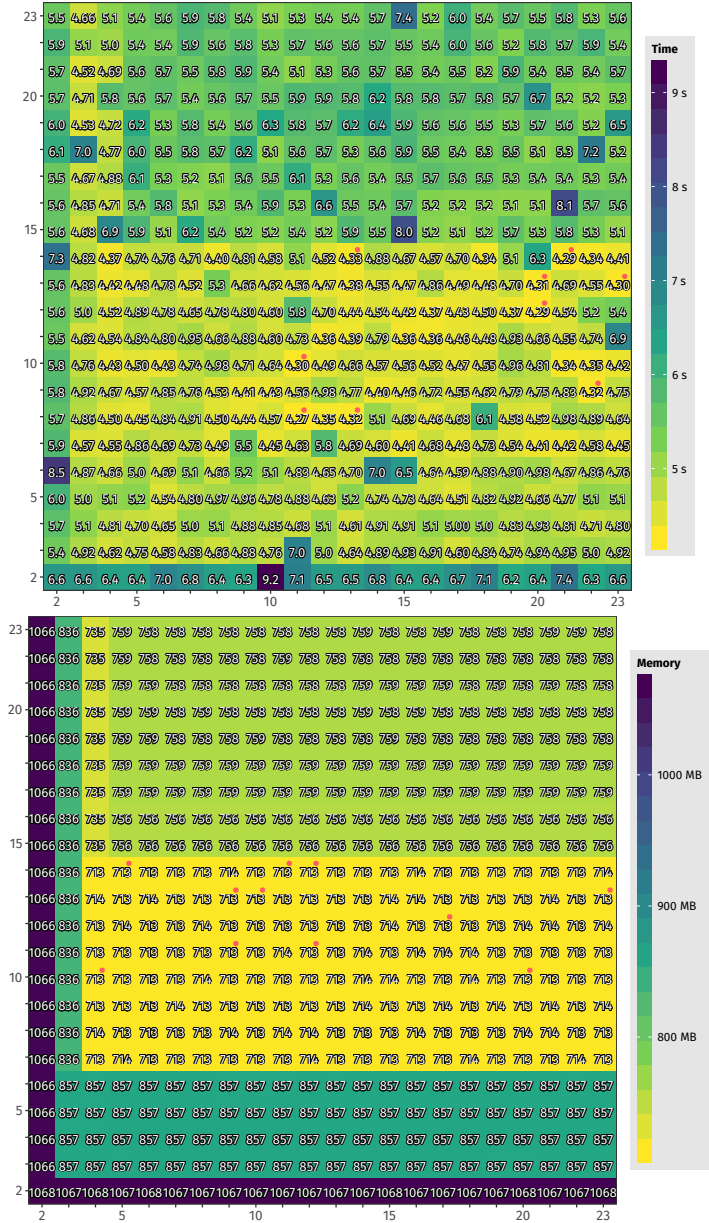
FIGURE 39: Parameter exploration for filter on RSqueak (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.

FIGURE 40: Parameter exploration for tree on Theseus , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.



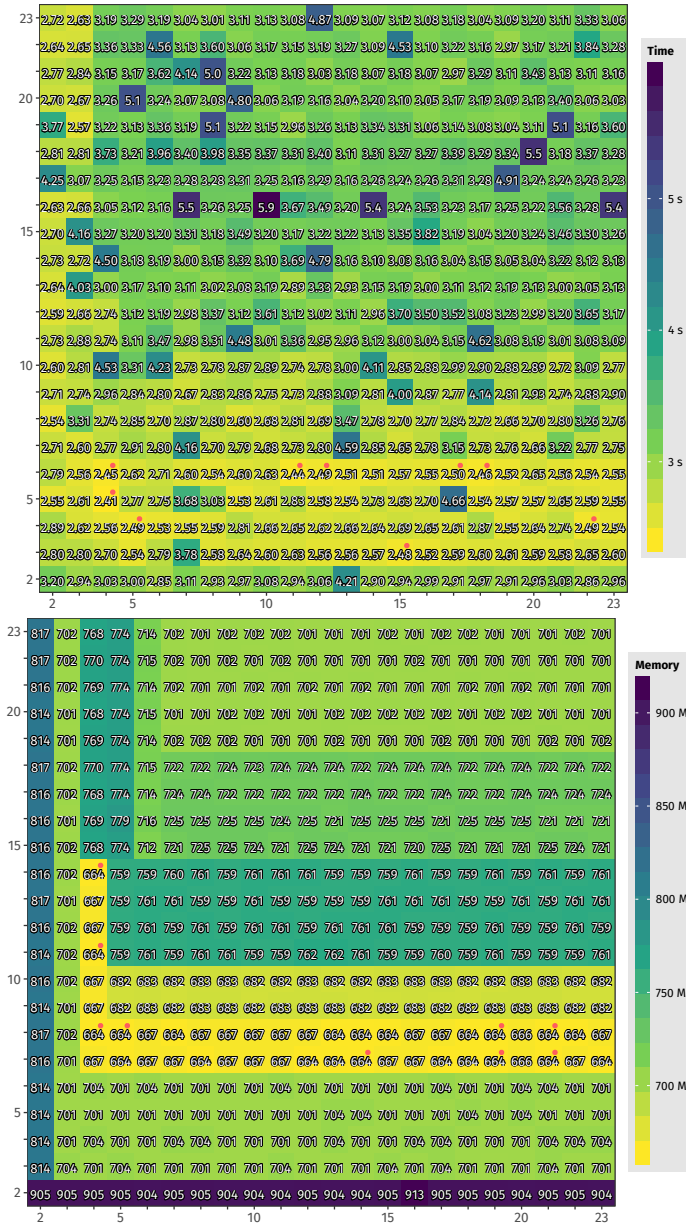

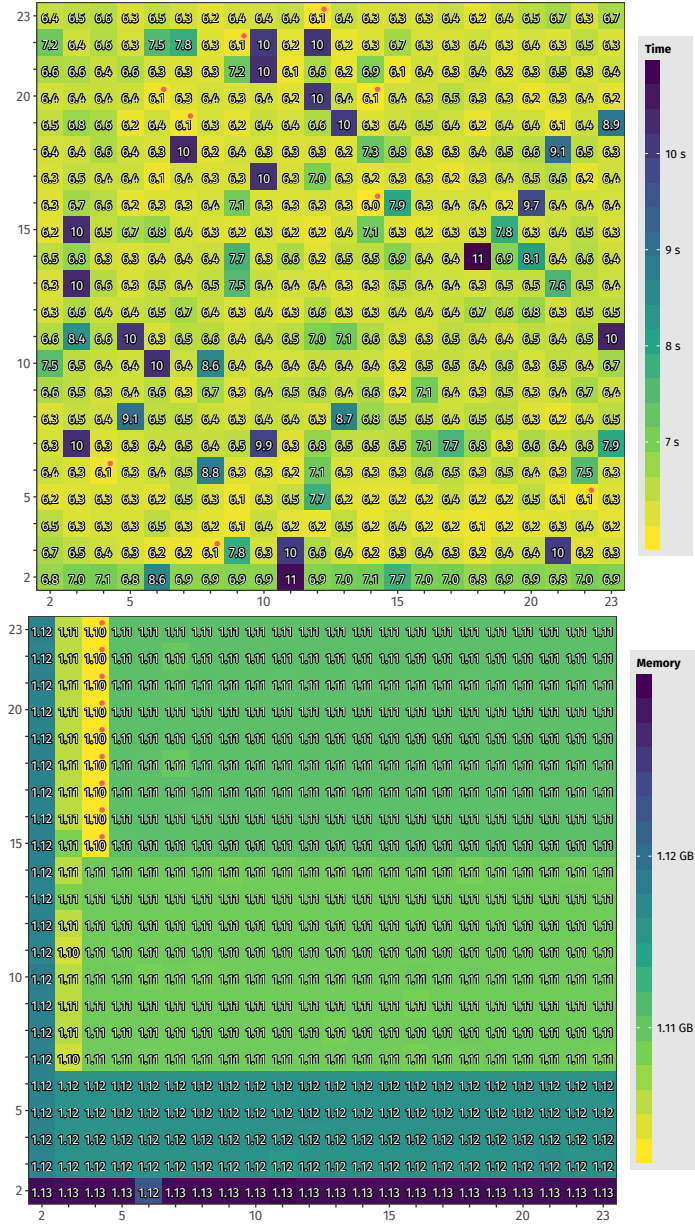


FIGURE 41: Parameter exploration for tree on Pycket (optimized) \square , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.

FIGURE 42: Parameter exploration for tree on RSqueak (optimized) , numeric elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.



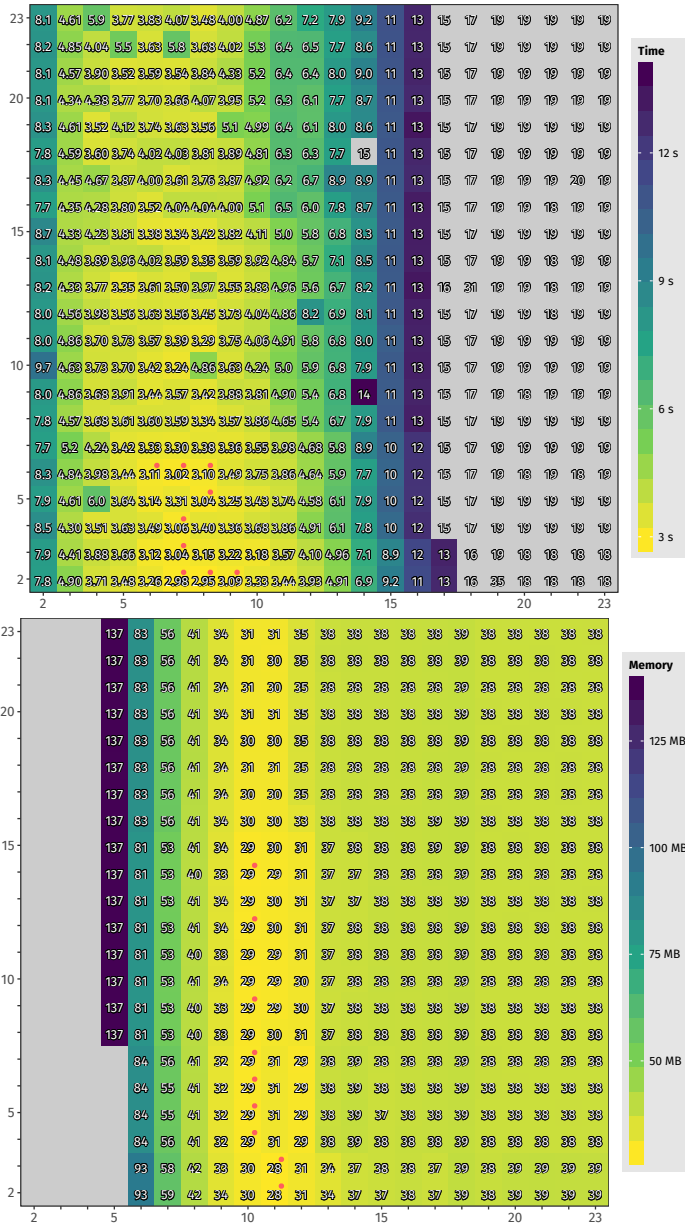

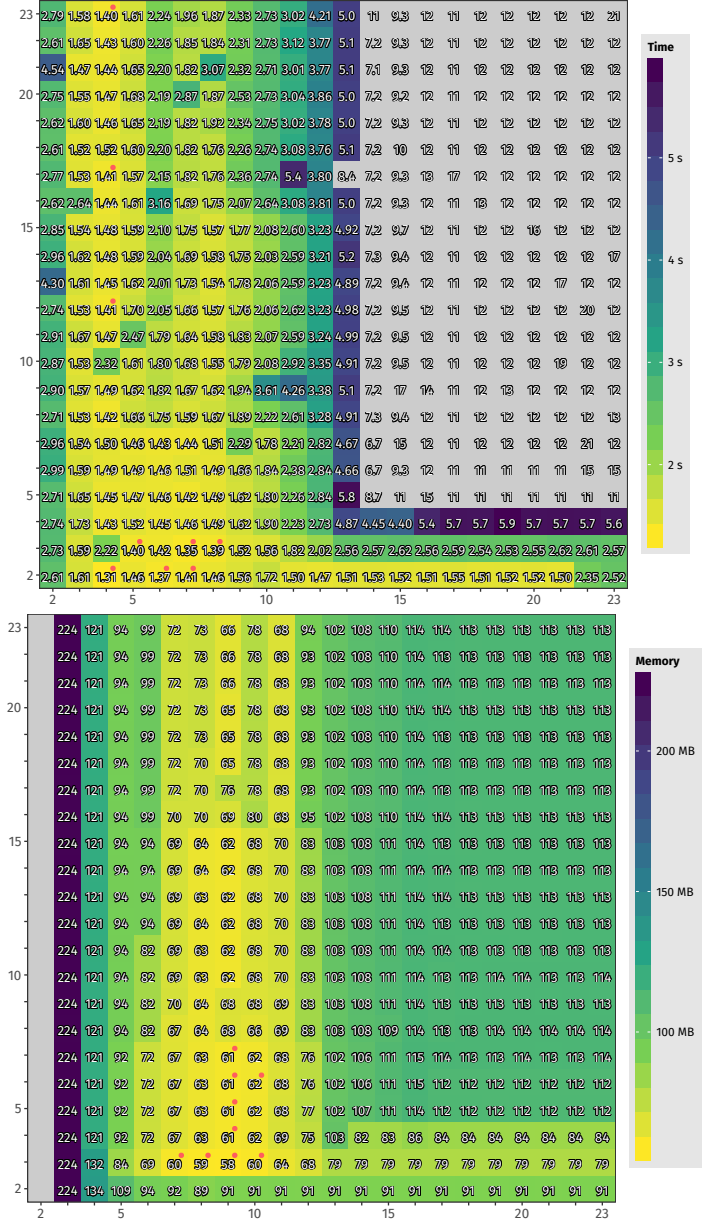


FIGURE 43: Parameter exploration for tree on Theseus \diamond , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate *measured* values.

FIGURE 44: Parameter exploration for tree on Pycket (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.



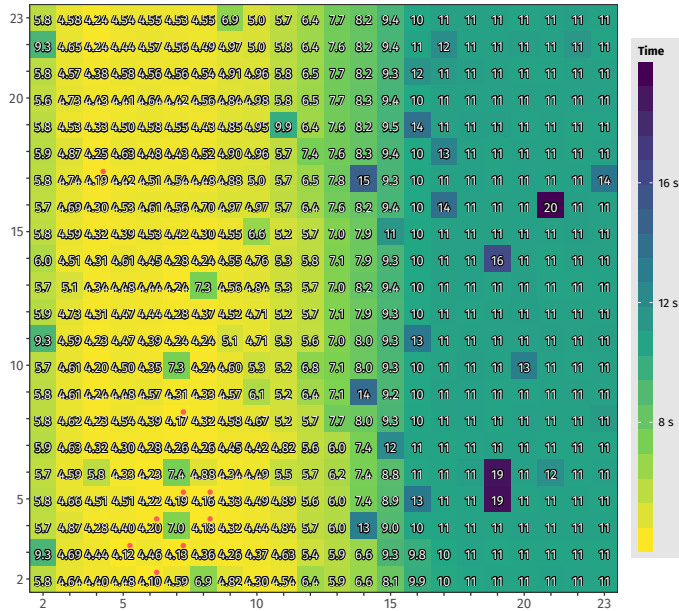



FIGURE 45: Parameter exploration for tree on RSqueak (optimized) , niladic elements variant. Top: execution time; bottom: memory consumption. Colors and numbers indicate measured values.

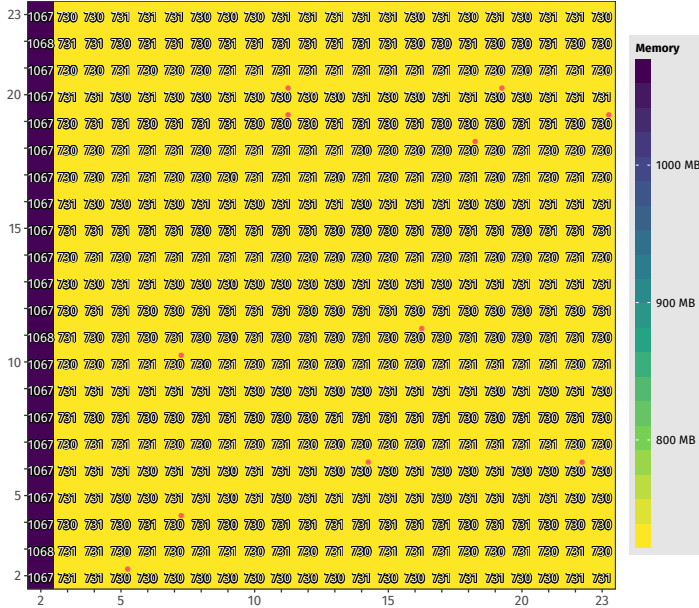
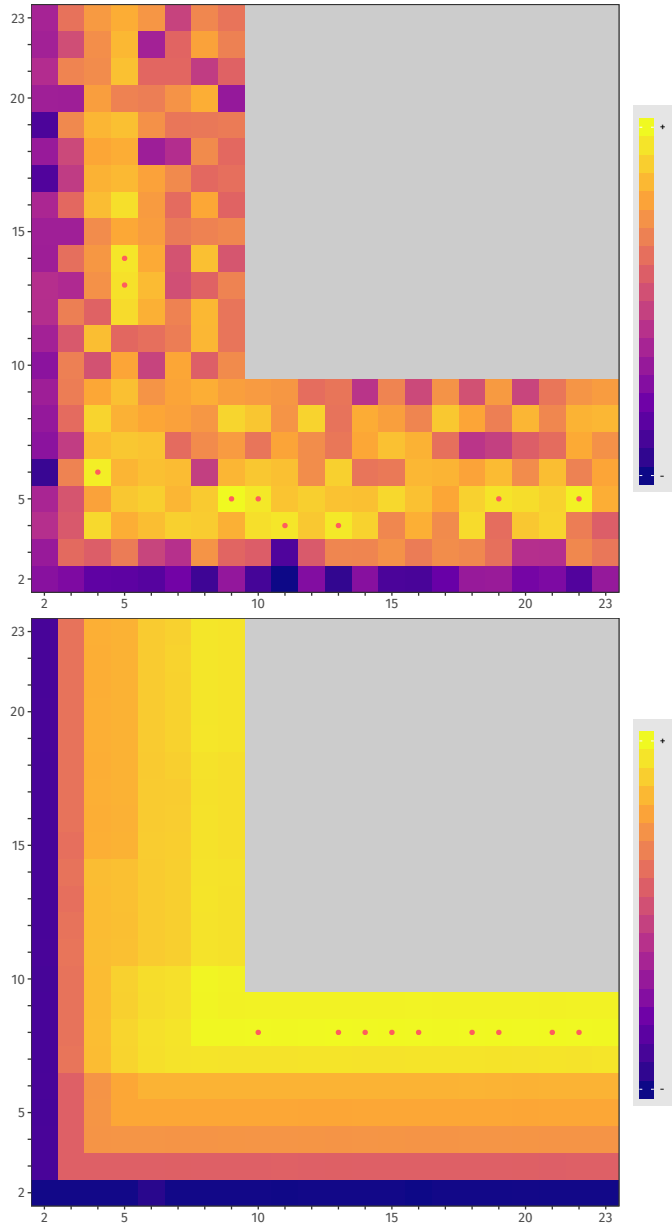


FIGURE 46: Accumulated parameter exploration over all benchmarks, numeric elements variant. Top: execution time; bottom: memory consumption. Colors indicate accumulated rank within original *benchmark*.



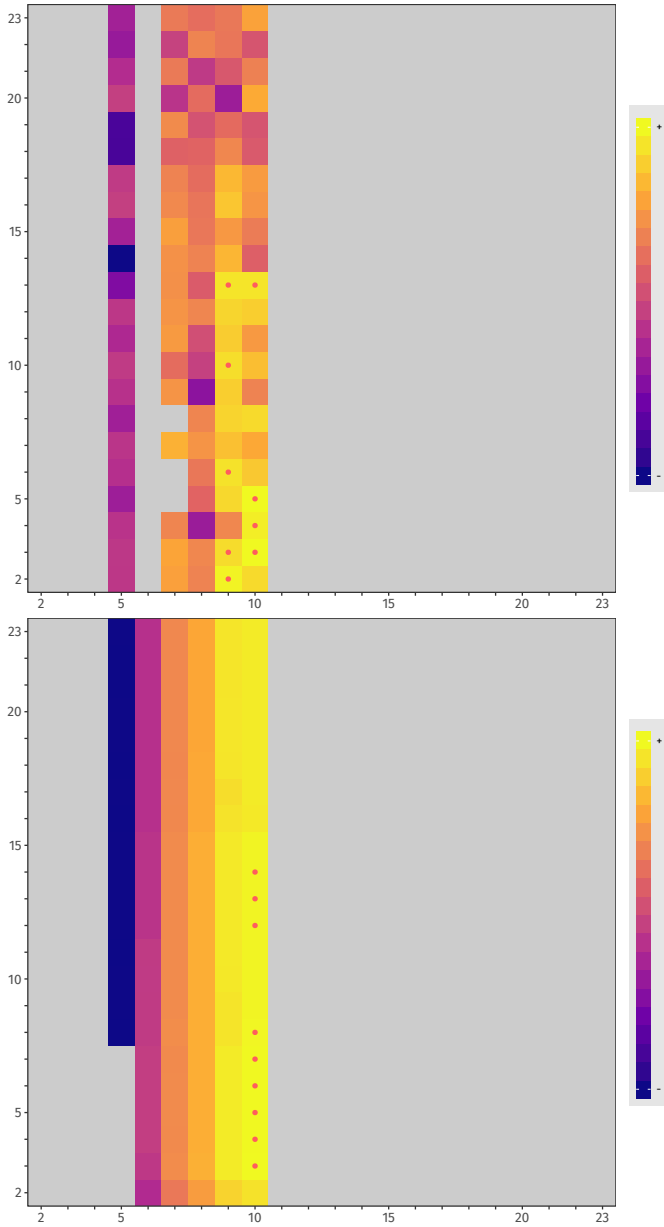
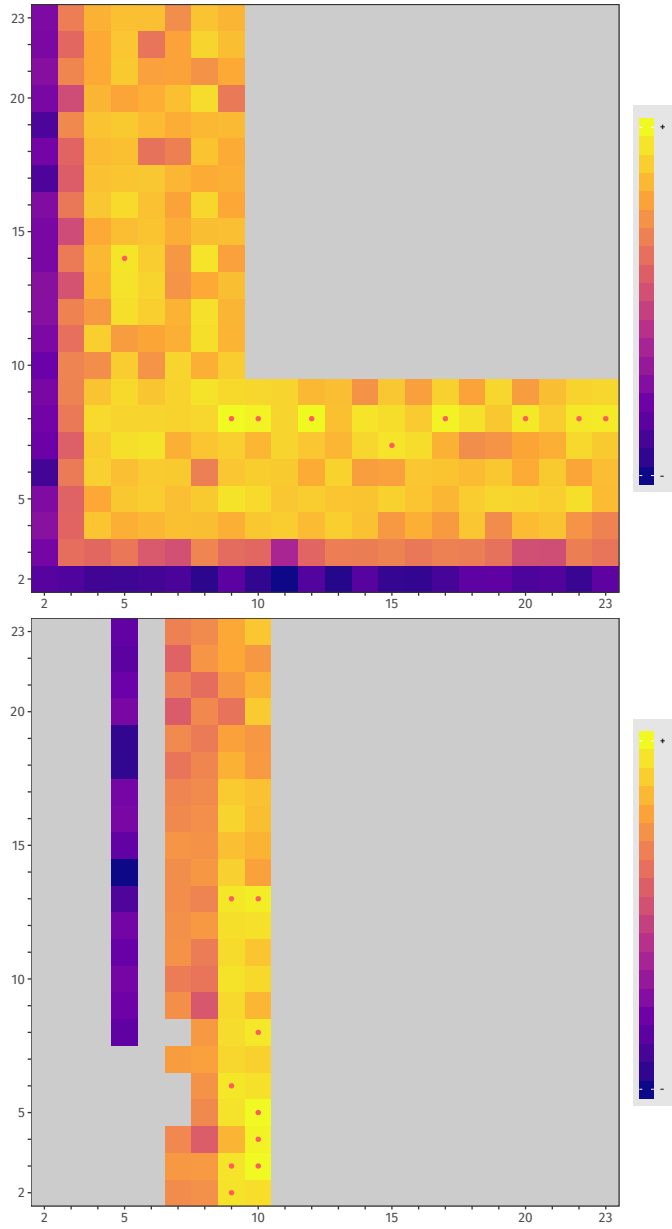


FIGURE 47: Accumulated parameter exploration over all benchmarks, niladic elements variant. Top: execution time; bottom: memory consumption. Colors indicate accumulated rank within original *benchmark*.

FIGURE 48: Accumulated parameter exploration over all benchmarks. Top: numeric elements; bottom: niladic elements. Colors indicate accumulated rank within original *variant*.



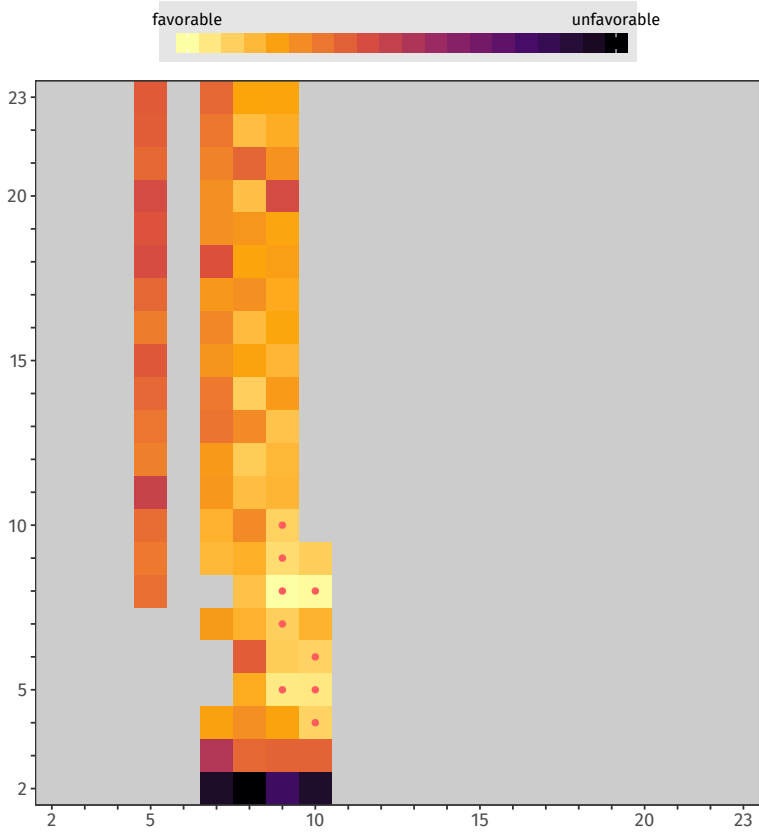


FIGURE 49: Accumulated parameter exploration over all benchmarks, overall result. Accumulation favors results of numeric elements over niladic elements 3 : 1. Colors indicate accumulated rank within original benchmark. (cf. page 143)

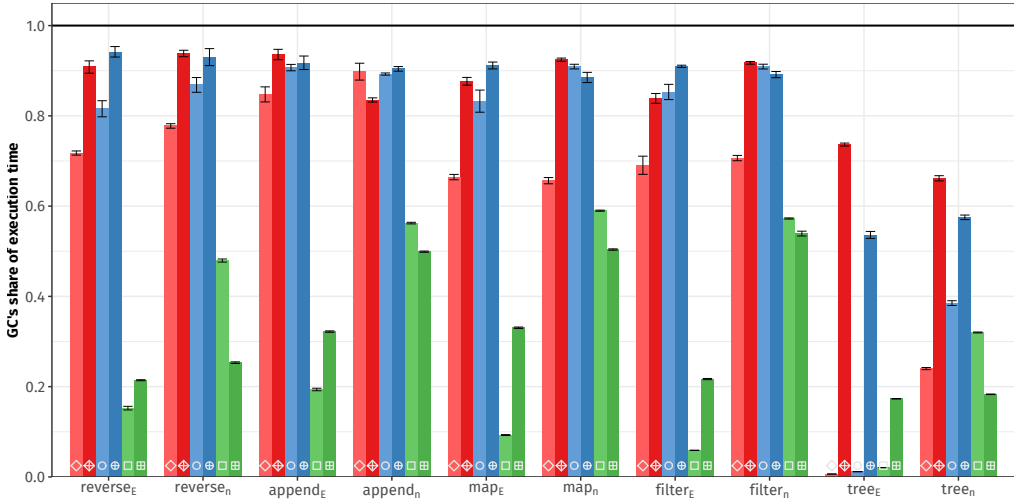


FIGURE 50: GC share results. Each bar shows the relative share of garbage collector (GC) time compared with the complete execution time. We include bootstrapped [32] confidence intervals showing the 95 % confidence level.

B.3 DETAILED RESULTS

In table 10 and table 11 we provide measurements of execution time and memory consumption of all implementations and benchmarks, respectively. We provide relative variants for optimized/not-optimized implementation pairs in table 13 and table 14, respectively.

Additionally, table 12 gives absolute garbage collection times for all measured implementations that support this metric, and table 15 gives the share of garbage collection for these implementations; figure 50 contains this information in graphical form for the optimized/not-optimized implementation pairs. This metric is informational; we have not drawn conclusions from the respective numbers.

Moreover, the absolute measurements for execution time, memory consumption, and garbage collection time (if applicable) are given in graphical form in figures 51 to 60.

TABLE 10: Absolute execution time for all benchmarks in seconds as averages of ten runs. We include the 95 % confidence interval of the measurements. Measurements for all implementations. Smaller numbers are better.














	reverse		append		map		filter		tree	
	niladic	numeric	niladic	numeric	niladic	numeric	niladic	numeric	niladic	numeric
Theseus 	127± 0	341± 2	2252± 35	3026± 50	2688± 17	2216± 18	1876± 34	1293± 7	3496± 17	4027± 19
Theseus (not optimized) 	3014± 35	2611± 14	5610± 57	2772± 13	4921± 43	4018± 12	2744± 19	3277± 10	10690± 39	7093± 42
Pycket (optimized) 	116± 2	857± 14	4346± 29	4258± 9	1778± 49	4198± 21	1521± 24	3135± 15	2100± 6	2529± 19
Pycket (original) 	2860± 32	2251± 40	5268± 69	5122± 25	4686± 29	4327± 37	3840± 4	3385± 15	2477± 19	2856± 12
Racket 	4560±216	4331±130	7325± 24	7389± 27	6930±201	9608±175	8535±271	9904±181	3086± 48	2919± 6
RSqueak (optimized) 	310± 3	586± 3	686± 3	39932±100	1507± 7	43622± 95	1078± 6	49024±134	4318± 6	5881± 14
RSqueak (original) 	11553± 44	12093± 35	26816±107	46670±137	27441±129	46137± 90	14743± 44	48960±69	57834±223	57372±188
Squeak 	4136± 8	4083± 16	8954± 16	28706± 37	9486± 11	28994± 36	5556± 15	34074±122	9068± 18	8973± 12
PyPy 	4087±106	4166±108	6377±116	6370± 99	6965± 68	6923± 36	3292± 17	3260± 13	6226± 32	6208± 22
Python 	26301±251	25927±137	39554±189	39011±361	40603±422	40211±228	35103±196	34443±335	113630±333	112301±221
MLton 	992± 12	988± 8	825± 12	819± 9	2090± 30	2125± 20	830± 8	824± 7	942± 4	941± 3
OCaml 	1704± 10	1719± 13	24844± 31	24920± 42	32510± 70	32379±123	12508± 17	12260± 25	2596± 6	2596± 6
SML/NJ 	13646± 7	13648± 13	279976± 61	280352±310	179448± 89	179966±312	76988± 58	77022± 53	2270± 23	2317± 3

TABLE II: Absolute memory consumption for all benchmarks in megabytes as averages of ten runs. We include the 95 % confidence interval of the measurements. Measurements for all implementations. Smaller numbers are better.

	reverse		append		map		filter		tree	
	niladic	numeric	niladic	numeric	niladic	numeric	niladic	numeric	niladic	numeric
Theseus	119.1to0	443.4to0	1110.0to0	1449.2to0	1041.1to0	1063.5to0	1154.4to0	807.9to0	29.8to0	721.4to0
Theseus (not optimized)	1863.7to0	1557.7to0	3408.0to0	2793.9to1	2794.9to0	2169.0to0	2294.3to0	1937.2to0	1103.7to0	1073.7to0
Pycket (optimized)	143.3to0	773.1to0	165.44to0	2131.9to0	1067.3to0	1978.5to0	903.9to0	1808.1to0	65.8to0	666.3to1
Pycket (original)	1578.4to0	1425.4to0	3423.6to1	3272.6to0	2494.0to1	2354.5to0	2152.4to1	1960.7to0	812.9to1.5	1020.2to0.8
Racket	1604.7to7	1604.1to5	2830.7to7	3308.0to6	2229.7to7	2504.0to8	2538.1to.5	2554.1to.5	306.9to0.6	281.3to0.6
RSqueak (optimized)	831.4to2	1629.4to0	831.2to2	8830.3to0	831.5to2	9816.5to1	913.1to2	10289.3to0	730.5to0	1135.5to2
RSqueak (original)	4062.3to1	4988.2to1	5052.4to1	10990.8to1	5298.7to2	10940.0to2	5028.3to1	12157.0to.5	1228.0to1.1	1301.3to.5
Squeak	1521.2to0	1521.2to0	1938.7to0	4940.6to0	1969.2to0	5007.5to0	1862.1to0	5935.2to0	413.4to0	414.0to0
PyPy	3474.0to2	3474.2to2	4562.0to2	4562.0to2	3477.7to2	3477.8to2	2933.0to2	2933.0to2	1170.0to3	1170.1to3
Python	3727.7to1	3727.6to0	3727.6to0	3727.6to1	2487.6to0	2487.6to0	2332.6to0	2332.6to1	592.0to0	592.0to0
MLton	824.9to0	824.9to0	132.9to0	1329.7to0	1672.6to0	1672.6to0	1310.5to0	1310.5to0	388.9to0	388.9to0
OCaml	826.5to0	826.5to0	177.9to0	1772.9to0	1599.0to0	1599.0to0	1248.0to0	1248.0to0	149.6to0	149.6to0
SML/NJ	319.1to0	319.0to0	926.1to0	926.1to0	770.3to0	770.3to0	486.3to0	486.4to0	127.6to0	127.6to0

TABLE 12: Absolute garbage collection time for all benchmarks in seconds as averages of ten runs. We include the 95 % confidence interval of the measurements. Note that not all implementations provide this measurement. Smaller numbers are neither better nor worse.

	reverse		append		map		filter		tree	
	niladic	numeric	niladic	numeric	niladic	numeric	niladic	numeric	niladic	numeric
Theseus	91± 1	265± 2	1909± 35	2716± 30	1786± 16	1453± 15	1296± 40	913± 8	22± 1	967± 11
Theseus (not optimized)	2739± 41	2449± 18	5250± 57	2315± 12	4314± 34	3715± 13	2301± 32	3007± 8	7871± 31	469± 41
Pycket (optimized)	95± 2	745± 13	3942± 27	3800± 9	1480± 29	3817± 18	1297± 23	2851± 15	24± 1	975± 14
Pycket (original)	2692± 26	2094± 36	4832± 73	4630± 23	4271± 35	3831± 48	3493± 11	3017± 26	1328± 21	1643± 16
Racket	4257± 216	4026± 131	5820± 24	5468± 21	4662± 200	6555± 180	6286± 249	6208± 120	2188± 45	2028± 9
RSqueak (optimized)	47± 1	281± 2	133± 2	22459± 68	140± 2	25733± 30	64± 0	28071± 24	89± 0	188± 6
RSqueak (original)	2476± 13	3061± 26	8635± 38	23290± 39	9067± 35	23244± 77	3197± 22	26391± 274	10014± 20	10503± 26
Squeak	3653± 3	3601± 11	8002± 15	25787± 39	8140± 6	25687± 18	4378± 9	30333± 109	2923± 10	2829± 6
PyPy	3786± 80	3837± 74	5769± 40	5779± 35	6089± 143	6050± 59	2665± 68	2615± 54	4344± 30	4340± 19
Python	—	±	—	±	—	±	—	±	—	±
MLton	656± 13	655± 9	274± 4	271± 4	1298± 15	1339± 10	274± 5	275± 4	216± 3	216± 4
OCaml	—	±	—	±	—	±	—	±	—	±
SML/NJ	10378± 90	10404± 43	208348± 307	208469± 476	125708± 333	126223± 372	54578± 158	54649± 169	1044± 29	1038± 17

TABLE 13: Relative execution time for all benchmarks. Execution times of *optimized* implementations are normalized to respective *not optimized* implementations. We include the bootstrapped [32] 95 % confidence interval. We include the geometric mean of all relative execution times per implementation. Smaller numbers are better.

	Theseus ◊	Pycket (optimized) ◯	RSqueak (optimized) ◻
reverse _E	0.042 ^{+0.000} _{-0.000}	0.041 ^{+0.001} _{-0.001}	0.027 ^{+0.000} _{-0.000}
reverse _n	0.131 ^{+0.001} _{-0.001}	0.381 ^{+0.008} _{-0.007}	0.048 ^{+0.000} _{-0.000}
append _E	0.402 ^{+0.006} _{-0.006}	0.825 ^{+0.010} _{-0.010}	0.026 ^{+0.000} _{-0.000}
append _n	1.092 ^{+0.015} _{-0.016}	0.831 ^{+0.004} _{-0.004}	0.856 ^{+0.003} _{-0.003}
map _E	0.546 ^{+0.005} _{-0.005}	0.379 ^{+0.009} _{-0.009}	0.055 ^{+0.000} _{-0.000}
map _n	0.552 ^{+0.004} _{-0.004}	0.970 ^{+0.008} _{-0.008}	0.945 ^{+0.002} _{-0.002}
filter _E	0.684 ^{+0.011} _{-0.011}	0.396 ^{+0.005} _{-0.005}	0.073 ^{+0.000} _{-0.000}
filter _n	0.395 ^{+0.002} _{-0.002}	0.926 ^{+0.005} _{-0.005}	1.001 ^{+0.005} _{-0.005}
tree _E	0.327 ^{+0.002} _{-0.002}	0.848 ^{+0.006} _{-0.006}	0.075 ^{+0.000} _{-0.000}
tree _n	0.568 ^{+0.003} _{-0.003}	0.886 ^{+0.006} _{-0.006}	0.103 ^{+0.000} _{-0.000}
geometric mean	0.360	0.505	0.124

TABLE 14: Relative memory consumption for all benchmarks. Memory consumption of *optimized* implementations is normalized to respective *not optimized* implementations. We include the bootstrapped [32] 95 % confidence interval. We include the geometric mean of all relative execution times per implementation. Smaller numbers are better.

	Theseus ◊	Pycket (optimized) ◯	RSqueak (optimized) ◻
reverse _E	0.064 ^{+0.000} _{-0.000}	0.091 ^{+0.000} _{-0.000}	0.205 ^{+0.000} _{-0.000}
reverse _n	0.286 ^{+0.000} _{-0.000}	0.542 ^{+0.000} _{-0.000}	0.327 ^{+0.000} _{-0.000}
append _E	0.326 ^{+0.000} _{-0.000}	0.483 ^{+0.000} _{-0.000}	0.165 ^{+0.000} _{-0.000}
append _n	0.518 ^{+0.000} _{-0.000}	0.651 ^{+0.000} _{-0.000}	0.803 ^{+0.000} _{-0.000}
map _E	0.373 ^{+0.000} _{-0.000}	0.428 ^{+0.000} _{-0.000}	0.157 ^{+0.000} _{-0.000}
map _n	0.490 ^{+0.000} _{-0.000}	0.840 ^{+0.000} _{-0.000}	0.897 ^{+0.000} _{-0.000}
filter _E	0.503 ^{+0.000} _{-0.000}	0.420 ^{+0.000} _{-0.000}	0.182 ^{+0.000} _{-0.000}
filter _n	0.417 ^{+0.000} _{-0.000}	0.922 ^{+0.000} _{-0.000}	0.846 ^{+0.000} _{-0.000}
tree _E	0.027 ^{+0.000} _{-0.000}	0.081 ^{+0.000} _{-0.000}	0.595 ^{+0.000} _{-0.000}
tree _n	0.672 ^{+0.000} _{-0.000}	0.653 ^{+0.000} _{-0.000}	0.873 ^{+0.000} _{-0.000}
geometric mean	0.271	0.403	0.398

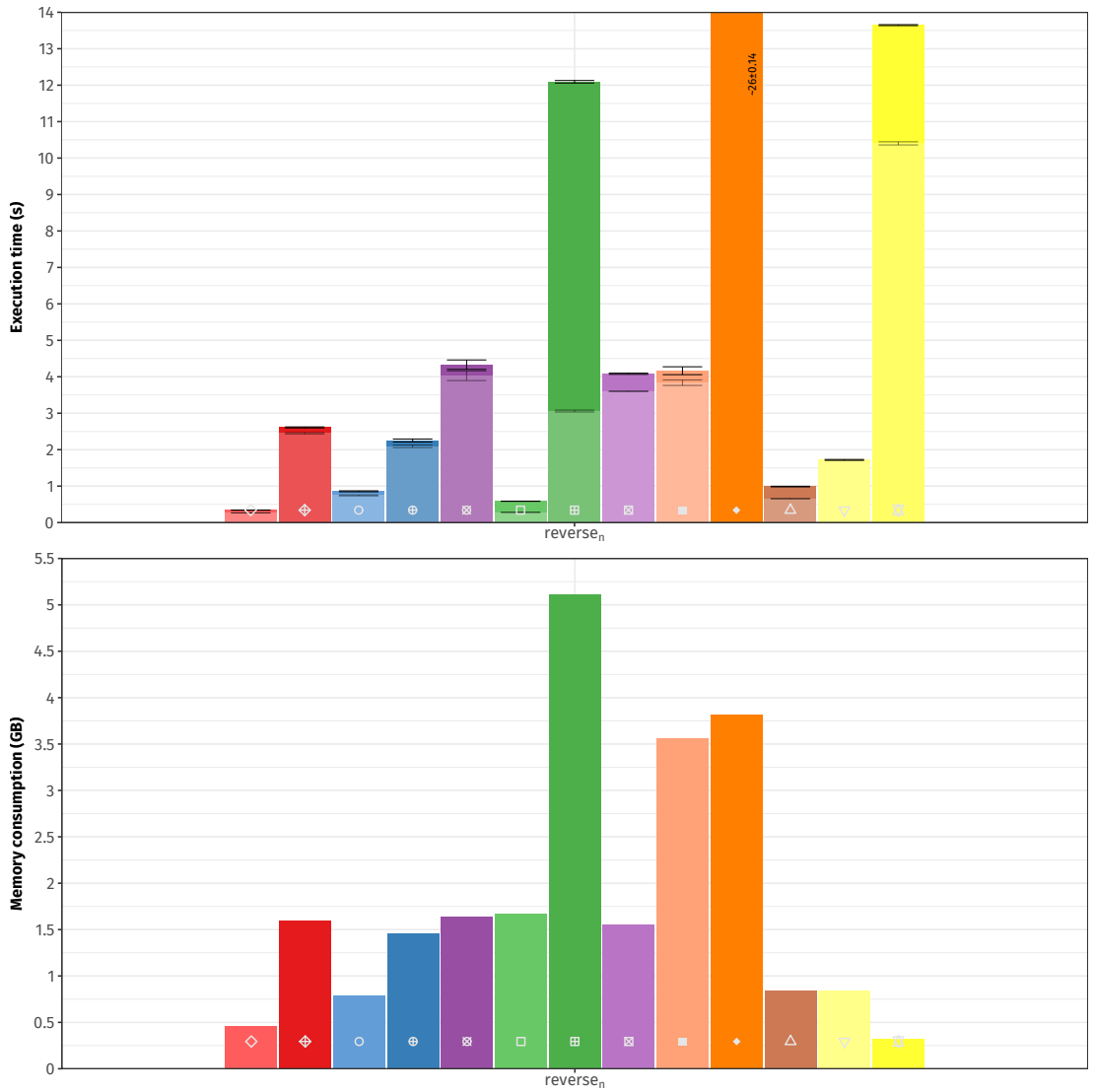


FIGURE 51: Results for benchmark reverse, numeric elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

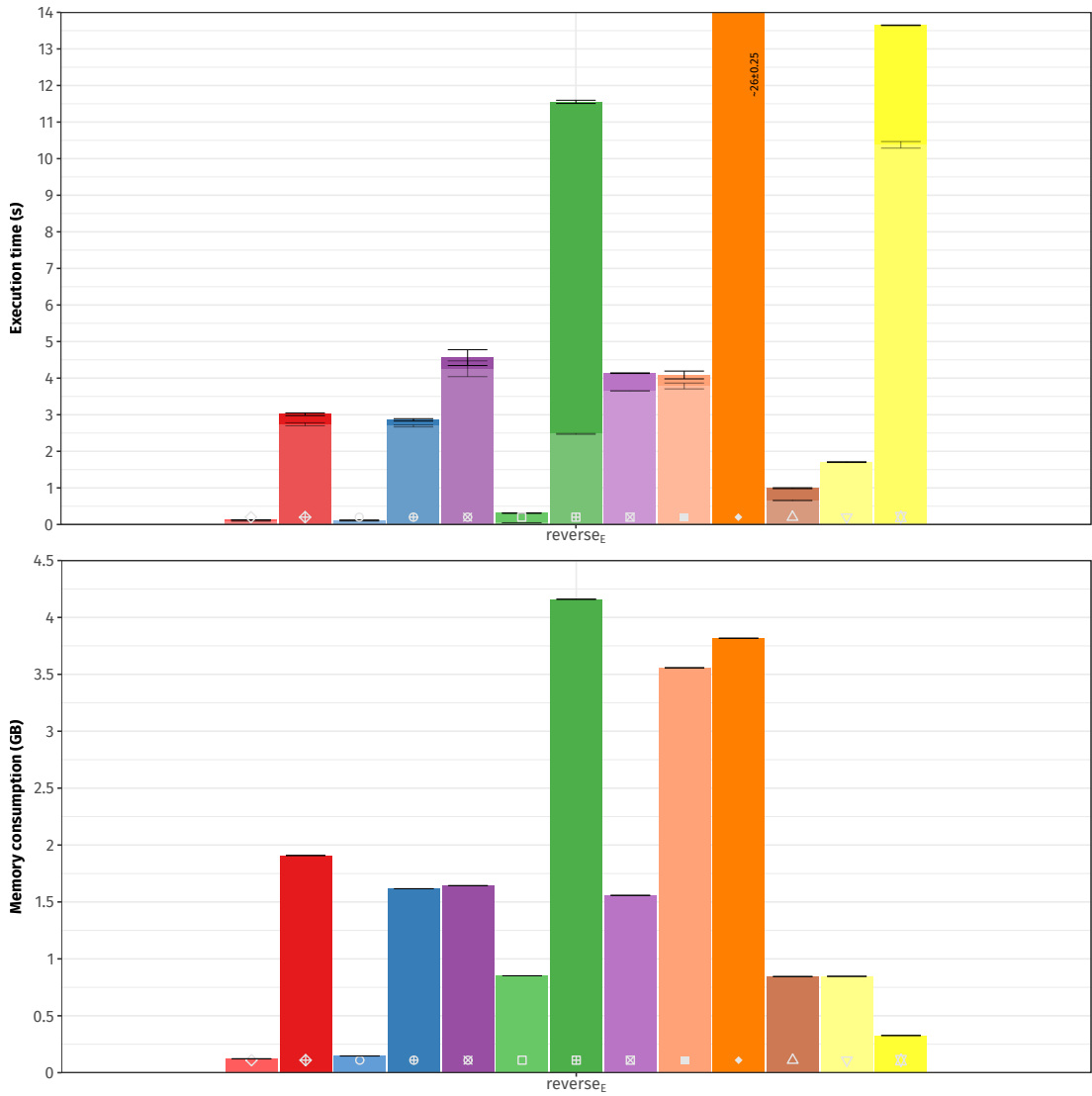


FIGURE 52: Results for benchmark reverse, niladic elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

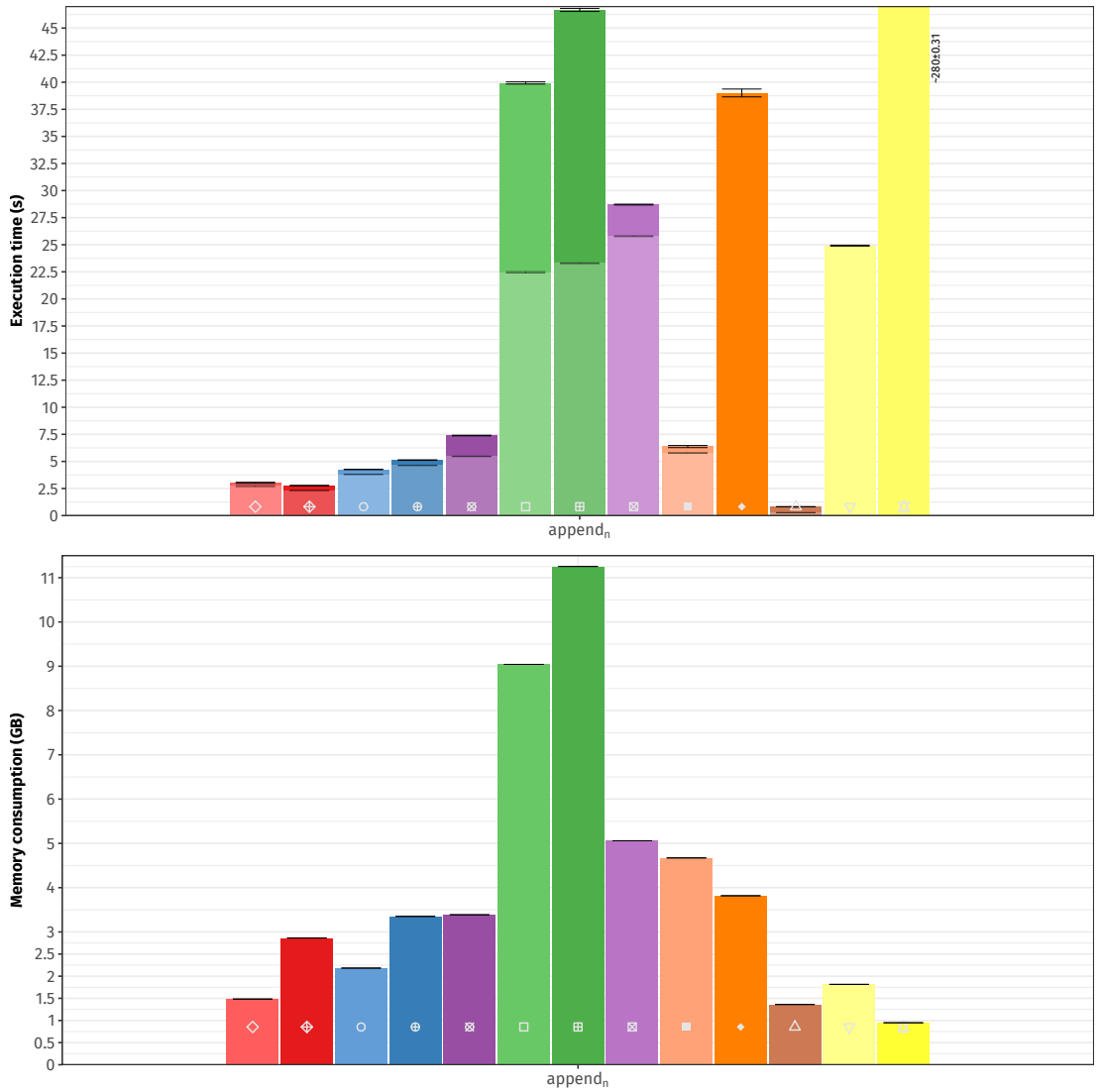


FIGURE 53: Results for benchmark `append`, numeric elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

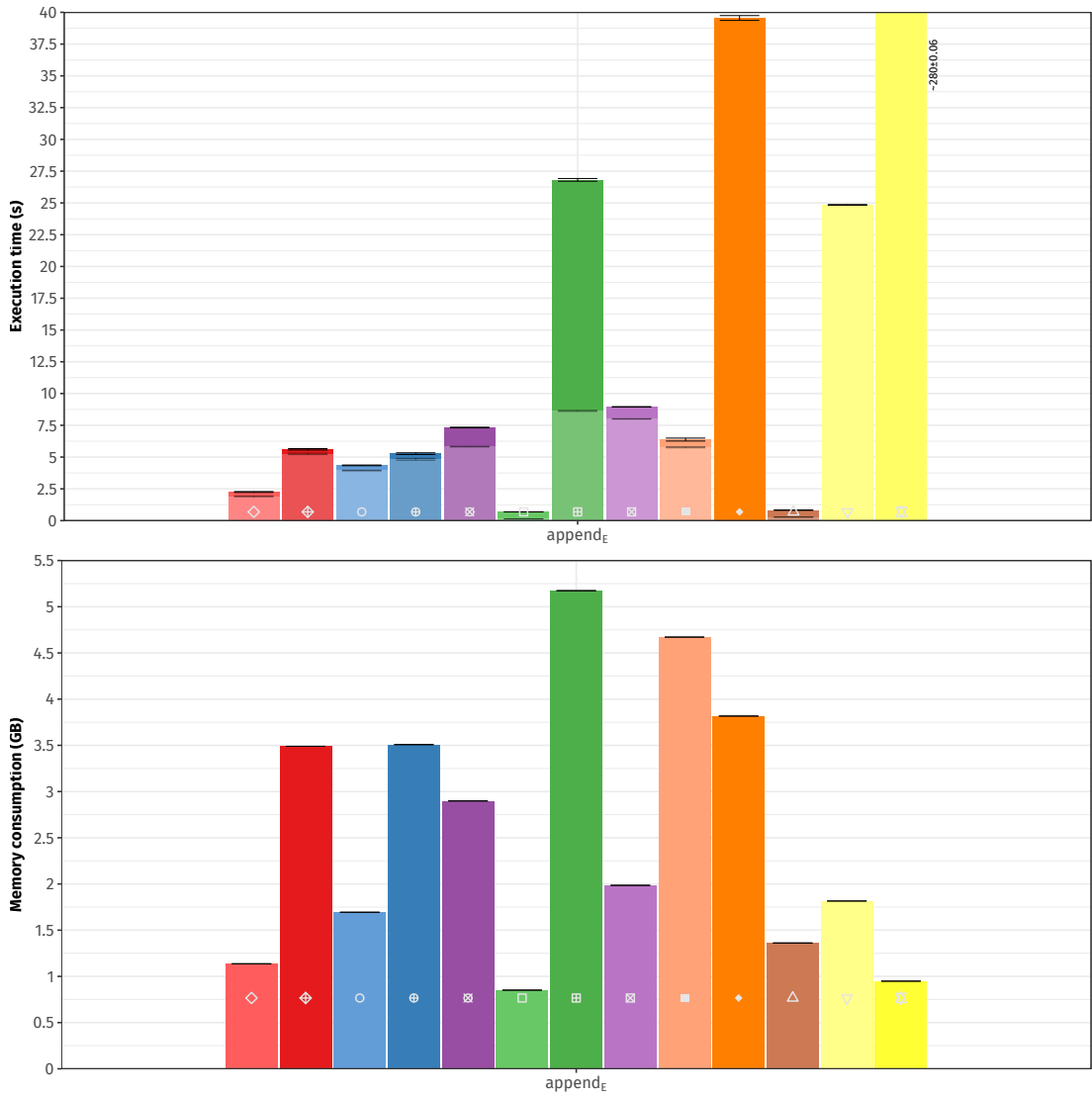


FIGURE 54: Results for benchmark append, niladic elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

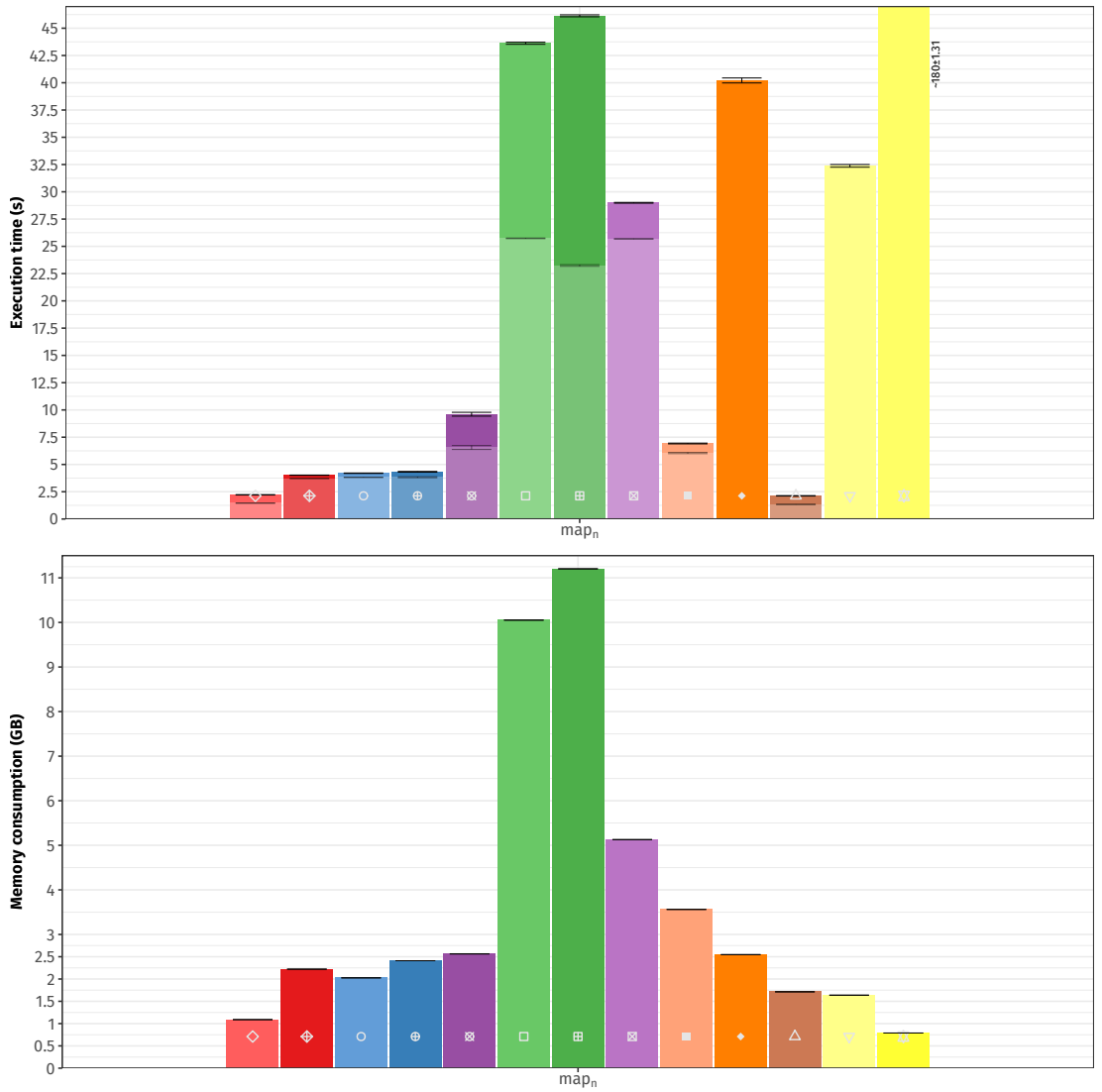


FIGURE 55: Results for benchmark map, numeric elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

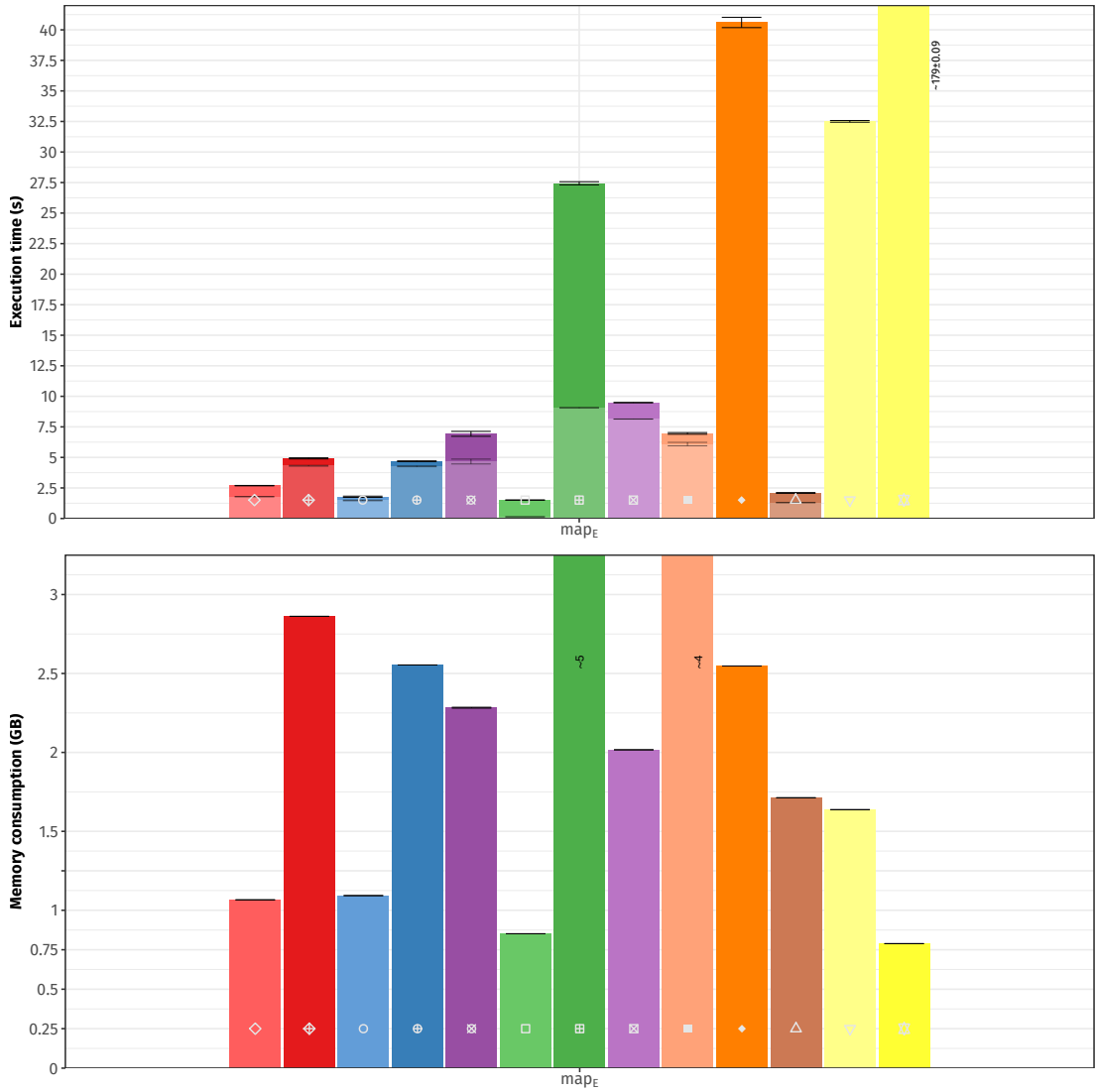


FIGURE 56: Results for benchmark map, niladic elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

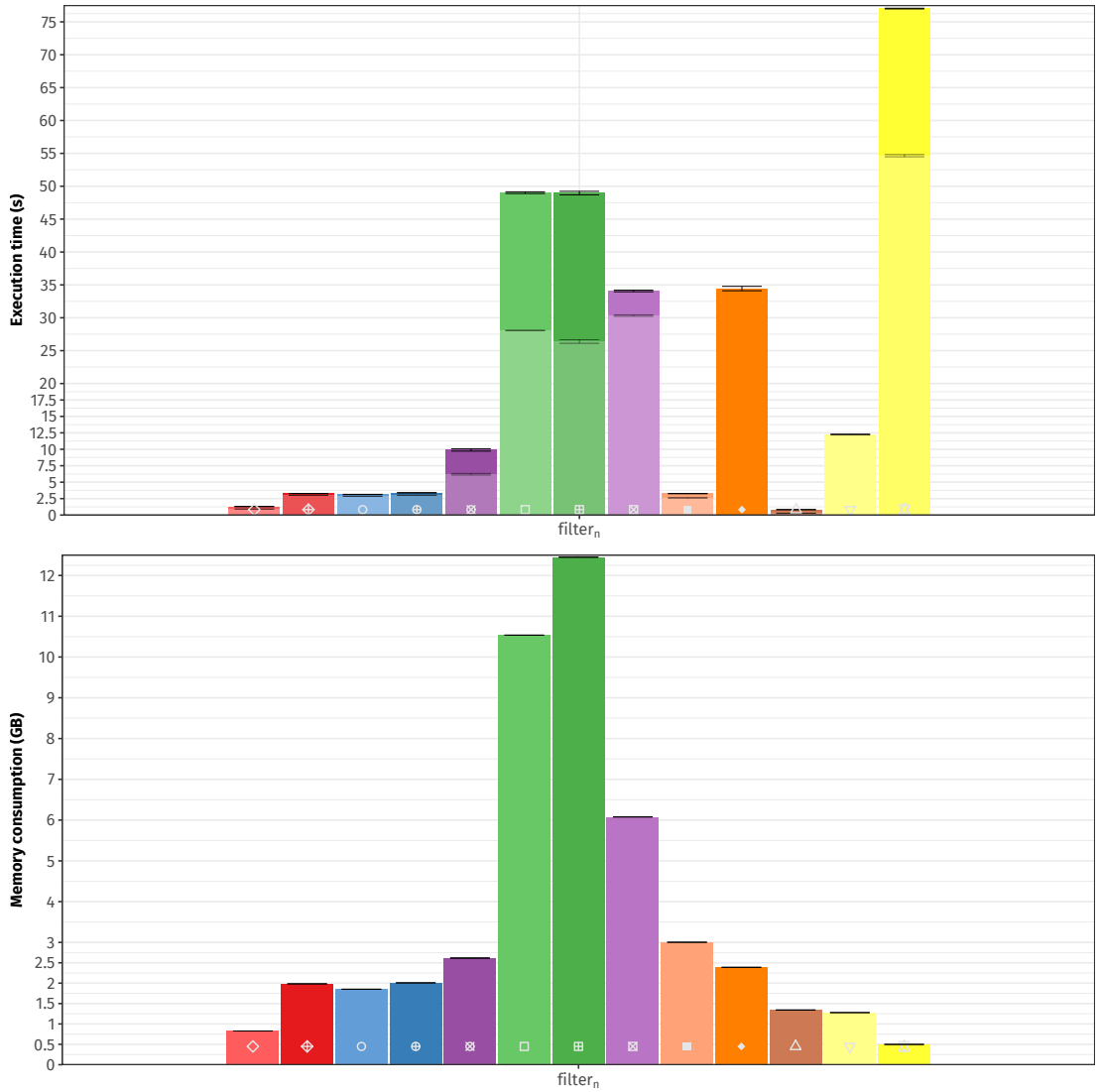


FIGURE 57: Results for benchmark filter, numeric elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

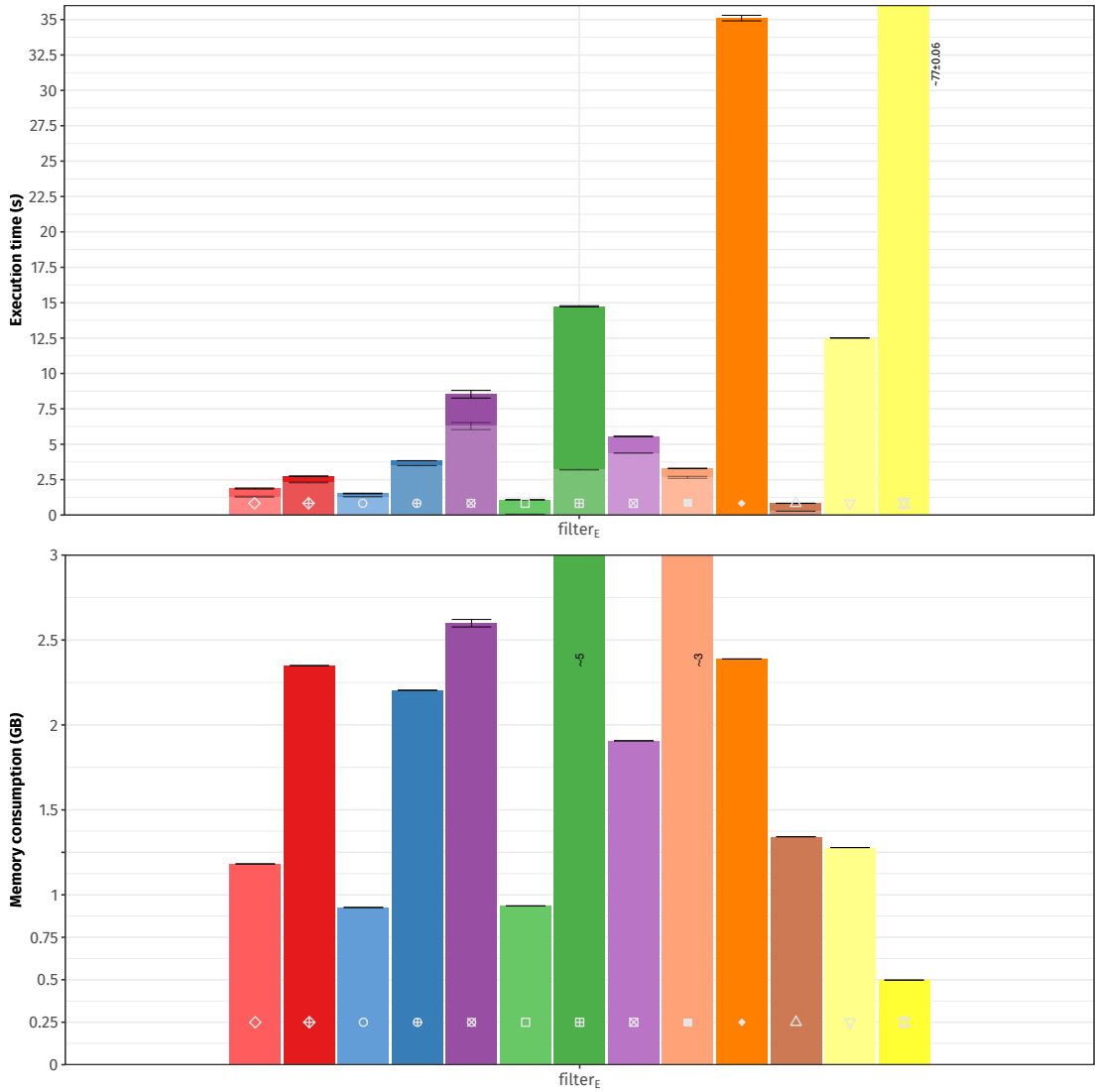


FIGURE 58: Results for benchmark filter, niladic elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

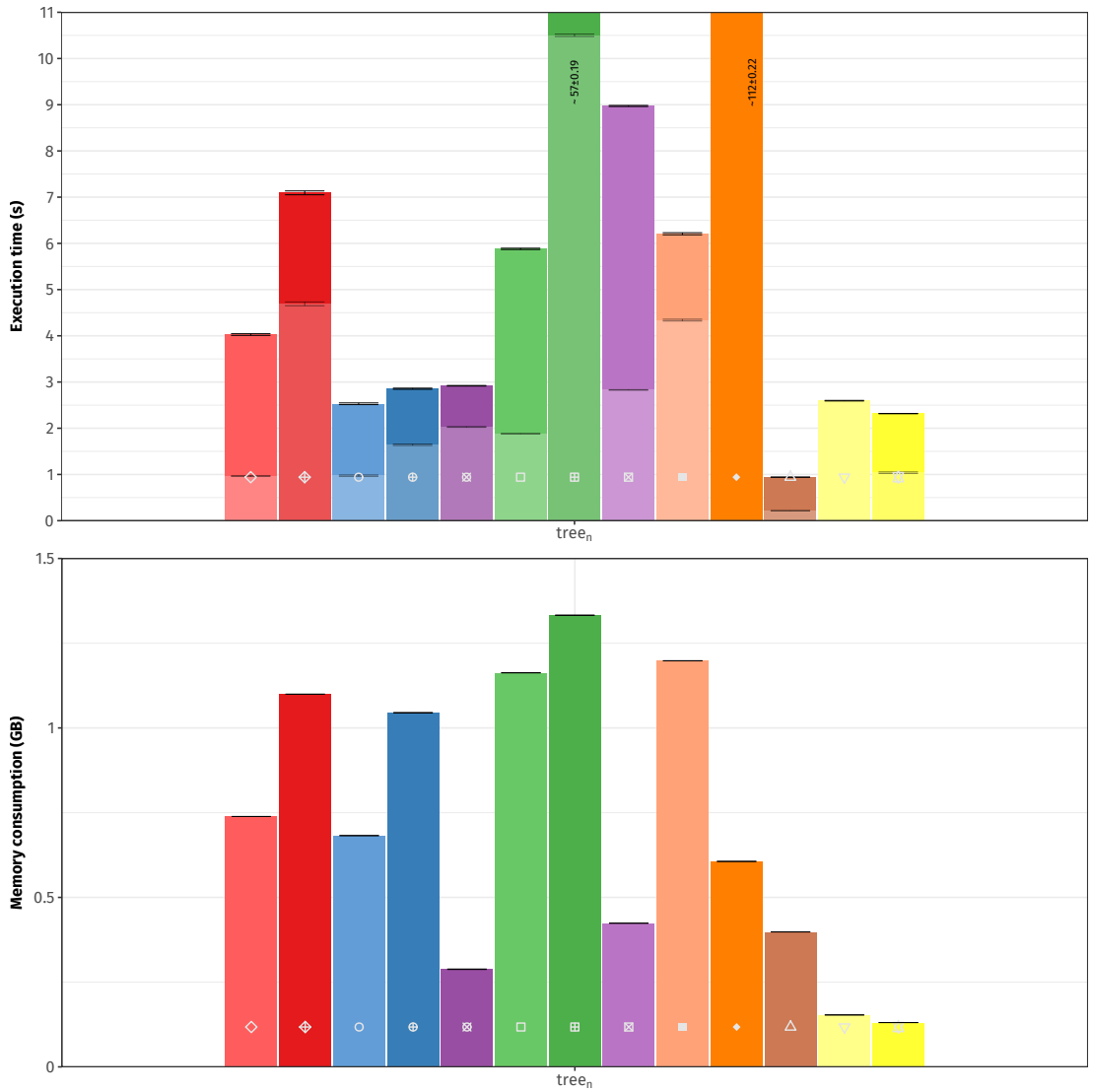


FIGURE 59: Results for benchmark tree, numeric elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

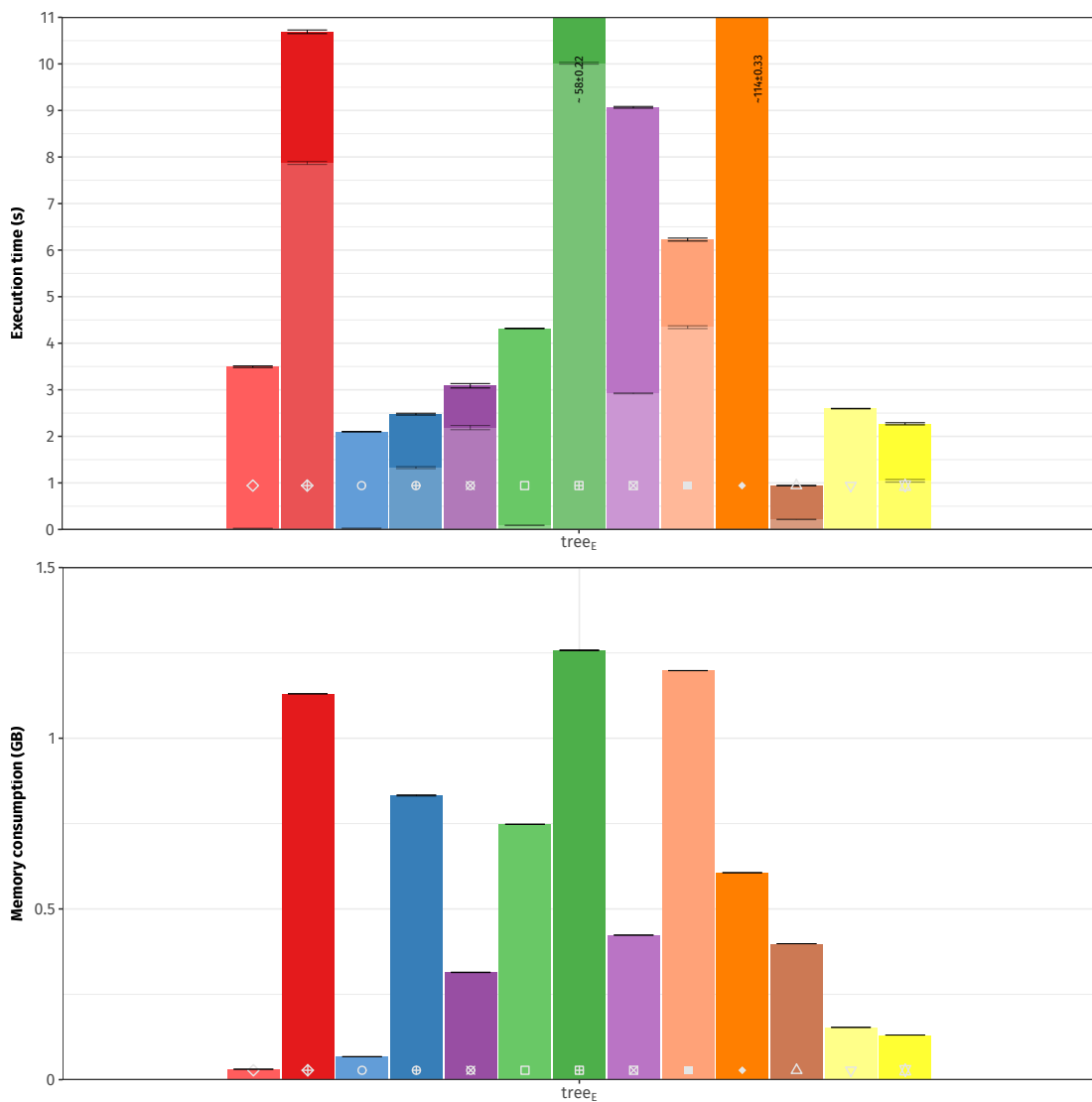








FIGURE 60: Results for benchmark tree, niladic elements variant. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

TABLE 15: Share of execution time spent in garbage collection for all benchmarks. Execution time of garbage collection is given as fraction of overall execution. We include the bootstrapped [32] 95 % confidence interval. Note that not all implementations provide this measurement. Smaller numbers are neither better nor worse. Implementations are abbreviated using the icons used throughout this work (cf. page 141).

												
reverse _E	0.718	+0.004 -0.004	0.909	+0.014 -0.014	0.816	+0.016 -0.017	0.941	+0.011 -0.011	0.152	+0.004 -0.004	0.214	+0.001 -0.001
reverse _n	0.778	+0.005 -0.005	0.938	+0.007 -0.007	0.869	+0.017 -0.017	0.930	+0.019 -0.019	0.479	+0.003 -0.004	0.253	+0.002 -0.002
append _E	0.848	+0.016 -0.017	0.936	+0.012 -0.012	0.907	+0.007 -0.007	0.917	+0.015 -0.015	0.194	+0.003 -0.003	0.322	+0.002 -0.002
append _n	0.898	+0.018 -0.018	0.835	+0.005 -0.005	0.893	+0.002 -0.002	0.904	+0.005 -0.005	0.562	+0.002 -0.002	0.499	+0.001 -0.001
map _E	0.664	+0.006 -0.006	0.877	+0.008 -0.008	0.833	+0.023 -0.024	0.911	+0.008 -0.008	0.093	+0.001 -0.001	0.330	+0.002 -0.002
map _n	0.656	+0.006 -0.007	0.925	+0.004 -0.004	0.909	+0.005 -0.005	0.885	+0.011 -0.011	0.590	+0.001 -0.001	0.504	+0.002 -0.002
filter _E	0.691	+0.021 -0.021	0.838	+0.011 -0.011	0.853	+0.017 -0.016	0.910	+0.002 -0.002	0.059	+0.000 -0.000	0.217	+0.001 -0.001
filter _n	0.706	+0.006 -0.006	0.917	+0.003 -0.003	0.909	+0.005 -0.005	0.891	+0.007 -0.007	0.573	+0.001 -0.001	0.539	+0.005 -0.005
tree _E	0.006	+0.000 -0.000	0.736	+0.003 -0.003	0.012	+0.000 -0.000	0.536	+0.008 -0.008	0.021	+0.000 -0.000	0.173	+0.001 -0.001
tree _n	0.240	+0.002 -0.002	0.661	+0.006 -0.006	0.385	+0.005 -0.005	0.575	+0.005 -0.005	0.320	+0.001 -0.001	0.183	+0.001 -0.001

Appendix C

Source code listings

C.I GRAMMAR FOR THE LANGUAGE OF THESEUS

```
1 # Tokens
2 NEWLINE: "[\n\r]";
3 LAMBDA: "λ.";
4 ULAMBDA: "Λ.";
5 MU: "μ";
6 MAPSTO: "→";
7 RIGHTWARDS_DOUBLE_ARROW: "⇒";
8 DEFINEDAS: "≐";
9 LEFT_PARENTHESIS: "(";
10 RIGHT_PARENTHESIS: ")";
11 LEFT_DOUBLE_ANGLE: "«";
12 RIGHT_DOUBLE_ANGLE: "»";
13 INTEGER: "[+-]?[0-9]+";
14 FLOAT: "[+-]?[0-9]*\.[0-9]+";
15 QSTRING: "'[^']*'";
16 QQSTRING: "\"[^\"]*\"";
17 NAME: "[^0-9,\\(\)\{\}\ \n\r\t\f#][^,\\(\)\{\}\ \n\r\t\f#]*";
18 IGNORE: "[\f\t]*#[^\n]*";
19
20 # Productions
21 ship : [NEWLINE]* >toplevel_expressions<? [EOF]
22     ;
23
24 topLevel_expressions : topLevel_expression ([NEWLINE]+ >toplevel_expressions< )* [NEWLINE]*
25                       ;
26
27 topLevel_expression : <definition>
```

Source code listings

```
28         | <expression>
29         ;
30
31 expression : <constructor>
32             | <application>
33             | <lambda>
34             | <variable>
35             | <primitive>
36             | <primary>
37             ;
38
39 primary : <INTEGER>
40         | <FLOAT>
41         | <QSTRING> | <QQSTRING>
42         ;
43
44 variable : NAME
45         ;
46
47 definition : <lambda_definition>
48             | <lambda_forward>
49             | <value_definition>
50             ;
51
52 lambda_definition : NAME [DEFINEDAS] lambda
53                 ;
54 lambda_forward : NAME [DEFINEDAS] [ULAMBDA]
55               ;
56 value_definition : NAME [DEFINEDAS] expression
57                 ;
58
59 constructor : NAME constructor_args
60             ;
61 constructor_args : [LEFT_PARENTHESIS] >arglist< [RIGHT_PARENTHESIS]
62                 | [LEFT_PARENTHESIS] [RIGHT_PARENTHESIS]
63                 ;
64
65 arglist : expression ([","] [NEWLINE]? expression)*
66         ;
67
68 application : [MU] [LEFT_PARENTHESIS] [NEWLINE]? expression application_args [RIGHT_PARENTHESIS]
```

```
69     | [MU] [LEFT_PARENTHESIS] [NEWLINE]? expression [RIGHT_PARENTHESIS]
70     ;
71
72 application_args : ([","] [NEWLINE]? expression )+
73                 ;
74
75 lambda : [LAMBDA] [NEWLINE]? >lambda_content<;
76
77 lambda_content : >rules<
78                 | rule
79                 ;
80
81 rules : [INTEGER] ["."] rule ([NEWLINE]+ >rules<)?
82        ;
83
84 rule : patterns? [MAPSTO] >body<
85       ;
86
87 body : continuation
88       | expression
89       ;
90
91 continuation : expression [RIGHTWARDS_DOUBLE_ARROW] [NEWLINE]? lambda_content
92              ;
93
94 patterns : pattern ([","] [NEWLINE]? pattern)*
95          ;
96
97 pattern : <constructor_pattern>
98          | <variable_pattern>
99          | <primary_pattern>
100         ;
101
102 variable_pattern : >variable<
103                 ;
104
105 primary_pattern : primary
106                 ;
107
108 constructor_pattern : NAME constructor_pattern_args
109                    ;
```

```
110 constructor_pattern_args : [LEFT_PARENTHESIS] >pattern_arglist< [RIGHT_PARENTHESIS]
111 | [LEFT_PARENTHESIS] [RIGHT_PARENTHESIS]
112 ;
113
114 pattern_arglist : pattern ([","] [NEWLINE]? pattern)*
115 ;
116
117 primitive : [LEFT_DOUBLE_ANGLE] NAME [RIGHT_DOUBLE_ANGLE]
118 ;
119 # EOF
```

C.2 FULL EXAMPLE OF A THESEUS PROGRAM

```
1 #!/usr/bin/env theseus
2
3 p := <<print_result_string>>
4 clock := <<clock>>
5 gctime := <<gctime>>
6
7 # type Nil, Cons(,)
8 cons := λ. A, B → Cons(A, B)
9 head := λ. Cons(A, B) → A
10 tail := λ. Cons(A, B) → B
11
12 make_list$aux := λ.
13     1. acc, 0 → acc
14     2. acc, n → μ(make_list$aux,
15                   μ(cons, E(), acc),
16                   μ(<<minus_int>>, n, 1))
17 make_list := λ. n → μ(make_list$aux, Nil(), n)
18
19
20 time$cont6 := λ. res, _ → res
21 time$cont5 := λ. res, diff, gcdiff → μ(time$cont6,
22                                         res,
```

```

23         μ(p, diff, diff, gcdiff))
24 time$cont4 := λ. res, gc1, t1, t2, gc2 → μ(time$cont5,
25         res,
26         μ(⟨⟨minus_float⟩⟩, t2, t1),
27         μ(⟨⟨minus_float⟩⟩, gc2, gc1))
28 time$cont3 := λ. res, gc1, t1, t2 → μ(time$cont4,
29         res,
30         gc1, t1, t2, μ(gctime))
31 time$cont2 := λ. res, gc1, t1 → μ(time$cont3, res, gc1, t1, μ(clock))
32 time$cont1 := λ. fun, arg, gc1, t1 → μ(time$cont2, μ(fun, arg), gc1, t1)
33 time$cont0 := λ. fun, arg, gc1 → μ(time$cont1, fun, arg, gc1, μ(clock))
34 time      := λ. fun, arg → μ(time$cont0, fun, arg, μ(gctime))
35
36
37 reverse$aux := λ.
38     1. acc, Nil() → acc
39     2. acc, Cons(h, t) → μ(reverse$aux, Cons(h, acc), t)
40 reverse := λ. l → μ(reverse$aux, Nil(), l)
41
42 num := μ(λ.
43     1. nil() → 20000000
44     2. cons(h, _) → μ(⟨⟨strtol⟩⟩, h), arguments)
45 l := μ(make_list, num)
46 μ(time, theseus_reverse, l)

```

*Source code
of the
benchmarks*

C.3 SOURCE CODE OF THE BENCHMARKS

For the comparative micro-benchmarks, we provide the essential part for every benchmark in each of the three compared implementations. We do not give individual listings for the two variants (list containing numeric elements, list containing niladic elements) as they do not change the code under benchmark but rather data processed. We do however provide basic information about how we achieved both variants.

C.3.1 Reverse

Theseus

```
1 reverse$aux := λ.  
2     1. acc, Nil()      ↦ acc  
3     2. acc, Cons(h, t) ↦ μ(reverse$aux, Cons(h, acc), t)  
4 reverse := λ. l ↦ μ(reverse$aux, Nil(), l)
```

*Source code
listings*

Pycket/Racket

```
1 (letrec  
2   ([head car]  
3    [tail cdr]  
4    [racket-reverse (lambda (l)  
5                      (letrec ((aux (lambda (list acc)  
6                                  (if (null? list)  
7                                      acc  
8                                      (aux (tail list) (cons (head list) acc))))))  
9                      (aux l '()))])  
10 ; ...  
11 )
```

RSqueak/Squeak

```
1 _____ VCons _____  
2 reversed  
3  
4 | list cons |  
5 list := VNil nil.  
6 cons := self.  
7 [cons isEmpty] whileFalse:  
8   [cons isCons ifFalse: [^ self error: 'Not a proper list'.  
9   list := self class car: cons car cdr: list.  
10  cons := cons cdr].  
11 ^ list
```

C.3.2 Append

Theseus

```
1 append := λ.  
2     1. Nil(), B      → B  
3     2. Cons(h, t), B → Cons(h, μ(append, t, B))
```

*Source code
of the
benchmarks*

Pycket/Racket

```
1 (letrec  
2   ([head car]  
3    [tail cdr]  
4    [racket-append (lambda (a b)  
5                    (if (null? a)  
6                        b  
7                        (cons (head a) (racket-append (tail a) b))))])  
8   ; ...  
9 )
```

RSqueak/Squeak

```
1 _____ VCons _____  
2 , aCons  
3 | cons acc |  
4 cons := self reversed. "reverse list to cons on aCons"  
5 acc := aCons.  
6 [cons isEmpty] whileFalse:  
7   [cons isCons ifFalse: [^ self error: 'Not a proper list'.  
8   acc := self class car: cons car cdr: acc.  
9   cons := cons cdr].  
10 ^ acc
```

C.3.3 Map

Theseus

```

1 map := λ.
2   1. fun, Nil()      → Nil()
3   2. fun, Cons(A, B) → Cons(μ(fun, A), μ(map, fun, B))

```

Source code listings

Pycket/Racket

```

1 (letrec
2   ([head car]
3    [tail cdr]
4    [racket-map (lambda (f l)
5                  (if (null? l)
6                      '()
7                      (cons (f (head l)) (racket-map f (tail l)))))]
8   ; ...
9  )

```

RSqueak/Squeak

```

1 _____ VCons _____
2 collect: aBlock
3
4   | cons acc |
5   cons := self.
6   acc := VNil nil.
7   [cons isEmpty] whileFalse:
8     [cons isCons ifFalse: [^ self error: 'Not a proper list'].
9     acc := self class car: (aBlock value: cons car) cdr: acc.
10    cons := cons cdr].
11   ^ acc reversed

```

C.3.4 Filter

Theseus

```

1 f := λ.
2   1. _, Nil(), _ → Nil()

```



```

3 2. pred, Cons(h,t), True() → Cons(h, μ(f, pred, t, Nil()))
4 3. pred, Cons(h,t), False() → μ(f, pred, t, Nil())
5 4. pred, Cons(h,t), _      → μ(f, pred, Cons(h, t), μ(pred, h))
6 filter := λ. pred, lst → μ(f, pred, lst, Nil())

```

Pycket/Racket

```

1 (letrec
2   ([head car]
3    [tail cdr]
4    [racket-filter (lambda (p l)
5                     (cond [(null? l) '()]
6                             [(p (head l)) (cons (head l) (racket-filter p (tail l)))]
7                             [else (racket-filter p (tail l))])])
8   ; ...
9 )

```

*Source code
of the
benchmarks*

RSqueak/Squeak

```

1 _____ VCons _____
2 select: aBlock
3
4 | cons acc |
5 cons := self.
6 acc := VNil nil.
7 [cons isEmpty] whileFalse: [
8   cons isEmpty iffFalse: [^ self error: 'Not a proper list'].
9   (aBlock value: cons car)
10   ifTrue: [acc := self class car: cons car cdr: acc].
11   cons := cons cdr].
12 ^ acc reversed

```

C.3.5 Tree

Theseus

Source code listings

```
1 kMinTreeDepth := 3
2
3 MakeTree := λ.
4     1. 0      → Leaf(E())
5     2. iDepth → Node(μ(MakeTree, μ(-, iDepth, 1)),
6                       E(),
7                       μ(MakeTree, μ(-, iDepth, 1)))
8
9
10 iter$0 := λ.
11     1. _, _, _, _, _, 0 → nil()
12     2. i, niter, d, _, _, _ → μ(iter$0, μ(+, 1, i), niter, d,
13                                     μ(MakeTree, d), μ(MakeTree, d),
14                                     μ(-, niter, i))
15 iter := λ. i, niter, d → μ(iter$0, i, niter, d, nil(), nil(), niter)
16
17 niter := λ. max_depth, d → μ(<<, 1, μ(-, max_depth, d))
18
19 loop_depths$0 := λ.
20     1. _, _, _, _, 0 → nil()
21     2. d, max_depth, nil(), _, _ → μ(loop_depths$0,
22                                     d, max_depth, μ(niter, max_depth, d),
23                                     nil(), nil())
24     3. d, max_depth, numIter, _, _ → μ(loop_depths$0,
25                                     μ(+, 1, d), max_depth, μ(niter, max_depth, d),
26                                     μ(iter, 1, numIter, d), μ(-, max_depth, d))
27
28 loop_depths := λ. d, max_depth → μ(loop_depths$0, d, max_depth, nil(), nil(), nil())
29
30 check$0 := λ.
31     1. Leaf(X),      nil() → E()
32     2. Node(l, X, r), nil() → μ(check$0, μ(check$0, l, nil()), r)
33     3. _,          r → μ(check$0, r, nil())
34 check := λ. X → μ(check$0, X, nil())
35
36 tree$cont3 := λ. _, longlived → μ(check, longlived)
37 tree$cont2 := λ. max_depth, longlived → μ(tree$cont3,
38                                     μ(loop_depths, kMinTreeDepth, max_depth), longlived)
39 tree$cont1 := λ. max_depth, _ → μ(tree$cont2, max_depth, μ(MakeTree, max_depth))
```

```

40 tree$cont0 := λ. stretch, max_depth → μ(tree$cont1, max_depth, μ(MakeTree, stretch))
41
42 tree := λ. num → μ(tree$cont0, μ(+, 1, num), num)

```

Pycket/Racket

```

1 (define (make item d)
2   (if (= d 0)
3     (leaf item)
4     (let ((d2 (- d 1)))
5       (node (make item d2) item (make item d2)))))
6
7 (define (check t)
8   (if (leaf? t)
9     e
10    (begin
11      (check (node-left t))
12      (check (node-right t)))))
13
14 (define min-depth 3)
15
16 (letrec
17   ([racket-tree
18    (lambda (num)
19      (letrec ((max-depth num)
20              (stretch-depth (+ max-depth 1))
21              (_ (make e stretch-depth))
22              (long-lived-tree (make e max-depth))
23              (depth-loop (lambda (d)
24                            (letrec ([iterations (expt 2 (- max-depth d))]
25                                    [iter (lambda (i)
26                                            (when (<= i iterations)
27                                              (begin
28                                                (make e d)
29                                                (make e d)
30                                                (iter (+ 1 i)))]))]
31                            (iter 1)
32                            (when (< d max-depth)
33                              (depth-loop (+ 1 d)))))))

```

*Source code
of the
benchmarks*

```

34         (depth-loop min-depth)
35         (check long-lived-tree))))))
36     ; ...
37 )

```

Source code
listings

RSqueak/Squeak

```

1 Object subclass: #VNode
2   instanceVariableNames: 'left val right'
3   _____
4   VNode class
5   _____
6   treeWithMin: aNumber max: anotherNumber element: item
7
8   | maxDepth stretchDepth ignore longLived minDepth |
9   minDepth := aNumber.
10  maxDepth := anotherNumber.
11  stretchDepth := anotherNumber + 1.
12  ignore := self make: item through: stretchDepth.
13  longLived := self make: item through: maxDepth.
14
15  minDepth to: maxDepth do:
16    [:d | | iterations |
17      iterations := 2 raisedTo: (maxDepth - d).
18      iterations timesRepeat:
19        [ | ignore1 ignore2 |
20          ignore1 := self make: item through: d.
21          ignore2 := self make: item through: d]].
22      ^ longLived check
23
24  _____
25  make: anObject through: aNumber
26
27  ^ aNumber <= 0
28    ifTrue: [VLeaf val: anObject]
29    ifFalse:
30      [ | down |
31        down := aNumber - 1.
32        self
33          left: (self make: anObject through: down)
34          val: anObject
35          right: (self make: anObject through: down)]

```

```

33 _____
34
35 Object subclass: #VLeaf
36   instanceVariableNames: 'val'
37   _____ VNode _____
38 check
39
40   self left check.
41   ^ self right check.
42
43   _____ VLeaf _____
44 check
45   ^ self val

```

*Source code
of the
benchmarks*

C.3.6 Variants

Theseus For reverse and append, the list-preparing function contains in the numeric elements variant $\mu(\text{cons}, 17, \text{acc})$, and in the niladic elements variant $\mu(\text{cons}, E(), \text{acc})$.

For filter, it additionally contains the filter predicate; for numeric elements:

```

1 flt := λ.
2   1. 17 ↦ True()
3   2. _ ↦ False()

```

and for niladic elements:

```

1 flt := λ.
2   1. E() ↦ True()
3   2. _ ↦ False()

```

Similarly, the mapped function in map; for numeric elements:

```

1 swap := λ.
2   1. 17 ↦ 36
3   2. 36 ↦ 17

```

and for niladic elements:

```
1 swap := λ.  
2   1. E() ↦ F()  
3   2. F() ↦ E()
```

*Source code
listings*

The tree benchmark above is given in the niladic elements variant. For the numeric elements variant, replace E() with 17.

Pycket/Racket For the list-based benchmark, in the numeric elements variant, the following definitions are used:

```
1 (letrec  
2   ([e 17]  
3    [f 36]  
4    ; filtering function for filter  
5    [flt (lambda (x) (= x e))]  
6    ; function mapped over in map  
7    [swap (lambda (x) (if (= x e) f e))]  
8    ; ...  
9  )
```

and for the niladic elements variant:

```
1 (struct E () #:transparent)  
2 (struct F () #:transparent)  
3  
4 (letrec  
5   ([e (E)]  
6    [f (F)]  
7    ; filtering function for filter  
8    [flt E?]; <=> (lambda (x) (E? x))  
9    ; function mapped over in map  
10   [swap (lambda (x) (if (E? x) f e))]  
11   ; ...  
12  )
```

For the tree benchmark with numeric elements, the following definitions are used:

```

1 (struct node (left val right))
2 (struct leaf (val))
3
4 (define e 17)

```

and with niladic elements:

```

1 (struct node (left val right) #:transparent)
2 (struct leaf (val) #:transparent)
3 (struct E () #:transparent)
4 (define e (E))

```

*Source code
of the
benchmarks*

RSqueak/Squeak For the list-based benchmark, in the numeric elements variant, the following definitions are used:

```

1 e := 17.
2 f := 36.

```

and for the niladic elements variant:

```

1 e := VVector new.
2 f := VNil nil.

```

Common to both variants, these objects are used for generating the lists and the filtering/mapping functions.

```

1 lst1 := VCons withAll: (Array new: self withAll: e).
2 " for map/filter "
3 lst2 := VCons withAll: (VCons alternatingArrayOfSize: self with: e andAlternate: f).
4 flt := [:element | element = e].
5 swap := [:element | element = e ifTrue: [f] ifFalse: [e]].

```

Note that = is used for comparison, which, in the numeric elements case, maps to ==, that is, identity; but in the niladic elements case, uses our custom implementation that respects the semantics of values:

```

1 _____ VVector _____
2 = otherCollection
3

```

```
4 " value semantics "  
5 ^ otherCollection class == self class  
6 and: [otherCollection size == self size  
7 and: [self size = 0  
8 or: [self hasEqualElements: otherCollection]]]
```

```
1 _____ VNil _____  
2 = other  
3  
4 " value semantics "  
5 ^ other class == self class
```

*Source code
listings*

Appendix D

Immutable Boolean Field Elision

Extract

The following is an excerpt from “Record Data Structures in Racket: Usage Analysis and Optimization” as appeared in *ACM SIGAPP Applied Computing Review*, December 2016, Vol. 16, No. 4 [94], that introduces the concept of *immutable boolean field elision* as used in the Racket-implementation Pycket.

I INTRODUCTION

For programming language implementations, performance is often key and, among other aspects, built-in data structures contribute to the overall performance of a language implementation. The lack of optimization of built-in data structures may result in poor performance and increased memory consumption of dynamic languages [2, 18]. In the context of modern VM development frameworks, such as RPython, some data structures, such as collections [5], are already in the focus of research.

Record data structures or *records* are one of the advanced common built-in data structures, which are not deeply investigated in the sense of optimizations for modern VMs. Basically, records aggregate heterogeneously typed, named fields, possibly with a definition in a record type. In some languages, such as Racket, records may not only be used to store the data, but have additional features. Racket is a dynamic multi-paradigm Scheme-family programming language with powerful built-in record data structures, where records can

behave like objects of a class or even like a function. Records also often provide identity, encapsulation, abstraction, and maybe behavior, thus providing key ingredients for object orientation. In fact, records can be used to implement object-oriented features, such as the class-based object orientation in Racket [13].

A simple, straight-forward implementation of records for dynamically typed languages implies a big overhead because of the semantics complexity. It is more important for the implementation to be simple than the interface. Thus, many languages prefer the “worse-is-better” approach [14], whereby the simplicity and efficiency of implementation are more important than the straight-forward following the semantics and perfect correctness. Our analysis shows that, at least for the Racket language, records have a noticeable optimization potential. In this work, we consider an efficient implementation of records for dynamic languages and for Racket in particular. We focus on the RPython-based implementation named Pycket.

In this work, we make the following contributions:

- We analyse and evaluate the usage of record data structures in Racket applications (section 3).
- We identify applicable optimization techniques for the efficient implementation of record data structures (section 4). In particular, *we propose a novel optimization technique for static immutable boolean fields in record data structures* (section 4.3).
- We implement Racket’s record data structures with optimizations and evaluate performance results (section 5 and 6).

2 BACKGROUND

[omitted from excerpt]

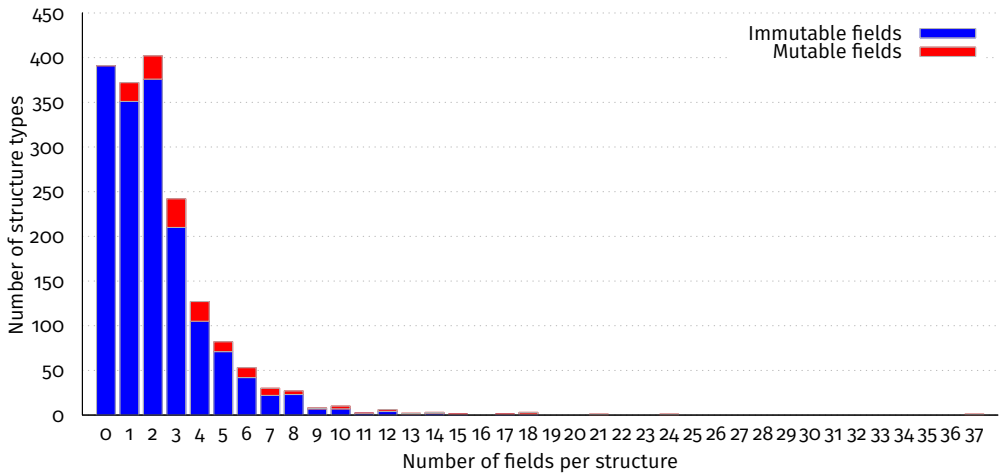


FIGURE A: Distribution of number of structure fields in the Racket standard library.

3 STRUCTURE USAGE IN RACKET

Racket structures are a powerful data structure with broad applicability. They are widely used in Racket packages¹ and projects on GitHub². Structures are essential for the Racket contracts implementation. In this section, we investigate how structures are actually used in different Racket applications. We perform a static and dynamic analysis of existing applications to identify the typical size of structures, types used within structures and the frequency of mutation.

We choose five Racket applications from different domains including development tools, text analysis, mathematics, and games. *I Write Like*³ — one of the biggest Racket applications — is a web application that analyses the style of a given text by comparing with styles of many famous writers. This

¹<http://pkgs.racket-lang.org> (visited 2015-12-05)

²<https://github.com/search?q=language%3Aracket> (v. 2015-12-05)

³<https://github.com/coding-robots/iwl> (visited 2015-12-05)

TABLE A: Results of the static analysis of Racket standard library source code files.

Structure Type	1765	100 %
With super-types	563	31.9 %
With mutable fields	148	8.4 %
Transparent	659	37.3 %
Prefabs	146	8.2 %

Immutable
Boolean
Field Elision

application represents a heavy text analysis application. The *markdown* parser application⁴ is a simple parser for *markdown* formatted text that is used in many other Racket projects as a library. *Racket CAS*⁵ is a simple computer algebra system for Racket with a good built-in test set. *2048*⁶ is a Racket implementation of a famous puzzle-game with numbers. Finally, *DrRacket* is a feature-rich Racket *integrated development environment (IDE)*, which is widely used by Racket-programmers.

3.1 Static Analysis

We perform a static source code analysis of the Racket v6.2.0.4 standard library comprising 4 812 Racket source code files. We track the number of immutable and mutable fields and super types per structure.

3.1.1 Results

Of all the source files, 11.6 % contain all 1765 structure type definitions (cf. table A), 31.9 % with super-types. Structures have 2.3 ± 2.6 fields on average, with a median of 2. The largest structure from the Racket library has 37 fields. 91.6 % of all structure types are immutable. Structures with mutable fields tend to be larger (maximum: 37, mean: 4.55 ± 4.56) than all-immutable

⁴<https://github.com/greghendershott/markdown> (visited 2015-12-05)

⁵<https://github.com/soegaard/racket-cas> (visited 2015-12-05)

⁶<https://github.com/danprager/racket-2048> (visited 2015-12-05)

structures (maximum: 18, mean: 2.10 ± 2.17). The distribution is shown in figure A.

The statically determined number of structure types in the applications analyzed is comparatively small; together, they define 22 structure types with at most 5 fields (average 1.64 ± 1.26 , median 1), all immutable. We refrain from plotting the distribution.

3.2 Dynamic Analysis

We instrumented the structure implementation in Racket to track the creation process of structure types, structure instances, the amount and types of structure field values, and the frequency of mutate operations. Our analysis reports the total usage of structures including the Racket core.

*Structure
Usage in
Racket*

3.2.1 Results

Refining the static analysis, about 85 % of all fields used are immutable, with *DrRacket* being an outlier with about 61 % of immutable fields. Structure instances have 1.62 fields on average with a median of 1. The number of instances of each structure type depends heavily on the specific application, ranging from 200 to 1500 in our tests. The number of mutations varies even more.

Although structures in Racket are typically used monomorphic, i.e. the data type of values stored in a field does not change, some instances' fields are used with values of more than one data type (*non-monomorphic*). The amount of structures containing at least one non-monomorphic field is between 5 % and 15 %.

The distribution of field types is homogeneous as illustrated in figure B. The most common data type used in structure field type is *boolean*. Up to 70 % of booleans have the value *#f* (false), which is used in up to 88 % as a placeholder default value for other data types, such as *procedure*. *Procedures* are also used widely, to the extent that some structures only contain exactly one procedure— such procedure-containers are often used as super-types for other structures. *Strings*, mutable and immutable, pose the most user-faced data type in field types while *symbols* and the *syntax* type (used by the Racket macro

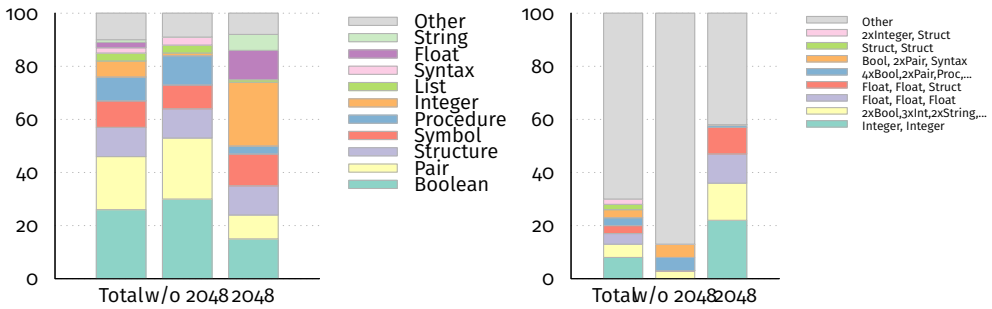


FIGURE B: Most frequent field types (left) and most frequent combinations of field types (right) in Racket applications

system) are more system-faced, or even meta-level types used in structures. Non-scalar field types, such as *pairs* and *lists*, and other *structures* are common as field types, too. Other types have a collective share of about 10 %.

Despite our initial assumption, *integers* are not very common, except for the *2048* game that heavily uses *integers* and *floats*. Other applications use numbers significantly less frequently. To show this, we separated *2048* in figure B.

We found only few common data type collocation patterns in structures, despite the homogeneous field type distribution. Such patterns include the use of *integer*, *integer*-structures in *2048* for coordinates, the most prevalent collocation in this application. This is, nevertheless, uncommon for other applications. Thus, combinations of stored together field types in structures are mostly application specific. No patterns can be derived in the general case as less than 30 % of all structures exhibit significant similarity. The right part of figure B shows this in more detail.

3.3 Discussion of Analysis Results

We found that Racket structures are relatively small and contain between one or two fields on average. Furthermore, about 85 % of structure fields are immutable. Initially unexpected, *booleans* are the most common data type in

structures. We found that `#f` (Racket's *false* value) is used a placeholder default value and that the corresponding filled value is often a procedure.

4 OPTIMIZING RECORDS

Based on the analysis in section 3, we propose fitting optimizations to use to improve performance when compared with a simple, straight-forward direct-mapping approach. We think that this catalogue of optimizations can be worthwhile beyond Racket, given the usage of record data structures is not completely dissimilar. In particular, we suggest applying four standard optimizations and propose a new one, **immutable boolean field elision (IBFE)**.
[omitted from excerpt]

*Optimizing
Records*

4.1 Direct Mapping Approach

[omitted from excerpt]

4.2 General Optimizations

[omitted from excerpt]

4.2.1 Flat Structure

[omitted from excerpt]

4.2.2 Inlining

[omitted from excerpt]

4.3 Immutable Boolean Field Elision

Booleans are the most frequent field type in Racket structures. However, up to 70 % of boolean fields have the value `#f`. Knowing that most (up to 85 %) fields are actually immutable, a high number of fields in Racket structures hence consist of **immutable boolean fields (IBFs)**.

*Immutable
Boolean
Field Elision*

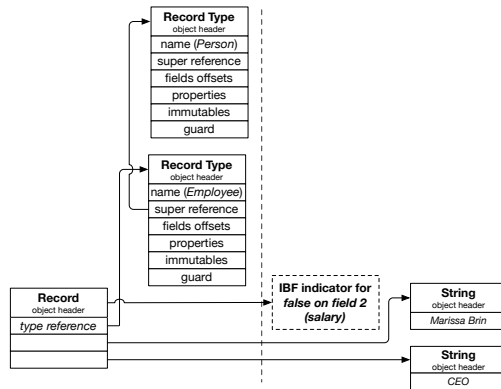


FIGURE C: employee structure with an **IBF** indicator denoting the elision of field 2

It seems feasible to actually *not* store this information as a field value per se. Instead of storing both positions and values of the boolean fields, we use an indicator to denote all positions of **IBFs** within a structure, effectively *eliding* the immutable *#f* values; we call this **immutable boolean field elision (IBFE)**. This indicator might be implementation specific; but in the same way structures that contain mutable fields or unboxed fields must be communicated to the runtime, **IBFs** can be communicated similarly, be it tagging, header bits, or class-based indication as in figure C, to name a few. It is crucial that all possible combinations of **IBFs** for an arbitrary record instance are present as indicators at structure allocation time. For example a record class with three fields, all immutable, that gets instantiated with an *#f* value on position two could use an implementation class that treats position two specially by not providing storage for it (cf. figure C). That implementation class would act as **IBF** indicator. Note that the *#t* value is not treated specially by **IBFE**, as are *#f* values in mutable fields. These are stored as if **IBFE** was not present at all.

The booleans optimization saves memory by reusing immutable *false* values. Assuming that structures have an average size of 2.3 fields, 26% of all fields

are booleans and 70% of booleans are *false*, and also that 85% of fields are immutable in Racket, for n structure objects, this saves

$$n \cdot 2.3 \cdot 0.26 \cdot 0.7 \cdot 0.85 - \text{sizeOfSpecializedClasses} \approx 0.36n \quad (4)$$

words in Racket on average, where *sizeOfSpecializedClasses* indicates the required memory for pre-defined structure classes with *false* fields. Although this optimization may have less positive impact on memory consumption on average, it does not add memory overhead for records in the worst case as unboxing with *field types* would. For extreme case, where every record has one immutable field with a value *false*, the saving would be approximately n .

Using **IBFE**, memory for immutable *#f* values can be saved at the expense of providing a large enough number of **IBF** indicators, which poses a trade-off. Applications with only few **IBFs** and large structures would be hit by the overhead of maintaining **IBF** indicators; however, our analysis shows that these cases are rare in Racket applications.

*Structures in
Pycket*

5 STRUCTURES IN PYCKET

We implemented the presented optimizations in Pycket, a Racket implementation using the RPython toolchain and its meta-tracing **JIT** compiler.

5.1 RPython and Pycket

[omitted from excerpt]

5.2 Optimization Steps

[omitted from excerpt]

5.3 Eliding Immutable Boolean Fields

Immutable Boolean Field Elision

To benefit from immutable boolean fields, we suggested **immutable boolean field elision (IBFE)** in [section 4.3](#). We chose to use the structure implementation class to represent the **IBF** indicators. As RPython does not support creation of RPython-level classes at run-time, all necessary indicators have to be generated in advance, before translation. However, a very high number of **IBF** classes can severely slow down allocation and possibly start-up time. Therefore, we assume an upper limit to the number of fields we consider for **IBFE**. The amount of indicators that are necessary for a given limit l is $\binom{l}{1} + \binom{l}{2} + \dots + \binom{l}{l}$. In Pycket, we chose 5 as the default limit, resulting in 21 pre-defined **IBF** indicator classes. This seems sufficient, given the average size of Racket structures but not overly restrictive, as it covers over 90% of the structure type encountered in the Racket standard library (cf. [section 3](#)). Nevertheless, all **IBF** indicator classes are subjected to the inlining described above, so that each **IBF** indicator is actually represented by 12 classes for the field inlining.

Hence, when instantiating a structure, Pycket has 252 structure classes to choose from. The operation that maps from all **IBF** positions to the matching structure class benefits from a lexicographical order of all structure classes; the combination of **#f** positions determines the position of a structure class uniquely. During instantiation, all positions of immutable fields about to be initialized with **#f** are shifted to account for their elision. This can also help the inlining optimization, as larger structures with many **IBFs** now can potentially use an inlined representation instead of a split one.

Accessing an **IBF** is cheap; with **IBFE** we make sure that all accesses to those fields are in constant time.

5.4 A Note on Unboxing

[omitted from excerpt]

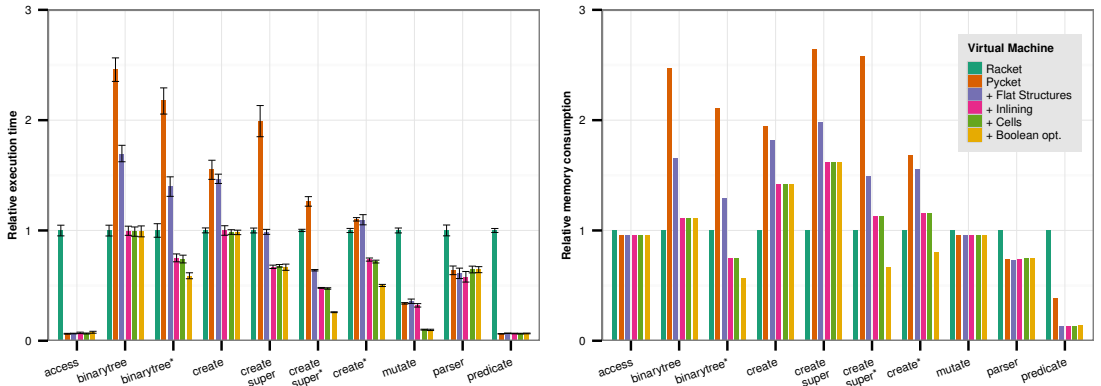


FIGURE D: Benchmark results with execution times (left) and memory consumption (right) normalized to Racket. Lower is better.

5.5 Implementation Summary

Overall, the whole structures implementation in Pycket includes 15 implementation classes, about 30 structure primitives, and about 50 general primitives, totalling in about 2000 lines of RPython code.

6 EVALUATION

Pycket is not yet a feature-complete Racket implementation and due to pending (non-structure related) features, the existing Racket structure benchmarks do not run yet. We therefore use a set of micro-benchmarks⁷ instead. We provide an evaluation and execution time and memory consumption based on these benchmarks.

SETUP All benchmarks were run on an Intel Core i5 (Haswell) at 1.3 GHz with 3 MB cache and 8 GB of RAM under OSX 10.10.2. All micro-benchmarks

⁷<https://github.com/vkirilichev/pycket-structs-benchmarks> (visited 2015-12-05)

are single-threaded. RPython at revision a10c97822d2a was used for translating Pycket. Racket v6.2.0.4 and Pycket at revision 3do229f were used for benchmarking.

METHODOLOGY Every micro-benchmark was run five times uninterrupted. The execution time was measured *in-system* and, hence, it does not include start-up time. However, it does include warm-up time and the time needed for JIT compilation. We show the execution times of all runs relative to Racket with bootstrapped [10] confidence intervals for a 95 % confidence level. The memory consumption was measured as maximum resident set size and is given relative to Racket; the confidence intervals were negligibly small and have been omitted.

6.1 Micro-benchmarks

The micro-benchmark set consists of ten tests. Besides examining basic operations, such as structure creation, call of the predicate procedure and accessing and mutating structure fields, we include two slightly more realistic use-cases.

6.1.1 Basic Operations

We used the following benchmarks for the basic operations: *create* creates simple structures representing two-dimensional coordinates with integer values; *create/super* re-uses the *create* benchmarks, but adds a third dimension using structure type inheritance; *create** is the same as *create*, but with an **IBF** as first field; *create/super** is the same as *create/super*, but with an **IBF** as first field; *predicate* checks the type of given structures including the whole type hierarchy; *access* performs accesses to various immutable fields of structures; and *mutate* changes every value of a structure and reads the stored value afterwards on each loop iteration. Each benchmark essentially contains a loop with few basic operations and collects the result in a variable to avoid elimination.

6.1.2 Binary Tree

In the *binary tree* benchmark, the base structure type represents a leaf, which has only a value. A node is a subtype of the leaf referencing two other nodes.

This benchmark tests several operation with structures of multiple types simultaneously. We use two versions of this micro-benchmark, where values of leaves are integers (*binarytree*) and booleans (*binarytree**), respectively.

6.1.3 Parser

The *parser* benchmark is a Brainfuck⁸ interpreter. It creates one instance of a structure referencing a list and a data pointer. The operations on the structure include mutations of the data pointer and accessing list elements, and hence, the *parser* benchmark tests the structure’s accessor and mutator, but not the constructor. The benchmark’s interpreter executes a simple program that generates a Sierpinsky triangle several times.

6.2 Optimization Impact and Results

We report the impact of all optimizations on execution time and memory consumption. The final performance results of optimized Pycket are shown in figure D. Note that we accumulate optimization, as they form dependencies. Hence, for example, *inlining* includes *flat structures*. By way of example, we show the validity of the predicted memory saving, using the *create*, *create/super*, and *binarytree* benchmarks. For **IBFE**, we however use their boolean counterparts *create**, *create/super**, and *binarytree**.

[omitted from excerpt]

6.2.3 Immutable Boolean Field Elision

All benchmarks with **IBFs** — that is *create**, *create/super** and *binarytree** — achieve a speed-up and reduced memory consumption. In these particular benchmarks, the execution time becomes about 30 % faster. Memory savings range from 25 to 40 %. At the same time, all other benchmarks are virtually untouched, showing next to no disadvantages of employing **IBFE**.

(“+ Booleans opt.”)

⁸Brainfuck is an esoteric programming language that models a Turing machine with eight operations on an array.

The actual memory saving, according to Equation 4, should be $n \cdot 2.3 \cdot 0.26 \cdot 0.7 \cdot 0.85 - \text{sizeOfSpecializedClasses} \approx 0.35n$ words, for n structure instances. However, the benchmarks deviate from the average numbers in the sense that the use of IBFs is well known and the three boolean-related benchmarks yield different but expected results, Also Pycket's automatic unboxing of small structure applies (cf. section 5.4). The size of the specialized classes turned out to be insignificantly low.

*Immutable
Boolean
Field Elision*

*create** Structures have one IBF per (two-field) instance, always being *false*, all fields immutable, yielding $n \cdot 2 \cdot \frac{1}{2} = n$. Considering Pycket's automatic unboxing, *create** makes use of Racket's a *fixnum* for the second structure field. Hence, compared with the cells optimization level, additional two words per structure are saved, yielding $n \cdot (2 + 2 \cdot \frac{1}{2}) = 3n$. Having 15 000 000 structure instances for *create**, we should save

$$3n = 3 \cdot 15\,000\,000 \cdot 64 \text{ bit} \approx 343.3 \text{ MB.}$$

The measured result 344.6 MB differs only slightly.

*create/super** Structures have two IBF per (three-field) instance, always being *false*, all fields immutable, yielding $n \cdot 3 \cdot \frac{2}{3} = 2n$. However, the third field being a Racket *fixnum*, and the number of *actual* fields dropping from three to one due to IBFE, Pycket's unboxing applies, and additional two words will be saved per structure instance, eventually yielding $n \cdot (2 + 3 \cdot \frac{2}{3}) = 4n$. Having 30 000 000 structure instances *create/super**, we should save

$$4n = 4 \cdot 30\,000\,000 \cdot 64 \text{ bit} \approx 915.5 \text{ MB.}$$

The measured result 881.8 MB deviates less than 4%.

*binarytree** Structures have one IBF with a *false* per instance, yielding n . With a tree depth of 22, we should save

$$n = 2^{23} \cdot 64 \text{ bit} = 64 \text{ MB}$$

which matches the measured result exactly.

6.3 Limitations

We only evaluated the efficiency of structures in Pycket on self-written benchmarks. Although they are well suited to test performance of basic operations with structures, real-world applications may show different behavior as part of future work. Once feasible, more elaborate benchmarks will be used.

JIT warm-up time has an impact on execution time. We use our benchmarks with a sufficient warm-up time, which is not guaranteed to be always reachable in real-world applications. Also, warm-up time may differ between benchmarks. In order to illustrate the importance of the sufficient warm-up time in micro-benchmarks, we ran the *create/super* micro-benchmark with different numbers of iterations. The results of this benchmark are presented in figure E. Pycket shows pure performance results with a small number of iterations, but starting with some sufficient number (about 30 millions in this particular micro-benchmark), Pycket is continuously faster. The slowness of Pycket at the beginning arise from the JIT warm-up. Therefore, we use different, sufficiently large numbers of iterations in every benchmark to show the well-established performance.

Finally, we are unable to influence internal CPU optimizations, such as enabling a boost-mode. However, such optimizations should work same for both Racket and Pycket running single threaded.

*Related
Work*

7 RELATED WORK

[omitted from excerpt]

8 CONCLUSION AND FUTURE WORK

We presented an analysis of record structure usage in Racket and proposed optimizations that are fit for an efficient implementation. We considered three common approaches and devised a novel optimization for immutable boolean

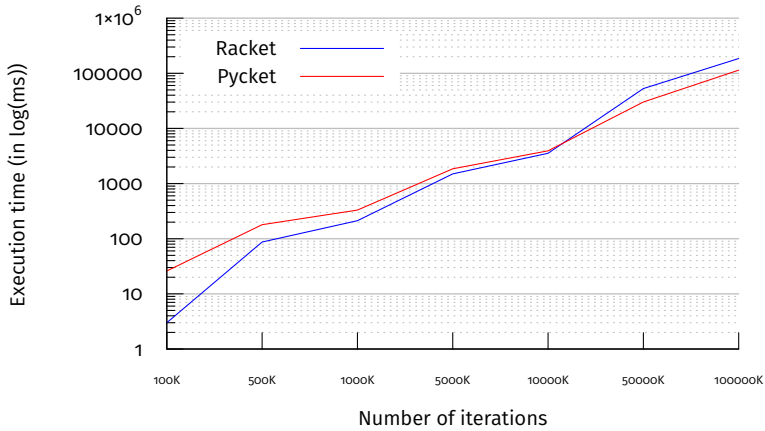


FIGURE E: Execution times (in log(ms)) of *create/super* micro-benchmarks for Racket and Pycket with different number of iterations illustrate the influence of JIT warm-up. Lower is better.

fields. We applied these approaches to Pycket, a tracing-JIT-based implementation of Racket, and achieve a significant speed-up compared to Racket in provided micro-benchmarks with a sufficient warm-up time. We evaluated the impact of our optimizations with a set of micro-benchmarks.

Our results suggest further investigation of *unboxing* values, as homogenised fields in structures make up about 85% in Racket on average. *Adaptive optimizations* [19] show promising initial results and may be applied to records in the future. Finally, once Pycket’s feature coverage is sufficient, we will run a broader range of benchmarks.

[omitted from excerpt]

REFERENCES

- [2] David F Bacon, Stephen J Fink, and David Grove. “Space- and time-efficient implementation of the Java object model”. In: *ECOOP 2002 — Object-Oriented Programming*. Edited by Boris Magnusson. Volume 2374. Lecture Notes in Computer Science. Springer, 2002, pages 111–132. DOI: [10.1007/3-540-47993-7_5](https://doi.org/10.1007/3-540-47993-7_5).
- [5] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. “Storage Strategies for Collections in Dynamically Typed Languages”. In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pages 167–182. ISBN: 978-1-4503-2374-1. DOI: [10.1145/2509136.2509531](https://doi.org/10.1145/2509136.2509531).
- [10] Anthony Christopher Davison and David V. Hinkley. In: *Bootstrap Methods and Their Application*. Cambridge, 1997. Chapter 5.
- [13] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. “Scheme with Classes, Mixins, and Traits”. In: *Programming Languages and Systems*. Edited by Naoki Kobayashi. Volume 4279. LNCS. Springer, 2006, pages 270–289. ISBN: 978-3-540-48937-5. DOI: [10.1007/11924661_17](https://doi.org/10.1007/11924661_17).
- [14] Richard P. Gabriel. “Lisp: Good News, Bad News, How to Win Big”. In: *AI Expert 6.6* (1991), pages 30–39.
- [18] Michael E. Noth. “Exploding Java Objects for Performance”. PhD thesis. University of Washington, 2003. HDL: [1773/6889](https://hdl.handle.net/1773/6889).
- [19] Tobias Pape, Carl Friedrich Bolz, and Robert Hirschfeld. “Adaptive Just-in-time Value Class Optimization: Transparent Data Structure Inlining for Fast Execution”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. Volume 2. SAC ’15. Salamanca, Spain: ACM, 2015. ISBN: 978-1-4503-3196-8. DOI: [10.1145/2695664.2695837](https://doi.org/10.1145/2695664.2695837).

Bibliography

- [1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification Version 1.0 β* . last seen at <http://web.cs.ucla.edu/~palsberg/course/cs239/papers/fortress1.0beta.pdf>. Sun Microsystems, Mar. 6, 2007.
- [2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. “RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages”. In: *Proceedings of the 2007 Symposium on Dynamic Languages*. DLS ’07. Montreal, Quebec, Canada: ACM, 2007, pages 53–64. ISBN: 978-1-59593-868-8. DOI: 10.1145/1297081.1297091.
- [3] Richard Artoul. *Javascript Hidden Classes and Inline Caching in V8*. Apr. 26, 2015. URL: <https://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html> (last accessed 2019-05-28).
- [4] John Aycock. “A Brief History of Just-in-time”. In: *ACM Computing Surveys* 35.2 (June 2003), pages 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.
- [5] David Bacon, Stephen Fink, and David Grove. “Space- and Time-Efficient Implementation of the Java Object Model”. In: *ECOOP 2002 — Object-Oriented Programming*. Edited by Boris Magnusson. Volume 2374. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, May 2002, pages 111–132. DOI: 10.1007/3-540-47993-7_5.
- [6] David F. Bacon. “Kava: a Java dialect with a uniform object model for lightweight classes”. In: *Concurrency and Computation: Practice and Experience* 15.3-5 (Feb. 12, 2003), pages 185–206. DOI: 10.1002/cpe.653.
- [7] Henry G. Baker. “Equal Rights for Functional Objects or, the More Things Change, the More They Are the Same”. In: *SIGPLAN OOPS Messenger* 4.4 (Oct. 1993), pages 2–27. ISSN: 1055-6400. DOI: 10.1145/165593.165596.

- [8] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. “Dynamo: A Transparent Dynamic Optimization System”. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI ’00. Vancouver, British Columbia, Canada: ACM, 2000, pages 1–12. ISBN: 1-58113-199-2. DOI: [10.1145/349299.349303](https://doi.org/10.1145/349299.349303).
- [9] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. “Virtual Machine Warmup Blows Hot and Cold”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), 52:1–52:27. ISSN: 2475-1421. DOI: [10.1145/3133876](https://doi.org/10.1145/3133876). arXiv: [1602.00602v6](https://arxiv.org/abs/1602.00602v6) [cs.PL].
- [10] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Krilichev, Tobias Pape, Jeremy Siek, and Sam Tobin-Hochstadt. “Pycket: A Tracing JIT For a Functional Language”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’15. Vancouver, British Columbia, Canada: ACM, 2015, pages 22–34. ISBN: 978-1-4503-3669-7. DOI: [10.1145/2784731.2784740](https://doi.org/10.1145/2784731.2784740).
- [11] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. “SPUR: A Trace-based JIT Compiler for CIL”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’10. Reno/Tahoe, Nevada, USA: ACM, Oct. 2010, pages 708–725. DOI: [10.1145/1869459.1869517](https://doi.org/10.1145/1869459.1869517).
- [12] Carl Friedrich Bolz. “Meta-tracing just-in-time compilation for RPython”. PhD thesis. Mathematisch-Naturwissenschaftliche Fakultät, Heinrich Heine Universität Düsseldorf, 2012.
- [13] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. “Allocation Removal by Partial Evaluation in a Tracing JIT”. In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM ’11. Austin, Texas, USA: ACM, 2011, pages 43–52. ISBN: 978-1-4503-0485-6. DOI: [10.1145/1929501.1929508](https://doi.org/10.1145/1929501.1929508).
- [14] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. “Runtime Feedback in a Meta-tracing JIT for Efficient Dynamic Languages”. In: *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages*,

- Programs and Systems*. IC00OLPS '11. Lancaster, United Kingdom: ACM, 2011, 9:1–9:8. ISBN: 978-1-4503-0894-6. DOI: 10.1145/2069172.2069181.
- [15] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. “Tracing the Meta-level: PyPy’s Tracing JIT Compiler”. In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. IC00OLPS '09. Genova, Italy: ACM, 2009, pages 18–25. ISBN: 978-1-60558-541-3. DOI: 10.1145/1565824.1565827.
- [16] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. “Storage Strategies for Collections in Dynamically Typed Languages”. In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pages 167–182. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509531.
- [17] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. “Back to the Future in One Week — Implementing a Smalltalk VM in PyPy”. In: *Self-Sustaining Systems*. Volume 5146. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pages 123–139. DOI: 10.1007/978-3-540-89275-5_7.
- [18] Carl Friedrich Bolz, Tobias Pape, Jeremy Siek, and Sam Tobin-Hochstadt. “Meta-tracing makes a fast Racket”. In: *Dyla'14*. Edinburgh, United Kingdom, June 2014.
- [19] Carl Friedrich Bolz and Laurence Tratt. “The impact of meta-tracing on VM design and implementation”. In: *Science of Computer Programming* (2013). ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.02.001.
- [20] Gilad Bracha. *Newspeak Programming Language Draft Specification, Version 0.1*. Feb. 8, 2018. URL: <http://newspeaklanguage.org/spec/newspeak-spec.pdf>.
- [21] Gilad Bracha, Peter Ahe, Vassili Bykov, Yaron Kashi, and Eliot Miranda. *The Newspeak Programming Platform*. Technical report. Cadence Design Systems, May 6, 2008.
- [22] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. “The Jalapeño Dynamic Optimizing Compiler for Java”.

- In: *Proceedings of the ACM 1999 Conference on Java Grande*. JAVA '99. San Francisco, California, USA: ACM, 1999, pages 129–141. ISBN: 1-58113-161-5. DOI: 10.1145/304065.304113.
- [23] Benjamin Cérat and Marc Feeley. “Structure Vectors and their Implementation”. In: *Scheme and Functional Programming Workshop*. 2014.
- [24] Craig Chambers, David Ungar, and Elgin Lee. “An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes”. In: *SIGPLAN Notices* 24.10 (Sept. 1989), pages 49–70. ISSN: 0362-1340. DOI: 10.1145/74878.74884.
- [25] Maxime Chevalier-Boisvert and Marc Feeley. “Extending Basic Block Versioning with Typed Object Shapes”. In: (2015). arXiv: 1507.02437 [cs.PL].
- [26] Will Clinger, Anne Hartheimer, and Eric Ost. “Implementation Strategies for Continuations”. In: *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*. LFP '88. Snowbird, Utah, USA: ACM, 1988, pages 124–131. ISBN: 0-89791-273-X. DOI: 10.1145/62678.62692.
- [27] William D. Clinger. “Proper Tail Recursion and Space Efficiency”. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: ACM, 1998, pages 174–185. ISBN: 0-89791-987-4. DOI: 10.1145/277650.277719.
- [28] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. “Exploring Language Support for Immutability”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, May 2016, pages 736–747. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884798.
- [29] John Cocke. “Global Common Subexpression Elimination”. In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, July 1970, pages 20–24. DOI: 10.1145/800028.808480.
- [30] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*. Volume 1. Studies in logic and the foundations of mathematics. North Holland Publishing Company, 1958. ISBN: 978-0-7204-2208-5.
- [31] Luis Damas and Robin Milner. “Principal Type-schemes for Functional Programs”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, 1982, pages 207–212. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176.

- [32] Anthony Christopher Davison and David V. Hinkley. “Confidence Intervals”. In: *Bootstrap Methods and Their Application*. Cambridge, 1997. Chapter 5.
- [33] Harry Deutsch and Pawel Garbacz. “Relative Identity”. In: *The Stanford Encyclopedia of Philosophy*. Edited by Edward N. Zalta. Fall 2018. Metaphysics Research Lab, Stanford University, Aug. 17, 2018.
- [34] L. Peter Deutsch and Allan M. Schiffman. “Efficient Implementation of the Smalltalk-80 System”. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’84. Salt Lake City, Utah, USA: ACM, 1984, pages 297–302. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800542.
- [35] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. “Making Data Structures Persistent”. In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. STOC ’86. Berkeley, California, USA: ACM, 1986, pages 109–121. ISBN: 978-0-89791-193-1. DOI: 10.1145/12130.12142.
- [36] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. “Traits: A mechanism for fine-grained reuse”. In: *ACM Transactions on Programming Languages and Systems* 28.2 (Mar. 2006), pages 331–388. ISSN: 0164-0925. DOI: 10.1145/1119479.1119483.
- [37] *ECMA-334: C# Language Specification*. Standard ECMA-334:2017. Also ISO/IEC 23270:2018. Ecma International, Dec. 2017.
- [38] *ECMA-335: Common Language Infrastructure (CLI)*. Standard ECMA-335. Also ISO/IEC 23271. Ecma International, June 2012.
- [39] Werner Eichelbaum. “Klima-Atlas von Schleswig-Holstein, Hamburg und Bremen”. Review of *Klima-Atlas von Schleswig-Holstein, Hamburg und Bremen*, by Deutscher Wetterdienst, 1967. In: *Petermanns geographische Mitteilungen* 114 (1970), page 79.
- [40] John Ellis, Pete Kovac, Hans-J. Boehm, and William Clinger. *An Artificial Garbage Collection Benchmark*. Software. 1997.
- [41] Андрей Петрович Ершов. “On Programming of Arithmetic Operations”. In: *Communications of the ACM* 1.8 (Aug. 1958), pages 3–6. ISSN: 0001-0782. DOI: 10.1145/368892.368907.
- [42] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. “How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain”. In: *Proceedings*

- of the 11th Edition of the International Workshop on Smalltalk Technologies. IWST'16. Prague, Czech Republic: ACM, 2016, 21:1–21:10. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991062.
- [43] Matthias Felleisen and Daniel P. Friedman. “Control operators, the SECD-machine and the λ -calculus”. In: *Proceedings of the 2nd Working Conference on Formal Description of Programming Concepts - III*. Edited by Martin Wirsing. Elsevier, 1987, pages 193–217.
- [44] Jean-Christophe Filliâtre and Sylvain Conchon. “Type-safe Modular Hash-consing”. In: *Proceedings of the 2006 Workshop on ML*. ML '06. Portland, Oregon, USA: ACM, 2006, pages 12–19. ISBN: 1-59593-483-9. DOI: 10.1145/1159876.1159880.
- [45] Matthew Flatt and PLT. *Reference: Racket*. Technical report PLT-TR-2010-1. PLT Inc., 2010.
- [46] Richard P. Gabriel. *Performance and evaluation of LISP systems*. Volume 263. Computer Systems. Cambridge, Mass.: MIT press, 1985. ISBN: 978-0-262-07093-5.
- [47] Richard P. Gabriel and Kent M. Pitman. “Endpaper: Technical issues of separation in function cells and value cells”. In: *LISP and Symbolic Computation* 1.1 (June 1, 1988), pages 81–101. ISSN: 1573-0557. DOI: 10.1007/BF01806178.
- [48] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. “Trace-based Just-in-time Type Specialization for Dynamic Languages”. In: *SIGPLAN Notices* 44.6 (June 2009), pages 465–478. ISSN: 0362-1340. DOI: 10.1145/1543135.1542528.
- [49] Andreas Gal, Christian W. Probst, and Michael Franz. “HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices”. In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*. VEE '06. Ottawa, Ontario, Canada: ACM, June 14, 2006, pages 144–153. ISBN: 1-59593-332-8. DOI: 10.1145/1134760.1134780.
- [50] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. “A Short Cut to Deforestation”. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: ACM, 1993, pages 223–232. ISBN: 0-89791-595-X. DOI: 10.1145/165180.165214.

- [51] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. The Blue Book. Boston, MA, USA: Addison-Wesley Longman, 1983. ISBN: 0-201-11371-6.
- [52] Eiichi Goto. *Monocopy and Associative Algorithms in Extended Lisp*. Technical Report TR-74-03. University of Tokyo, Japan, 1974.
- [53] David Gudeman. *Representing type information in dynamically-typed languages*. Technical report TR93-27. University of Arizona at Tucson, Oct. 1993.
- [54] Greg Hewgill, “igaurav”, Peter Mortensen, Martijn Pieters, and Jim Fasaki Hilliard. “*is*” operator behaves unexpectedly with integers - Stack Overflow. Jan. 23, 2016. URL: <https://stackoverflow.com/q/306313> (last accessed 2019-05-09).
- [55] Rich Hickey. “The Clojure Programming Language”. In: *Proceedings of the 2008 Symposium on Dynamic Languages*. DLS ’08. invited talk. Paphos, Cyprus: ACM, 2008, 1:1-1:1. ISBN: 978-1-60558-270-2. DOI: 10.1145/1408681.1408682.
- [56] Rich Hickey. “The Value of Values”. Keynote at GOTO Copenhagen Conference 2012. May 22, 2012.
- [57] C. A. R. “Tony” Hoare. “Notes on Data Structuring”. In: Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. “Tony” Hoare. *Structured Programming*. A.P.I.C. Studies in Data Processing 8. London, UK: Academic Press Ltd., 1972. Chapter 1.1. ISBN: 978-0-12-200550-3.
- [58] Thomas Hobbes. *Elements of Philosophy: The first section, concerning body*. Printed by R. & W. Leybourn for Andrew Crooke, 1656.
- [59] Marja Hölttä. *Crankshafting from the ground up*. Technical report. Google, Aug. 2013.
- [60] Daniel H. H. Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. “Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself”. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, Oct. 1997, pages 318–326. DOI: 10.1145/263698.263754.
- [61] “jlandercy”, “Badger”, and Jim Fasaki Hilliard. *Deeper understanding of Python object mechanisms - Stack Overflow*. Dec. 2, 2016. URL: <https://stackoverflow.com/q/40930331> (last accessed 2019-04-23).

- [62] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto, and Victor Ng. *The Definitive Guide to Jython: Python for the Java Platform*. Berkely, CA, USA: Apress, Mar. 5, 2010. ISBN: 978-1-4302-2527-0.
- [63] Tomas Kalibera and Richard Jones. *Quantifying Performance Changes with Effect Size Confidence Intervals*. Technical Report 4–12. University of Kent, June 2012, page 55.
- [64] Tomas Kalibera and Richard Jones. “Rigorous Benchmarking in Reasonable Time”. In: *SIGPLAN Notices* 48.11 (June 2013), pages 63–74. ISSN: 0362-1340. DOI: 10.1145/2555670.2464160.
- [65] Andrew W. Keep and R. Kent Dybvig. “A run-time representation of scheme record types”. In: *Journal of Functional Programming* 24.6 (Special Issue 06 Sept. 2014), pages 675–716. ISSN: 1469–7653. DOI: 10.1017/S0956796814000203.
- [66] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. Boston, Massachusetts: ACM, Oct. 1973, pages 194–206. DOI: 10.1145/512927.512945.
- [67] Bent Bruun Kristensen, Ole Lehrmann Madsen, and Birger Møller-Pedersen. “The when, Why and Why Not of the BETA Programming Language”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, June 9, 2007, pages 10-1–10-57. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238854.
- [68] Peter Lee and Mark Leone. “Optimizing ML with Run-time Code Generation”. In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI ’96. Philadelphia, Pennsylvania, USA: ACM, 1996, pages 137–148. ISBN: 0-89791-795-2. DOI: 10.1145/231379.231407.
- [69] Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA, 1990.
- [70] Bruce J. MacLennan. “Values and Objects in Programming Languages”. In: *SIGPLAN Notices* 17.12 (Dec. 1982), pages 70–79. ISSN: 0362-1340. DOI: 10.1145/988164.988172.
- [71] Stefan Marr, Max Leske, Dominik Aumayr, Guido Chari, and Tobias Pape. *smarr/ReBench: 1.0 Release Candidate 2*. Version v1.orc2. Software. June 9, 2019. DOI: 10.5281/zenodo.3242039.

- [72] Stefan Marr and Hanspeter Mössenböck. “Optimizing Communicating Event-Loop Languages with Truffle”. In: *Presentation at 5th International Workshop on Programming based on Actors, Agents, and Decentralized Control*. AGERE!’15. Pittsburgh, PA, USA, Oct. 26, 2015.
- [73] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, revised edition*. MIT Press, 1997.
- [74] Eliot Miranda. “The Cog Smalltalk Virtual Machine: writing a JIT in a high-level dynamic language”. In: *Workshop on Virtual Machines and Intermediate Languages (VMIL)*. 2011.
- [75] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. “Two Decades of Smalltalk VM Development: Live VM Development Through Simulation Tools”. In: *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2018. Boston, MA, USA: ACM, Nov. 4, 2018, pages 57–66. ISBN: 978-1-4503-6071-5. DOI: [10.1145/3281287.3281295](https://doi.org/10.1145/3281287.3281295).
- [76] James George Mitchell. “The Design and Construction of Flexible and Efficient Interactive Programming Systems”. PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 1970.
- [77] Richard Mitchell, Jim McKim, and Bertrand Meyer. *Design by contract, by example*. Addison Wesley, 2001.
- [78] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997. ISBN: 978-1-55860-320-2.
- [79] James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Mark S. Miller. “The Left Hand of Equals”. In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2016. Amsterdam, Netherlands: ACM, 2016, pages 224–237. ISBN: 978-1-4503-4076-2. DOI: [10.1145/2986012.2986031](https://doi.org/10.1145/2986012.2986031).
- [80] Michael E. Noth. “Exploding Java Objects for Performance”. PhD thesis. University of Washington, 2003. HDL: [1773/6889](https://hdl.handle.net/1773/6889).
- [81] Charles O. Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java*. Facets of Ruby. Pragmatic Bookshelf, Feb. 4, 2011. ISBN: 978-1-934356-65-4.
- [82] *Objects By Value: Joint Revised Submission - w/Errata*. OMG TC document orbos/98-01-18. Object Management Group, Jan. 18, 1998.

- [83] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *An overview of the Scala programming language*. Technical report LAP-REPORT-2006-0001. Lausanne, Switzerland: EFPL, 2006.
- [84] Martin Odersky, Jeff Olson, Paul Phillips, and Joshua Suereth. *SIP-15 - Value Classes*. Feb. 7, 2012. URL: <https://docs.scala-lang.org/sips/completed/value-classes.html> (last accessed 2019-04-24).
- [85] Michael Paleczny, Christopher A. Vick, and Cliff Click. “The Java HotSpot™ Server Compiler”. In: *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1*. JVM’01. Monterey, California: USENIX Association, Apr. 24, 2001.
- [86] Tobias Pape. *Theseus benchmarking*. Version 2019. Software. Nov. 27, 2019. DOI: 10.5281/zenodo.3555257.
- [87] Tobias Pape, Spenser Bauman, Caner Dericci, Sam Tobin-Hochstadt, Carl Friedrich Bolz-Tereick, Vasily Kirilichev, Anton Gulenko, Rajan Chandi, Vishesh Yadav, Patrick Maier, Jeremy G. Siek, and Ben Greenman. *Pycket-OnAShip: Pycket with shapes*. Version 2019. Software. Dec. 12, 2019. DOI: 10.5281/zenodo.3572274.
- [88] Tobias Pape, Carl Friedrich Bolz, and Robert Hirschfeld. “Adaptive Just-in-time Value Class Optimization for Lowering Memory Consumption and Improving Execution Time Performance”. In: *Science of Computer Programming* 140 (June 15, 2017), pages 17–29. ISSN: 0167-6423. DOI: 10.1016/j.scico.2016.08.003.
- [89] Tobias Pape, Carl Friedrich Bolz, and Robert Hirschfeld. “Adaptive Just-in-time Value Class Optimization: Transparent Data Structure Inlining for Fast Execution”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. Volume 2. SAC ’15. Salamanca, Spain: ACM, 2015. ISBN: 978-1-4503-3196-8. DOI: 10.1145/2695664.2695837.
- [90] Tobias Pape and Carl Friedrich Bolz-Tereick. *Theseus*. Version 2019. Software. Nov. 27, 2019. DOI: 10.5281/zenodo.3555255.
- [91] Tobias Pape, Tim Felgentreff, Robert Hirschfeld, Anton Gulenko, and Carl Friedrich Bolz. “Language-independent Storage Strategies for tracing-JIT-based Virtual Machines”. In: *Proceedings of the 11th Symposium on Dynamic*

- Languages*. DLS 2015. Pittsburgh, PA, USA: ACM, Oct. 2015, pages 104–113. ISBN: 978-1-4503-3690-1. DOI: 10.1145/2816707.2816716.
- [92] Tobias Pape, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. “Let Them Fail: Towards VM Built-in Behavior That Falls Back to the Program”. In: *Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. Programming ’19. Genova, Italy: ACM, 2019, 35:1–35:7. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3338056.
- [93] Tobias Pape, Tim Felgentreff, Fabio Niephaus, Lars Wassermann, Anton Gulenko, Jakob Reschke, Bastian Kruck, Matthias Springer, Carl Friedrich Bolz-Tereick, Philipp Otto, Johannes Henning, Jan Graichen, Patrick Siegler, Felix Wolff, Jonas Chromik, Daniel Werner, Patrick Rrein, and Stefan Marr. *RSqueakOnABoat: RSqueak with shapes*. Version 2019. Software. Dec. 12, 2019. DOI: 10.5281/zenodo.3572256.
- [94] Tobias Pape, Vasily Kirilichev, Carl Friedrich Bolz, and Robert Hirschfeld. “Record Data Structures in Racket: Usage Analysis and Optimization”. In: *SIGAPP Applied Computing Review* 16.4 (Jan. 2017), pages 25–37. ISSN: 1559-6915. DOI: 10.1145/3040575.3040578.
- [95] Tobias Pape, Vasily Kirilichev, and Robert Hirschfeld. “Optimizing Record Data Structures in Racket”. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. Volume 2. SAC ’16. Pisa, Italy: ACM, 2016, pages 1798–1805. ISBN: 978-1-4503-3739-7. DOI: 10.1145/2851613.2851732.
- [96] “Pegasus”. *Why a=1, b=1, id(a) == id(b) but a=1.0, b=1.0, id(a) != id(b) in python? - Stack Overflow*. Aug. 13, 2015. URL: <https://stackoverflow.com/q/31978476> (last accessed 2019-04-28).
- [97] Plutarch. “Theseus”. In: *Plutarch’s Lives: Translated From the Greek by Several Hands*. To which is prefixt the life of Plutarch. Translated from the Greek by John Dryden. Volume 1. London, 1683.
- [98] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. “Immutability”. In: *Aliasing in Object-Oriented Programming: Types, Analysis and Verification*. Edited by Dave Clarke, James Noble, and Tobias Wrigstad. Volume 7850. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pages 233–269. ISBN: 978-3-642-36946-9. DOI: 10.1007/978-3-642-36946-9_9.

- [99] Armin Rigo and Samuele Pedroni. “PyPy’s approach to virtual machine construction”. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pages 944–953. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176753.
- [100] John Rose. *JEP 169: Value Objects*. May 20, 2015. URL: <http://openjdk.java.net/jeps/169> (last accessed 2019-04-24).
- [101] Cristina Ruggieri and Thomas P. Murtagh. “Lifetime Analysis of Dynamically Allocated Objects”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’88. San Diego, California, USA: ACM, 1988, pages 285–293. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73585.
- [102] Erik Sandewall. “A Proposed Solution to the FUNARG Problem”. In: *SIGSAM Bulletin* 17 (Jan. 1971), pages 29–42. ISSN: 0163-5824. DOI: 10.1145/1093420.1093422.
- [103] Thomas J. Schrader and Christian Haider. “Complex Values in Smalltalk”. In: *Proceedings of the International Workshop on Smalltalk Technologies*. IWST ’09. Brest, France: ACM, 2009, pages 126–137. ISBN: 978-1-60558-899-5. DOI: 10.1145/1735935.1735957.
- [104] Zhong Shao, John H. Reppy, and Andrew Wilson Appel. “Unrolling lists”. In: *SIGPLAN Lisp Pointers* VII.3 (July 1994), pages 185–195. ISSN: 1045-3563. DOI: 10.1145/182590.182453.
- [105] Guy L. Steele, Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. *Fortress (Sun HPCS Language)*. In: *Encyclopedia of Parallel Computing*. Edited by David Padua. Springer, Boston, MA, 2011, pages 718–735. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_190.
- [106] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. “Dynamic Native Optimization of Interpreters”. In: *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*. IVME ’03. San Diego, California: ACM, June 8, 2003, pages 50–57. ISBN: 1-58113-655-2. DOI: 10.1145/858570.858576.

- [107] Gerald Jay Sussman and Guy Lewis Steele Jr. “Scheme: A interpreter for extended lambda calculus”. In: *Higher-Order and Symbolic Computation* 11.4 (Dec. 1, 1998), pages 405–439. ISSN: 1573-0557. DOI: 10.1023/A:1010035624696.
- [108] Akihiko Takano and Erik Meijer. “Shortcut Deforestation in Calculational Form”. In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA ’95. La Jolla, California, USA: ACM, 1995, pages 306–313. ISBN: 0-89791-719-7. DOI: 10.1145/224164.224221.
- [109] Even Wiik Thomassen. “Trace-based just-in-time compiler for Haskell with RPython”. Master’s thesis. Norwegian University of Science and Technology Trondheim, 2013.
- [110] Ben L. Titzer and Jens Palsberg. “Vertical Object Layout and Compression for Fixed Heaps”. In: *Semantics and Algebraic Specification: Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday*. Edited by Jens Palsberg. Volume 5700. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pages 376–408. ISBN: 978-3-642-04164-8. DOI: 10.1007/978-3-642-04164-8_18.
- [111] Vlad Ureche, Eugene Burmako, and Martin Odersky. “Late data layout: unifying data representation transformations”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’14. ACM. Portland, Oregon, USA: ACM, 2014, pages 397–416. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660197.
- [112] Philip Wadler. “Deforestation: transforming programs to eliminate trees”. In: *Theoretical Computer Science* 73.2 (1990), pages 231–248. ISSN: 0304-3975. DOI: 10.1016/0304-3975(90)90147-A.
- [113] Daniel Weinreb and David Moon. *Lisp Machine Manual*. Second Preliminary Version. Cambridge, Massachusetts, USA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Jan. 1979.
- [114] Christian Wimmer. “Automatic object inlining in a Java virtual machine”. PhD thesis. Linz, Austria: Institute for System Software, Johannes Kepler University, May 27, 2008. ISBN: 978-3-85499-417-6.

- [115] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Hanspeter Mössenböck, and Christian Humer. “An Object Storage Model for the Truffle Language Implementation Framework”. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. Cracow, Poland: ACM, 2014, pages 133–144. ISBN: 978-1-4503-2926-2. DOI: [10.1145/2647508.2647517](https://doi.org/10.1145/2647508.2647517).
- [116] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! '13. Indianapolis, Indiana, USA: ACM, 2013, pages 187–204. ISBN: 978-1-4503-2472-4. DOI: [10.1145/2509578.2509581](https://doi.org/10.1145/2509578.2509581).
- [117] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. “Self-optimizing AST Interpreters”. In: *Proceedings of the 8th Symposium on Dynamic Languages*. DLS '12. Tucson, Arizona, USA: ACM, 2012, pages 73–82. ISBN: 978-1-4503-1564-7. DOI: [10.1145/2384577.2384587](https://doi.org/10.1145/2384577.2384587).

COLOPHON

This book is set 11/13½ pt on a 26 pc measure. The text face is Georg Mayr-Duffner and Octavio Pardo's EB Garamond, a digitization of cuts by Claude Garamond and *Robert Granjon* as found on the Egenolff-Berner specimen. It is accompanied by Fira Sans, designed by Erik Spiekermann, Ralph du Carrois, Anja Meiners and Botio Nikoltchev, issued by bBoxType, Berlin.



S. D. G.



