Science of Computer Programming 140 (2017) 17–29

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Adaptive just-in-time value class optimization for lowering memory consumption and improving execution time performance

Tobias Pape [a,*], Carl Friedrich Bolz [b], Robert Hirschfeld [a]

[a] Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany
[b] Software Development Team, King's College London, UK

ARTICLE INFO

Article history:
Received 20 December 2015
Received in revised form 21 June 2016
Accepted 5 August 2016
Available online 20 August 2016

Keywords:
Meta-tracing
JIT
Data structure optimization
Value classes

ABSTRACT

The performance of value classes is highly dependent on how they are represented in the virtual machine. Value class instances are immutable, have no identity, and can only refer to other value objects or primitive values and since they should be very lightweight and fast, it is important to optimize them carefully. In this paper we present a technique to detect and compress common patterns of value class usage to improve memory usage and performance. The technique identifies patterns of frequent value object references and introduces abbreviated forms for them. This allows to store multiple inter-referenced value objects in an inlined memory representation, reducing the overhead stemming from meta-data and object references. Applied to a small prototype and an implementation of the Racket language, we found improvements in memory usage and execution time for several micro-benchmarks.

1. Introduction

The way data structures are represented affects their performance. Especially virtual machine developers carefully choose the representation of their data structures, classes, or objects so that using them is efficient. In this paper we propose, implement, and evaluate an optimized representation for *value classes* [1] on the virtual machine level. Value class instances are immutable objects without identity that can reference only other value classes instances or primitive data. They have been suggested for an extended Java [1], Java itself [2], exist in .NET [3] and—in a limited form—in Scala [4]. However, related constructs of immutable identity-less structures also occur in several other languages, particularly in functional ones. Examples include the algebraic data types of ML and Haskell, Prolog's terms, cons cells in certain LISPs,[1] and structures in Racket [5]. Therefore, our optimization should be applicable to a number of other contexts. Nevertheless, in this paper we will use the terminology *value classes* and *instances of value classes* (*value objects* for short).

The simplest approach to a machine representation of value objects is a class pointer together with their fields as a list of pointers to other value objects and primitive values. We propose an object layout that stores nested value object groups in a compacted, linearized fashion. This works by observing that in practice some shapes in the object graph are much more

* Corresponding author.
E-mail addresses: tobias.pape@hpi.uni-potsdam.de (T. Pape), cfbolz@gmx.de (C.F. Bolz), hirschfeld@hpi.uni-potsdam.de (R. Hirschfeld).

[1] This is a special case, since LISP only supports one "value type", `cons`. Also, other LISPs exist where cons cells do have identity or are mutable.

http://dx.doi.org/10.1016/j.scico.2016.08.003
0167-6423/

common than other shapes. There are often repeating patterns of how value objects reference each other. For example, a cons cell is likely to reference another cons cell in its tail field, or a tree node often references other tree nodes.

For such common shapes we *inline* the fields of the referenced value object into the referring object to save space and to accelerate the traversal of the object graph. This inlining can be repeated with fields of nested value objects, potentially several levels deep. We detect which object graph shapes are common by keeping statistics at run-time, since it is often impossible to statically infer what shapes will be common in practice.[2] The inlining is only possible because of the key properties of value objects:

a) Value objects are immutable, so the reference to an inlined object can never be replaced by another reference.
b) Value objects do not have identity, so the fact that an inlined object does not have a separate memory address that can be used as its identity does not create problems. Likewise, multiple copies of an inlined object are not problematic for identity concerns.

We implement the proposed optimization in two prototypes. One implements a variant of the lambda calculus extended with value objects and pattern matching, which we used to prototype and evaluate the proposed optimization in isolation. To also evaluate the approach in a more realistic setting, we implemented the same optimization for Pycket [6], a re-implementation of the Racket language. Both languages use the RPython virtual machine implementation framework and its tracing just-in-time (JIT) compiler. The tracing JIT compiler is instrumental to our approach since it is responsible for producing fast machine code for accessing the modified representation.

The contributions of this paper are as follows:

- We propose an approach for finding patterns in value object usage at run-time.
- We present a compressed layout for value objects that makes use of those patterns to store value objects more efficiently.
- We report on the performance of micro-benchmarks for a small prototype language and a Racket implementation.

The paper is structured as follows. Section 2 gives a brief introduction to tracing JIT compilers. In section 3, we present our approach to just-in-time optimization of data structures. Our two implementations are presented briefly in section 4 and their performance is evaluated in section 5. Our approach is put into context in section 6 and we conclude in section 7.

2. Tracing just-in-time compilers

We briefly introduce tracing just-in-time (JIT) compilers [7], as some of their properties are key to the performance characteristics of our approach (cf. section 3.2 and section 3.3).

Just-in-time (JIT) compilation has become a mainstream technique for, among other reasons, speeding up 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 [8] to today's popular languages such as Java [9] or JavaScript [10].

One approach to writing JIT compilers is using *tracing* [11]. 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 be optimized independently or transformed to machine code and used instead of the interpreter to execute the same part of that program [12] 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, it switches back to use the interpreter. Tracing JIT compilers have been successfully used for optimizing native code [11] and also for efficiently executing object-oriented programs [13].

Meta-tracing 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 [14, 15]. Therefore, it is not necessary for language implementers 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 [16]. RPython's tracing JIT also contains a very powerful escape analysis [17], which is an important building block for the optimization described in this paper. Meta-tracing has been most prominently applied to Python with PyPy [18].

3. Optimization approach

Our optimization uses an unconventional memory representation for value objects within the virtual machine to save memory and to speed up access. The optimization stays invisible to the programmer.

[2] Note that these shapes are totally different what some JavaScript VMs such as Firefox' IonMonkey and Higgs call shapes. Those JavaScript "shapes" are equivalent to Self *maps* or V8's hidden classes. We will discuss the relationship to Self maps in the related work section.

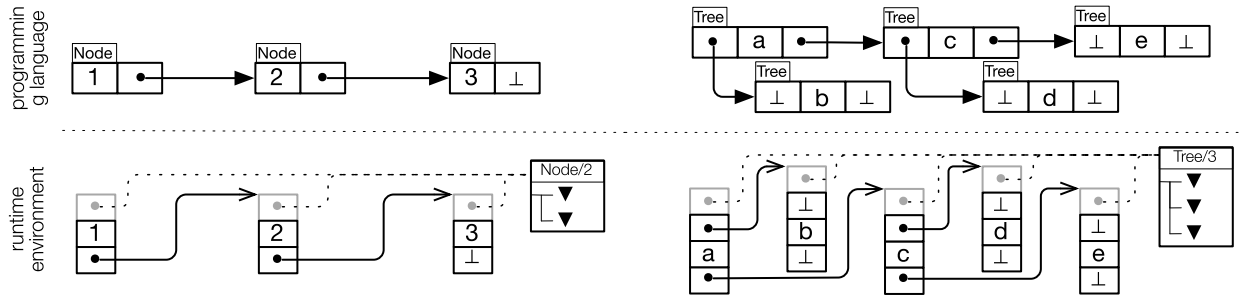
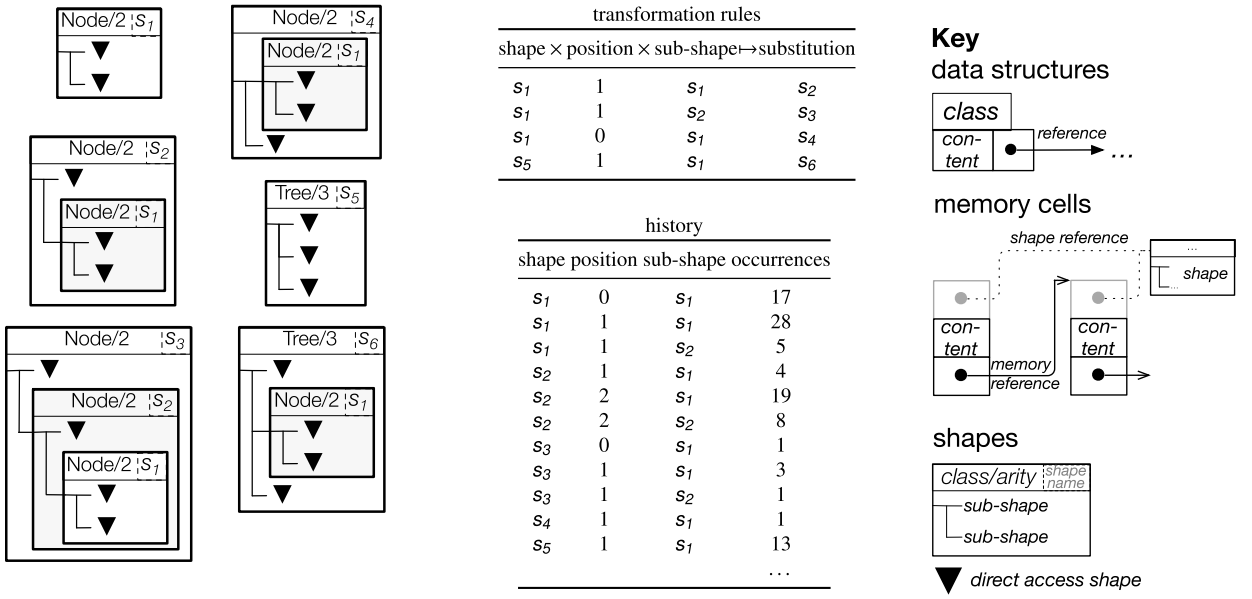


Fig. 1. Straightforward value class representation for a linked list and a tree. Top: the language view; bottom: runtime environment view with *storage* and *shape*.



transformation rules

shape × position × sub-shape ↦ substitution			
s_1	1	s_1	s_2
s_1	1	s_2	s_3
s_1	0	s_1	s_4
s_5	1	s_1	s_6

history

shape	position	sub-shape	occurrences
s_1	0	s_1	17
s_1	1	s_1	28
s_1	1	s_2	5
s_2	1	s_1	4
s_2	2	s_1	19
s_2	2	s_2	8
s_3	0	s_1	1
s_3	1	s_1	3
s_3	1	s_2	1
s_4	1	s_1	1
s_5	1	s_1	13
			...

Fig. 2. Left: Shapes comprise a class reference, an arity, and a structure of sub-shapes. Center: "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 value object creation. Right: Key to the visual language used.

A straightforward representation for a value object in memory is a chunk of memory that stores a reference to the object's class first, followed by references for each of its fields. We call the latter the *storage* of the object. An example of this straightforward representation can be seen in Fig. 1, which shows a linked list and a tree structure.

The 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 form to store the pattern. Newly created instances that exhibit this pattern use the abbreviated form to save memory.

The abbreviated form uses object inlining for instances with these common patterns. 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 the inlining process, we need to maintain certain meta-information to keep track of which fields belong to which level of an inlined object and in order to remember the classes of the inlined objects. Therefore, we replace the pointer to the object's class with a pointer to this meta-information, which we call the *shape* of the object. If no inlining occurs, we still give the object a shape, which only references the class and the fact that no inlining is being performed. This is called the *default shape* of a class.

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.

To understand the rest of the system, we now need to look at (a) how structure patterns are recognized, (b) how the construction of values ensures the proper usage of shapes, and (c) how the access to of inlined fields is implemented.

3.1. Shapes and their recognition

A *shape* describes the abstract, structural representation of value objects. It is shared between all identically structured instances of the same value class[3] and captures the structure of these instances. Value objects have a permanent reference to their shape during their life time.

Shapes can be nested; they consist of sub-shapes for each field in a value object's storage. 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. Value objects with these shapes are treated as black boxes, for example scalar data or unoptimized objects that are stored directly. This is depicted in the bottom part of Fig. 1; all three nodes in the list share the same shape, which denotes that each node consists of two references with *direct access* shapes. The same holds for the nodes of the tree in that figure, but with three references.

As long as no optimization has taken place, a value object refers to the *default shape* of its value class that solely consist of *direct access* sub-shapes. The shapes in Fig. 1 are the default shapes for their value classes. Initially, all value object 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 value objects. We explain this profiling data, which we call the *history*, in subsubsection 3.1.1.

Based on the history profiles, we determine the fields in a value class where inlining value objects could be worthwhile. We infer new shapes for value objects with certain referenced value objects inlined, and record a transition from the old to the new shape. We call this process *shape recognition* and explain it in subsubsection 3.1.2.

We collect all results from the shape recognition in a table that we call the *transformation rules*. We explain its structure briefly in subsubsection 3.1.3.

3.1.1. History

The history is a table that counts how often certain sub-shapes are found in the fields of new value objects. It is essentially a histogram of all sub-shapes. It is rather simple to maintain, as due to the immutability of value objects, modifications of this table are only necessary during value object creation. At this point, all objects that will constitute a new value object are available and we can count the occurrences of *sub-shapes* at specific positions in the value object.

As example, the history table in Fig. 2 shows that for shape s_1 at position 1, the shape s_1 itself has been encountered 17 times as sub-shape, while shape s_2 has been encountered 5 times as sub-shape 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.

3.1.2. Shape recognition

During the creation of a value object 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, create a new shape that combines the value object'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.

Considering Fig. 2 as example, shape s_2 would be the result of turning the history entry $(s_1, 1, s_1, 17)$ into the transformation rule $(s_1, 1, s_1) \mapsto s_2$.

3.1.3. Transformation rules

We maintain the set of all transformation rules as a lookup table that is used during value object 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. Find an example transformation rule table in the top center of Fig. 2.

Note that we consider both history and transformation rules to *conceptually* be tables. Depending on circumstances it may be advisable to merge them into one table or split them by the first column's entries and attach them directly to those shape.

3.2. Compaction through inlining

The information of what shapes occur often and which shape transformations to use can be applied at run-time to create value objects in a compacted representation. The process of creating such a compacted value object is outlined in the following. As running example, we will use the combination of the primitive datum "1" with a linked list into a new linked list as depicted in Fig. 3 and using the shapes and transformation rules as given in Fig. 2.

[3] We refer to a value class by its name and the arity of its type in a Prolog style, for example Node/2 for binary node objects.

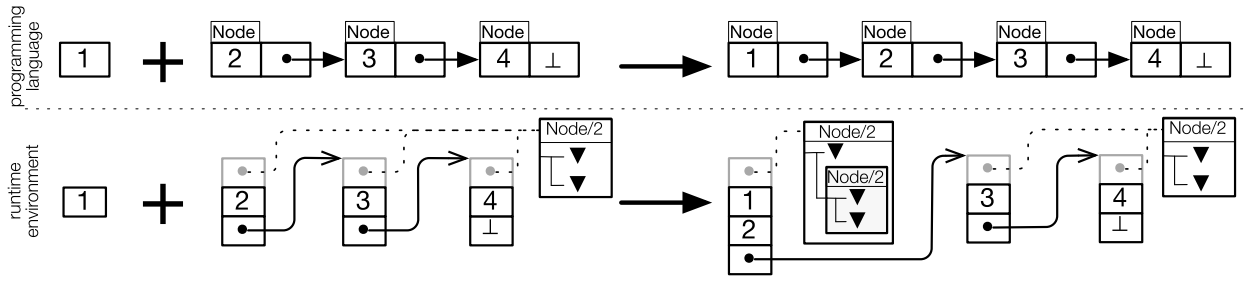


Fig. 3. When creating a new node value object that should contain "1" and the list "Node/2[2, Node/2[3, Node/2[4, ⊥]]]", a new value object that merges the "1" with the "2" object and a different shape is created instead.

First, it is only necessary to consider compaction when creating new value objects. Since they are immutable, there is no need to consider compaction 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 our example, the class is Node/2 and the new fields are "1" and a Node/2 value object ("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 Fig. 2. With the default shape and the 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, ...]".

Algorithm 1: Determining shape and fields of a value object during its creation. The shape is derived based on transformation rules and the fields are inlined based on the resulting shape.

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

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 value object f_i with a copy of its *storage* in the new fields f (line 7); the value object f_i is now *inlined*. The new shape s' becomes the new value object's shape s (line 8) and the inlining process is *restarted* (line 10) with the new shape and fields. This allows possible other transformation rules to be applied due to the shape change.

Once no further transitions are found, the value object's shape s and the current fields f are returned as the shape and storage of the new value object (line 16).

For 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_2 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.[4] The fields of f are now "1", "2", and "Node/2[3, ...]". Furthermore, we change the shape of the new value object to s_2 (line 8). At that point, we restart the inlining process by resetting the counter (line 9). 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.

[4] The original value object "Node/2[2, ...]" remains untouched and can still be referenced from other objects.

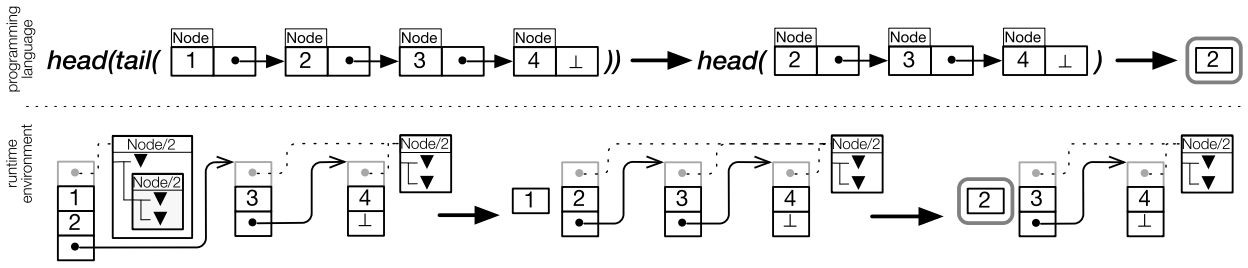


Fig. 4. Referenced value object reification. Accessing the second item 2 of the list $l \leftarrow \text{Node/2}[1, \text{Node/2}[2, \text{Node/2}[3, \text{Node/2}[4, \bot]]]]$ by two operations $head(tail(l))$ results in two reified rest lists to be created.

We continue with the third field f_2, which is "Node/2[3, ...]". The sub-shape of this value object is s_1, and since s is s_2, we can look up a transformation rule for $(s_2, 2, 2_1)$ 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 value object's new shape s_2 and its new fields $[1, 2, \text{Node/2}[3, \ldots]]$.

During the inlining process, potentially short-lived objects might be created. This can happen when the storage of a value object is inlined into its surrounding list of fields. Typically, a new list of correct lengths is created and the old list will be un-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 modern JIT compilers. We expect those allocations to happen in tight loops, but more importantly, in a very restricted scope. Hence, JIT compilers with good escape analysis and allocation removal, such as meta-tracing JIT compilers [19], should be able to completely remove all allocations during the inlining process.

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

3.3. Implementing field access

While optimization of data structures takes place during construction, we have to apply the reverse during deconstruction, that is when accessing a value object referenced by another. This is no longer trivial, as several (formerly referenced) value objects may have been inlined into their referencing value objects. Therefore, we construct new value objects whenever a reference is navigated, essentially reifying it. We use the information a value object's shape provides to identify which parts of the value object's storage comprise the value object to be reified. The structural information allows a direct mapping from the language view of the data structure to the actually stored elements. In Fig. 4, the structural information in the shape of the leftmost list allow the reasoning that the first element of the storage is equivalent to the head of the language level node value object and the remaining three storage elements are equivalent to the tail of that value object, as recored in the shape. Hence the middle view in that figure; both the element "1" and the rest list have been reified. The same goes for the rightmost view.

Note that this reification is completely invisible to programmers. Taking, for example the tail of a node value object or accessing the third element of a ternary tree repeatedly, the operations remain the same on the language level, no matter what is the shape inlining status of the value objects on the implementation level.

3.4. Benefits

With the shape inlining approach, fewer value objects need to be created for long living data structures, since the references to the now-inlined value objects are elided. Combining this with the reification and the shape recognition, more memory is saved the longer a program runs; the shapes will be tailored to fit the specific application running. 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.

4. Implementation in RPython with a tracing JIT compiler

We present two implementations of our approach, both integrating the tracing JIT compiler of RPython as presented in section 2.

4.1. JIT interaction

While the techniques we described so far can lead to a good amount of memory usage reduction, shape recognition, shape inlining, and reified reference access combined, do not yield a performance increase on their own. In fact, implementing the approach naively yields significantly worse performance, due to the constant check of the transformation rules every

time a new value object is created. Additionally, reading inlined fields of compacted 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 value object creation down to a series of type checks for the types of the referenced value objects. 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 reification of referenced value objects when it is being read from a value object it has been inlined into. For that, the observation that most of these intermediate value objects are actually short-lived is crucial; most value object 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. Hence, if the tail is read off a linked list node which has the tail inlined (as the transition from left to middle in Fig. 4) and needs to be reified, that tail is usually soon deconstructed itself into its head and tail 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 reification. Since the value objects allocated when reifying a field are short-lived, the built-in escape analysis [19] will fully remove their allocation and thus remove the overhead of reification.

4.2. Best-case prototype

To assess best-case performance, we implemented our optimization approach using a simple execution model prototype.[5] It provides a λ-calculus with pattern matching as the sole control structure and is implemented as a direct application of the CEK-machine [20]. The only structured data types available are value classes. We used the RPython tool chain to incorporate its meta-tracing JIT compiler [7]. The implementation has been carefully unit-tested during development to make sure that various complex substitutions and compactions work correctly.

4.3. Structures in Racket and Pycket

Since the best-case prototype is arguably unfit for comparison with existing languages and their implementations, we applied our optimization to an implementation of the Racket language [5], a dynamically typed, multi-paradigm programming language in the Scheme family. Racket supports, among others, immutable-by-default lists, a design-by-contract [21] implementation, and heterogeneous *structure* datatypes.

The structure types are of special interest because, if applied carefully, they can be used like value classes.[6] Moreover, structures can form hierarchies and—by default—are immutable with the option to make some or all fields mutable. Racket structures go beyond other structured heterogeneous datatypes; 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.

Pycket [6] is an implementation of Racket using the RPython toolchain and its tracing JIT. While not feature-complete, it provides a fair amount of functionality and can compete with the reference implementation performance-wise, in certain areas even outperforming high-performance ahead-of-time (AOT) Scheme compilers. The support for Racket structures in Pycket is recent [22] and showed potential for the optimization presented here. Furthermore, the implementation technique (CEK machine) and environment (RPython, tracing JIT) come close to the prototype and suggest a good base for comparison.

Our approach is present in a modified Pycket implementation.[7] The existing structure implementation [22] 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 the prototype with little modification and use it as storage for all structure kinds. Only few adaptions 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.

4.4. Configuration parameters

Our approach makes use of three parameters that may influence performance:

Maximum object size Only value objects up to this size are considered for inlining. Setting this to zero disables our optimization, setting it to a very high number might result in very large value object at runtime, which might be undesirable.

[5] Available at https://bitbucket.org/krono/lamb.

[6] Structures in the Racket language actually do not by default compare based on their value and do have identity which is relied upon. However, value-based comparison can be enabled explicitly. Also, plans exist to provide a structure derivative that has a concept of identity compatible with value classes. [Sam Tobin-Hochstadt, personal communication.]

[7] https://github.com/samth/pycket/releases/tag/shapes-scp (last accessed 2015-12-15).

Maximum shape depth The number nested shape occurrences per value object is bounded by this parameter. 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 runtime should there be a lot of value objects with no fields at all.

Substitution threshold The threshold for transformation rule creation (as in section 3.1.3), when set to zero or a very low value can lead to excessive transformation rule creation for value object combinations that are only rarely used. A very high number might inhibit the creation of such rules at all and practically disables our optimization.

5. Results

We present two kinds of results. First, we show that the shape recognition part (cf. section 3.1) of our approach is feasible and can be used instead of manually specifying shape transformation rules. And second, we present the execution time and memory consumption for selected micro-benchmarks on our two implementations and three more language implementations.

5.1. Setup

Hardware The processor used was an Intel Xeon E5410 (Harpertown) clocked at 2.33 GHz with 2×6 MB cache; 16 GB of RAM were available. All runs are un-parallelized, hence the number of cores (four) was irrelevant to the experiment. Although virtualized on Xen, the machine was dedicated to the benchmarks.

Software The machine ran Ubuntu 14.04.3 LTS with a 64 bit Linux 3.13.0. *ReBench*[8] was used to carry out all execution of the benchmarks and collection of measurements. RPython as of revision 0c8d6f715aac served for translation of our prototype (tag `shapes-scp`) and the optimized Pycket (tag `shapes-scp`).

Optimization configuration During the measurements of our implementations, we used the following settings for the configuration parameters as described in section 4.4:

Maximum object size We used a maximum size of 7 fields.
Maximum shape depth We used a maximum depth of 7 shapes.
Substitution threshold We used a threshold of 17 shape occurrences.

5.2. Shape recognition fitness

To assess whether our recognition approach is favorable to manually specifying shape transformation rules, we ran several list operations on increasingly longer, large lists in our prototypical implementation in three configurations: no optimization at all (None), optimization using our approach but only using ahead-of-time, manually specified transformation rules without using shape recognition or history data (Inlining only), and optimization with transformation rules derived using shape recognition and history data (Recognition). We provide the execution time results for reversing a long list in Fig. 5. In this case, we found that

a) both optimized versions are always significantly faster than the not optimized version,
b) initially, the version with manually specified transformation rules is faster than the version with shape recognition, but
c) for most data points, the version with shape recognition and transformation rule inference is as fast as or even faster than the version with manually specified transformation rules.

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

The results suggest that the shape recognition approach could be fitting in the context of our optimization and could be favorable to specifying transformation rules manually.

5.3. Comparative micro-benchmarks

We report the performance of five micro-benchmarks with their execution time and peak memory consumption.

Compared implementations For the benchmarks, we included an unmodified Pycket ◻, Racket ┼, and PyPy ⊠[9] in the comparison. For all these, value classes or equivalent means supporting immutable data are available. The unmodified Pycket is the baseline of our implementation and does not include our optimization. Racket's *cons* cells, *structs* and classes can act

[8] ReBench is a benchmarking framework. https://github.com/smarr/ReBench.
[9] Pycket revision 291d80fbd43a; Racket version 6.3; PyPy version 4.0.1.

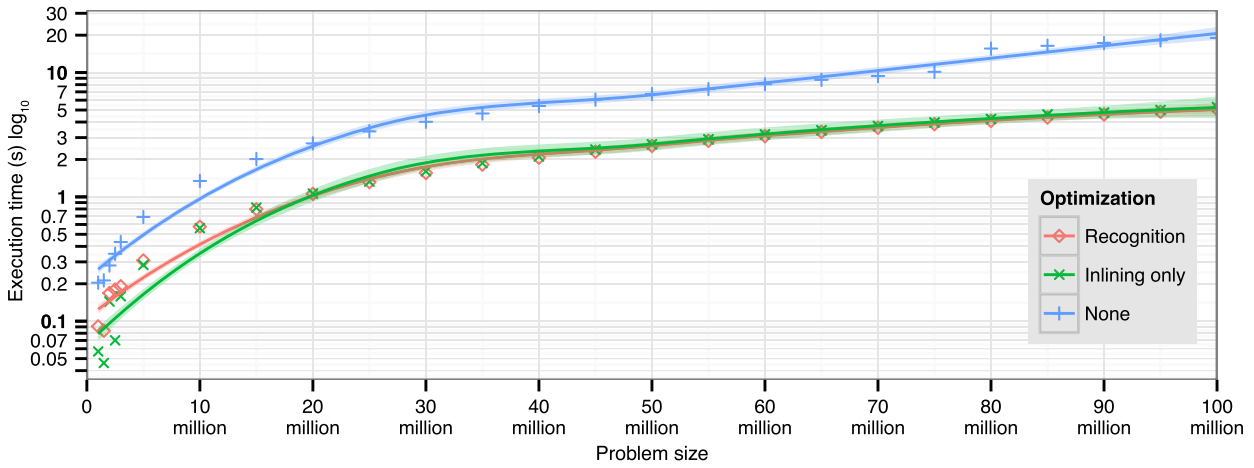


Fig. 5. Runtime results for reversing list of different lengths. *None* is without our optimization approach. *Inlining only* uses our optimization approach with ahead-of-time, manually specified transformation rules without using shape recognition. *Recognition* uses our optimization approach with transformation rules derived using shape recognition. (The data points were smoothed using local regression [23]; the semi-transparent areas are based on standard deviation of each data point. Note the logarithmic scale on the "Execution time" axis.)

Table 1
Benchmark execution times. We give means of the execution time along with the confidence interval showing the 95 % confidence level.

Benchmark	Prototype		Pycket (optimized)		Pycket (original)		Racket		PyPy	
	mean	error	mean	error	mean	error	mean	error	mean	error
append	5088	±27 ms	5545	±32 ms	9432	±52 ms	13218	±42 ms	10655	±32 ms
filter	1285	±4 ms	4743	±42 ms	5590	±87 ms	14240	±172 ms	6691	±43 ms
map	6344	±87 ms	5609	±22 ms	8332	±63 ms	15010	±161 ms	9632	±83 ms
reverse	350	±6 ms	1172	±5 ms	3347	±45 ms	6421	±159 ms	4864	±27 ms
tree	2814	±17 ms	2420	±17 ms	3926	±21 ms	3893	±27 ms	7949	±41 ms

Table 2
Benchmark memory consumption. We give means of the memory consumption along with the confidence interval showing the 95 % confidence level.

Benchmark	Prototype		Pycket (optimized)		Pycket (original)		Racket		PyPy	
	mean	error	mean	error	mean	error	mean	error	mean	error
append	1220256	±0 kB	1826172	±3 kB	3342362	±2 kB	3393476	±329 kB	4625309	±2 kB
filter	522631	±1 kB	1360879	±3 kB	2195701	±5 kB	2713104	±5883 kB	2996973	±9 kB
map	1141762	±1 kB	1600280	±1 kB	2709864	±4 kB	2518191	±9704 kB	3512922	±2 kB
reverse	192552	±1 kB	651634	±8 kB	1604697	±2 kB	1647545	±1935 kB	3512605	±5 kB
tree	71233	±1 kB	209180	±55 kB	502130	±2 kB	300500	±16 kB	956974	±4 kB

as value classes. Racket acts as a virtual machine with a handwritten JIT compiler. PyPy is the RPython implementation of Python and has a meta-tracing JIT compiler. While Python has no actual concept of value classes, we used regular classes without mutating them. PyPy detects this case and is able to apply special optimizations, effectively treating them like value classes. We intended to also include the standard Python (CPython) but it was too slow and would have rendered the comparison meaningless.

Methodology Every benchmark was run ten times uninterrupted at highest priority, in a new process. The execution time (*total 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. 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; for the execution time we include confidence intervals showing the 95 % confidence level. The memory measurements only indicate a negligible error[10] that was hence omitted. We provide all numbers for execution time in Table 1 and memory consumption in Table 2. Our benchmarking code and infrastructure are publicly available.[11]

[10] Except for Racket, which we attribute to its garbage collector (cf. Tables 1 and 2).
[11] https://bitbucket.org/krono/lamb-bench.

5.3.1. 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*. Due to the limited feature scope of our best-case prototype, more sophisticated applications are currently not available for benchmarking. For our optimization of Pycket, the structure benchmarks shipped with Racket would be interesting for our measurements. However, the structure benchmarks do not run yet on Pycket due to missing (not structure related) features [22].

5.3.2. Non-regression

Our optimization should not influence anything except value classes. To ensure this for Pycket, we ran the *shootout* benchmarks described in the original paper on Pycket [6]. These benchmarks hardly make use of structures. On average, the execution time for these benchmarks deviates less than 6 % (both faster or slower) from the original implementation. This low deviation shows that our approach has very little overhead when structs are not used.

5.3.3. Performance results

In the top part of 6, the execution time of all benchmarks is reported. Our first implementation, labeled *prototype* ○, is significantly faster—from two to ten times faster. Our second implementation, labeled *optimized Pycket* △, performs as expected. It is not as fast as the best-case prototype, as the language semantics of Racket have to be maintained as much as possible. However the speed-up compared with the unmodified, unoptimized version of Pycket is apparent. The optimized version is 1.2 to 2.9 times faster than the unoptimized version. In the case of *map* and *filter*, the optimized Pycket version is even faster than the prototype. We attribute this to the more mature status of Pycket compared with the prototype, which is a pretty direct implementation of the λ-calculus.

For memory consumption, shown in the bottom part of Fig. 6, our implementations always use significantly less memory than the other implementations. The optimized Pycket implementation is always second to our best-case prototype and in the best case uses only 40 % of the memory the unoptimized Pycket uses. The memory consumption of our best-case prototype is very low, as its execution model is quite restricted, and the only data structure types available are value classes, the subject of our optimization. On the other hand, the other language implementation face more complex execution models with more meta-data and other kinds of data structures besides value classes. Under this assumption, we think the differences between the optimized Pycket and the unoptimized Pycket are the most significant result from the memory analysis.

One key reason for our implementations' performance is the interaction between escape analysis and the compacted 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 reification of the actual value object is necessary and, furthermore, that a certain number of such operations occur consecutively. Hence, operations can happen block wise, for example for a list inlined n levels deep, reverse can operate in chunks of n items. Proper tail recursion amplifies this effect.

Given our parameters (maximum object size of 7 and maximum shape depth of 7), we expect the inlining for to result in chunks of 6 consecutive list elements. This means that (a) five class references and five 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 achieves. Moreover, the tracing JIT compiler can make assumptions on these chunk and remove almost all type checks, reduce the number of allocations to a minimum, has to follow less references, and reduce the overall number of operations the tracing JIT processes by up to 60 %.

6. Related work

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

Algebraic data types From a data structure optimization point of view, value classes are similar to *algebraic data types* as found in languages in the ML family [24,25]. Hence, optimizations done to this category of data structures are relevant to value classes, too [26,27].

Object inlining Wimmer has proposed object inlining [28] 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. This latter property is however also a weakness: it restricts the approach to statically typed programs where global analysis is possible, which hampers the use in dynamic languages and in settings where reflection or class loading is used. Additionally, the inlining decision is done per class, while in our approach different shapes and thus inlining patterns can be created for a single value class.

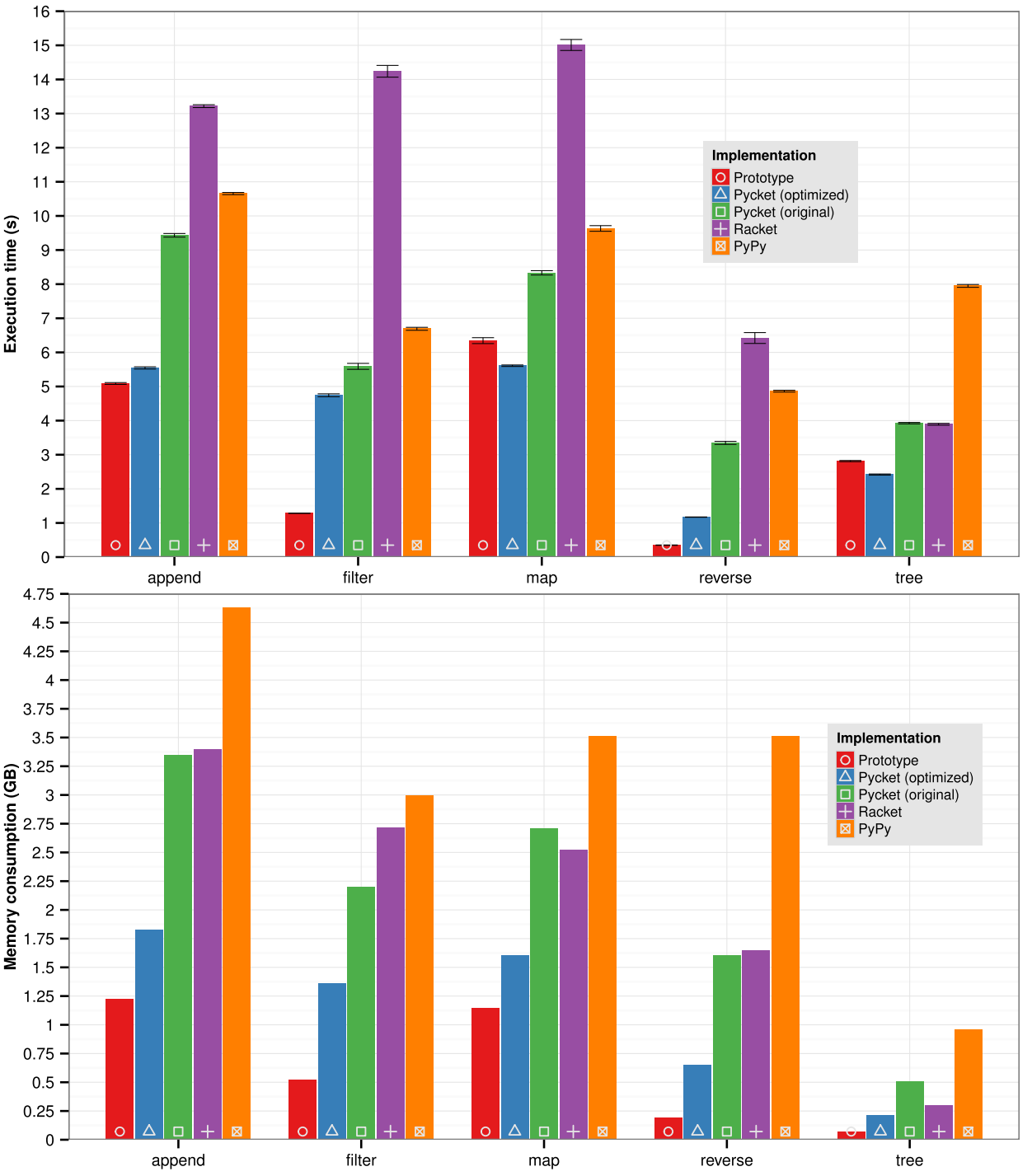


Fig. 6. Benchmarking results. Each bar shows the arithmetic mean of ten runs for execution time (top) and memory consumption (bottom). Lower is better.

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 [29]. Typically, those optimizations are restricted to linked lists of cons-cells.

One of the key effects in our optimization is avoiding to allocate intermediate data structures. In that respect, *hash consing* [30–32], 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.

Ahead-of-time optimization Deforestation [33–35] 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.

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 [8]. Prominent examples include the Smalltalk-80 bytecode-to-native-code compiler by Deutsch and Schiffman [36], and the optimizing JIT compiler of Self, with type specialization and speculative inlining [37]. These concepts were later used in the HotSpot JIT compiler [9] for Java.

The prevalence of web browsers has made JIT compilation an important topic for JavaScript implementations, for example the int V8 JavaScript implementation [10]. The map transitions for hidden classes used in V8 [38] and inspired by Self [37], 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 value object from its default shape to its most optimized one during the value object'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 [39]) 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 [12] have seen implementations for Java [13], JavaScript [40], or Lua,[12] to name a few. In the context of a JavaScript implementation, the SPUR project [41] 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 it the core idea of meta-tracing JIT compilers, pioneered in the DynamoRIO project [42]. PyPy [18,14] 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, for example Smalltalk [43], Haskell [44], PHP,[13] or R.[14] The meta-tracing JIT used in this work is provided by RPython, as well.

7. Conclusion and future work

Our approach to just-in-time optimization of value classes provides very good initial results both for execution time and memory consumption for a small prototype implementation on selected micro-benchmarks. They are promising and motivate us to investigate the matter further.

However, the current results are not yet fit for generalization. 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 languages and especially more realistic programs remains to be assessed in future work. Hence, immediate next steps include broadening the benchmarks for the Pycket-based implementation so that we can assess the viability of our approach in more representative context.

Racket supports more datatypes that may be subject to our approach, for example (immutable) *cons* cells. We plan to integrate these with our approach.

Our aim is then to broaden the scope of our approach beyond value classes. We want to support objects that have identity as well as mutable objects. While the usage of a cell indirection in the Pycket implementation has proven worthwhile to allow mutability, we do not yet know whether this approach of quasi-immutability is portable to other languages. Even more, maintaining identity, and hence object-oriented concepts, needs more in-depth investigation.

Acknowledgements

We gratefully acknowledge the financial support of HPI's Research School and the Hasso Plattner Design Thinking Research Program (HPDTRP). Carl Friedrich Bolz is supported by the EPSRC *Cooler* grant EP/K01790X/1. We thank Alan Borning for comments on a draft version of this paper. We thank the anonymous reviewers for their detailed feedback.

References

[1] D.F. Bacon, Kava: a Java dialect with a uniform object model for lightweight classes, Concurr. Comput., Pract. Exp. 15 (3–5) (2003) 185–206, http://dx.doi.org/10.1002/cpe.653.

[2] J. Rose, JEP 169: value objects, http://openjdk.java.net/jeps/169, 2014 (visited on 2014-09-15).

[12] http://luajit.org.

[13] http://hippyvm.com/.

[14] https://bitbucket.org/roy_andrew/rapydo.

[3] Microsoft Developer Network, Common type system, http://msdn.microsoft.com/en-us/library/zcx1eb1e(d=default,l=en-us,v=vs.110).aspx, 2014 (visited on 2014-09-15).
[4] M. Odersky, P. Alther, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, An overview of the Scala programming language, Tech. rep. LAP-REPORT-2006-0001, EFPL, Lausanne, Switzerland, 2006.
[5] M. Flatt, PLT, Reference: Racket, Tech. rep. PLT-TR-2010-1, PLT Inc., 2010, http://racket-lang.org/tr1/.
[6] S. Bauman, C.F. Bolz, R. Hirschfeld, V. Krilichev, T. Pape, J. Siek, S. Tobin-Hochstadt, Pycket: a tracing JIT for a functional language, in: Proc. of ICFP 2015, ICFP '15, Vancouver, BC, Canada, ACM, 2015.
[7] C.F. Bolz, Meta-tracing just-in-time compilation for RPython, Ph.D. thesis, Mathematisch-Naturwissenschaftliche Fakultät, Heinrich Heine Universität Düsseldorf, 2012.
[8] J. Aycock, A brief history of just-in-time, ACM Comput. Surv. 35 (2) (2003) 97–113, http://dx.doi.org/10.1145/857076.857077.
[9] M. Paleczny, C.A. Vick, C. Click, The Java HotSpot server compiler, in: Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium, vol. 1, JVM '01, Monterey, CA, USENIX Association, 2001.
[10] M. Hölttä, Crankshafting from the ground up, Tech. rep., Google, 2013.
[11] V. Bala, E. Duesterwald, S. Banerjia, Dynamo: a transparent dynamic optimization system, ACM SIGPLAN Not. 35 (5) (2000) 1–12.
[12] J.G. Mitchell, The design and construction of flexible and efficient interactive programming systems, Ph.D thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970.
[13] A. Gal, C.V. Probst, M. Franz, HotpathVM: an effective JIT compiler for resource-constrained devices, in: Proceedings of the 2nd International Conference on Virtual Execution Environments, VEE '06, Ottawa, ON, Canada, ACM, 2006, pp. 144–153.
[14] C.F. Bolz, A. Cuni, M. Fijalkowski, A. 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, ICOOOLPS '09, Genova, Italy, ACM, 2009, pp. 18–25.
[15] C.F. Bolz, L. Tratt, The impact of meta-tracing on VM design and implementation, Sci. Comput. Program. (2013), http://dx.doi.org/10.1016/j.scico.2013.02.001.
[16] C.F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, A. Rigo, Runtime feedback in a meta-tracing JIT for efficient dynamic languages, in: Proc. ICOOOLPS, 2011, pp. 9:1–9:8.
[17] C.F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, A. Rigo, Allocation removal by partial evaluation in a tracing JIT, in: Proc. PEPM, 2011, pp. 43–52.
[18] A. Rigo, S. 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, OR, USA, ACM, 2006, pp. 944–953.
[19] C.F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, A. 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, TX, USA, ACM, 2011, pp. 43–52.
[20] M. Felleisen, D.P. Friedman, Control operators, the SECD-machine and the λ-calculus, in: M. Wirsing (Ed.), Proceedings of the 2nd Working Conference on Formal Description of Programming Concepts, vol. III, Elsevier, 1987, pp. 193–217.
[21] R. Mitchell, J. McKim, B. Meyer, Design by Contract, by Example, Addison–Wesley, 2001.
[22] T. Pape, V. Kirilichev, R. Hirschfeld, Optimizing record data structures in Racket, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, Pisa, Italy, ACM, 2016, pp. 1798–1805.
[23] W.S. Cleveland, E. Grosse, W.M. Shyu, Local regression models, in: J.M. Chambers, T.J. Hastie (Eds.), Statistical Models in S, Wadsworth & Brooks/Cole, Pacific Grove, CA, 1992, pp. 309–376.
[24] R. Milner, M. Tofte, R. Harper, D. MacQueen, The Definition of Standard ML, revised edition, MIT Press, 1997.
[25] L. Damas, R. Milner, Principal type-schemes for functional programs, in: Proceedings of the 9th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, POPL '82, Albuquerque, NM, ACM, 1982, pp. 207–212.
[26] P. Lee, M. 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, PA, USA, ACM, 1996, pp. 137–148.
[27] X. Leroy, The ZINC experiment: an economical implementation of the ML language, Technical report 117, INRIA, 1990.
[28] C. Wimmer, Automatic object inlining in a Java virtual machine, Ph.D. thesis, Johannes Kepler Universität, Linz, Austria, 2008.
[29] Z. Shao, J.H. Reppy, A.W. Appel, Unrolling lists, SIGPLAN Lisp Pointers VII (3) (1994) 185–195, http://dx.doi.org/10.1145/182590.182453.
[30] A.P. Ershov, On programming of arithmetic operations, Commun. ACM 1 (8) (1958) 3–6, http://dx.doi.org/10.1145/368892.368907.
[31] E. Goto, Monocopy and associative algorithms in extended Lisp, Technical report TR-74-03, University of Tokyo, Japan, 1974.
[32] J.-C. Filliâtre, S. Conchon, Type-safe modular hash-consing, in: Proceedings of the 2006 Workshop on ML, ML '06, Portland, OR, USA, ACM, 2006, pp. 12–19.
[33] P. Wadler, Deforestation: transforming programs to eliminate trees, Theor. Comput. Sci. 73 (2) (1990) 231–248, http://dx.doi.org/10.1016/0304-3975(90)90147-A.
[34] A. Gill, J. Launchbury, S.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, pp. 223–232.
[35] A. Takano, E. Meijer, Shortcut deforestation in calculational form, in: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA '95, La Jolla, CA, USA, ACM, 1995, pp. 306–313.
[36] L.P. Deutsch, A.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, UT, USA, ACM, 1984, pp. 297–302.
[37] C. Chambers, D. Ungar, E. Lee, An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes, SIGPLAN Not. 24 (10) (1989) 49–70, http://dx.doi.org/10.1145/74878.74884.
[38] Google, Inc., Chrome V8 documentation: design elements, https://developers.google.com/v8/design, 2012 (visited on 2014-09-11).
[39] M. Chevalier-Boisvert, M. Feeley, Extending basic block versioning with typed object shapes, arXiv:1507.02437 [cs.PL], 2015.
[40] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M.R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E.W. Smith, R. Reitmaier, M. Bebenita, M. Chang, M. Franz, Trace-based just-in-time type specialization for dynamic languages, SIGPLAN Not. 44 (6) (2009) 465–478, http://dx.doi.org/10.1145/1543135.1542528.
[41] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, H. Venter, SPUR: a trace-based JIT compiler for CIL, SIGPLAN Not. 45 (10) (2010) 708–725, http://dx.doi.org/10.1145/1932682.1869517.
[42] G.T. Sullivan, D.L. Bruening, I. Baron, T. Garnett, S. Amarasinghe, Dynamic native optimization of interpreters, in: Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators, IVME '03, San Diego, CA, ACM, 2003, pp. 50–57.
[43] C.F. Bolz, A. Kuhn, A. Lienhard, N.D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, T. Verwaest, Back to the future in one week—implementing a Smalltalk VM in PyPy, in: Self-Sustaining Systems, in: Lecture Notes in Computer Science, vol. 5146, Springer, Berlin–Heidelberg, 2008, pp. 123–139.
[44] E.W. Thomassen, Trace-based just-in-time compiler for Haskell with RPython, Master's thesis, Norwegian University of Science, Technology Trondheim, 2013.