Science of Computer Programming ••• (••••) •••–•••

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 xxxx

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.

Please cite this article in press as: T. Pape, C.F. Bolz, and R. Hirschfeld, Adaptive just-in-time value class optimization for lowering memory consumption and improving execution time performance, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.08.003