Reshape Your Layouts, Not Your Programs: A Safe Language Extension for Better Cache Locality

Alexandros Tasos
Imperial College London, United Kingdom
at1917@ic.ac.uk

Juliana Franco
Microsoft Research, London, United Kingdom
juliana.franco@microsoft.com

Sophia Drossopoulou
Imperial College London, United Kingdom
Microsoft Research, London, United Kingdom
scd@doc.ic.ac.uk

Tobias Wrigstad
Uppsala University, Sweden
tobias.wrigstad@it.uu.se

Susan Eisenbach
Imperial College London, United Kingdom
sue@doc.ic.ac.uk

Abstract

The vast gap between CPU and RAM speed means that on modern architectures, developers need to carefully consider data placement in memory to exploit spatial and temporal cache locality and use CPU caches effectively. To that extent, developers have devised various strategies regarding data placement; for objects that should be close in memory, a contiguous *pool* of objects is allocated and then new instances are constructed inside it; an array of objects is *clustered* into multiple arrays, each holding the values of a specific field of the objects[1]. Such data placements, however, have to be performed manually, hence readability, maintainability, memory safety, and key OO concepts such as encapsulation and object identity need to be sacrificed and the business logic needs to be modified accordingly.

We propose a language extension, SHAPES, which aims to offer developers high-level fine-grained control over data placement, whilst retaining memory safety and the look-and-feel of OO. SHAPES extends an OO language with the concepts of *pools* and *layouts*: Developers declare *pools* that contain objects of a specific type and specify the pool's *layout.* A *layout* specifies how objects in a pool are laid out in memory. That is, it dictates how the values of the fields of the pool's objects are grouped together into *clusters.* Objects stored in pools behave identically to ordinary, standalone objects; the type system allows the code to be oblivious to the layout being used. This means that the business logic is completely decoupled from any placement concerns and the developer need not deviate from the spirit of OO to better utilise the cache.

In this paper, we present the features of SHAPES, as well as the design rationale behind each feature. We then showcase the merit of SHAPES through a sequence of case studies; we claim that, compared to the manual pooling and clustering of objects, we can observe improvement in readability and maintainability, and comparable (i.e., on par or better) performance.

We also present SHAPES^h, an OO calculus which models the SHAPES ideas, we formalise the type system, and prove soundness. The SHAPES^h type system uses ideas from Ownership Types [1] and Java Generics [2]: In SHAPES^h, pools are part of the types; SHAPES^h class and type definitions are enriched with pool parameters. Moreover, class pool parameters are enriched with bounds, which

[1] Commonly referred to as an *Array-of-Structs* (AoS) to *Struct-of-Arrays* (SoA) transformation.

© Alexandros Tasos, Juliana Franco, Sophia Drossopoulou, Tobias Wrigstad, and Susan Eisenbach;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).
Editors: Robert Hirschfeld and Tobias Pape; Article No. 31; pp. 31:1–31:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
class Professor<pProf: [Professor<pProf>]> {
    name: String;   ssn: String;
}
class Student<pStu: [Student<pStu, pProf>], pProf: [Professor<pProf>]> {
    name: String;   age: int;   supervisor: Professor<pProf>;
}
layout ProfL: Professor = rec{name} + rec{ssn};
layout StuL: Student = rec{name, age} + rec{supervisor};
...
pools pStu1: StuL<pStu1, pProf1>, pProf1: ProfL<pProf1>;
stu = new Student<pStu1, pProf1>;
prof = new Professor<pProf1>;
stu.supervisor = prof;
```

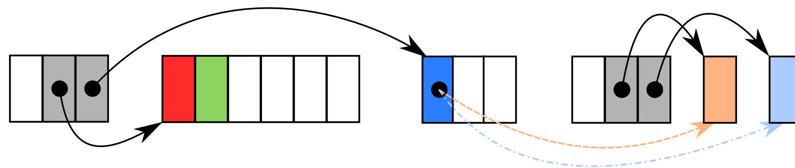

Figure 1 Example SHAPES code and memory layout.

is what allows the business logic of SHAPES to be oblivious to the layout being used. SHAPES^h types also enforce pool uniformity and homogeneity. A pool is uniform if it contains objects of the same class only; a pool is homogeneous if the corresponding fields of all its objects point to objects in the same pool. These properties allow for more efficient implementation.

For performance considerations, we also designed SHAPES^l, an untyped, unsafe low-level language with no explicit support for objects or pools. We argue that it is possible to translate SHAPES^l into existing low-level intermediate representations, such as LLVM [3], present the translation of SHAPES^h into SHAPES^l, and show its soundness.

Thus, we expect SHAPES to offer developers more fine-grained control over data placement, without sacrificing memory safety or the OO look-and-feel.

2012 ACM Subject Classification Software and its engineering → Classes and objects; Theory of computation → Formalisms; General and reference → Performance

Keywords and phrases Cache utilisation, Data representation, Memory safety

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.31

Category SCICO Journal-first

Related Version Full article available at `https://doi.org/10.1016/j.scico.2020.102481`.

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.2.19`.

Funding *Alexandros Tasos*: Supported by an EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS) Grant (Reference EP/L016796/1).

References

1 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. *Ownership Types: A Survey*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_3.

2 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 8 Edition (Java Series), 2014.

3 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.