You Can Have It All

Abstraction and Good Cache Performance

Juliana Franco
Imperial College London
United Kingdom
j.vicente-franco@imperial.ac.uk

Martin Hagelin*
Dirac
Sweden

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

Sophia Drossopoulou
Imperial College London
United Kingdom
s.drossopoulou@imperial.ac.uk

Susan Eisenbach
Imperial College London
United Kingdom
s.eisenbach@imperial.ac.uk

Abstract

On current architectures, the optimisation of an application's performance often involves data being stored according to access affinity — what is accessed together should be stored together, rather than logical affinity — what belongs together logically stays together. Such low level techniques lead to faster, but more error prone code, and end up tangling the program's logic with low-level data layout details.

Our vision, which we call SHAPES— Safe, High-level, Abstractions for oPtimisation of mEmory cacheS — is that the layout of a data structure should be defined only once, upon instantiation, and the remainder of the code should be layout agnostic. This enables performance improvements while also guaranteeing memory safety, and supports the separation of program logic from low level concerns. In this paper we investigate how this vision can be supported by extending a programming language.

We describe the core language features supporting this vision: classes can be customized to support different layouts, and layout information is carried around in types; the remaining source code is layout-unaware and the compiler emits layout-aware code. We then discuss our SHAPES implementation through a prototype library, which we also used for preliminary evaluations. Finally, we discuss how the core could be expanded so as to deliver SHAPES's full potential: the incorporation of compacting garbage collection, ad hoc polymorphism and late binding, synchronization of representations of different collections, support for dynamic change of representation, etc.

CCS Concepts • **Software and its engineering → Secondary storage**; **Object oriented languages**; **Data types and structures**; *Extra-functional properties*;

Keywords object layout

ACM Reference Format:
Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. 2017. You Can Have It All: Abstraction and Good Cache Performance. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3133850.3133861

1 Introduction

When the speed of memory accesses rivalled that of the CPU, the perception that memory accesses are "for free" was a valid one. However, today, CPU speeds greatly exceed memory bandwidths on most platforms, to the point where computation is almost for free, and the real cost of execution, both in terms of speed and power consumption, is in accessing memory (*c.f.* the memory wall [Wulf and McKee 1995]).

For decades, to hide this latency, cache memories or hierarchies of cache memories have been part of computer architectures, exploiting the temporal and spatial locality inherent in most programs. To improve a program's performance, programmers must minimise cache misses by understanding "what goes into cache" when data is loaded from memory. Writing programs that leverage cache effects requires the programmer to keep a mental image of the high-level program logic as well as the low level memory concerns, and is at odds with mainstream programming abstractions.

For concreteness, the table below shows access times and sizes for the machine we used in our evaluation on which we report in Appendix A — an Intel i7-3770 (Ivy Bridge) [Ivy-Bridge 2016]. Accessing RAM is 42× slower than accessing the fastest cache.

*Work done while at Uppsala University.

Onward!'17, October 25–27, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5530-8/17/10…$15.00
https://doi.org/10.1145/3133850.3133861

		Slowdown		
	Access Time	Absolute	Relative	Size
L1	5 cycles	–	–	32 Kb
L2	12 cycles	×2,4	×2,4	256 Kb
L3	30 cycles	×6	×2,5	8 Mb
RAM	210 cycles	×42	×7	16 GB

When a value is loaded from main memory, it, and some surrounding values (a *cache line*), are copied to the cache. This makes subsequent accesses to *any of these values* significantly faster. Loading a value is a *cache hit* if the value is in cache and a *cache miss*, if it is not. Furthermore, if the hardware detects a pattern in the addresses loaded (*e.g.*, 10, 26, 42, 58…), it will speculatively load — *prefetch* — data into cache in time for subsequent access. Such patterns arise easily when iterating over arrays, but not necessarily with pointer-based data structures.

Figure 1 shows how access time is affected by fragmentation, which impacts both how much of a cache line is useful and whether access patterns arise. The program creates a linked list of 10^7 nodes in allocation order but with some probability of "garbage" allocated between each node. It then iterates over the list, accessing 4 adjacent fields in each node. We ran this program and measured the iteration times for varying size of of objects (*i.e.* amount of "garbage" per node).

As expected, more fragmentation means slower iteration. Moreover, the larger the part of the node not used, the slower the iteration, as cache utilisation drops. So objects accessed consecutively should be allocated contiguously and fields of the same object that are not used together should not be adjacent in memory.

Like on our test machine, caches are typically much smaller than many data structures in programs. This means that to get good cache utilisation, a programmer has to carefully configure data structures in a way that ignores the ideas about data abstraction developed over the last 50 years and may even require a rewrite to move to a machine with a different architecture.

A programmer optimising for memory performance will find herself using arrays for data structures – clearly not the appropriate abstraction for problems that require complex linked structures. Also, in some managed languages such as Java or Scala, arrays are *arrays of pointers* (*i.e.* they store pointers to values or objects) rather than *arrays of values* (*i.e.* they store the actual values or objects), and therefore arrays do not guarantee good cache behaviour. *Splitting* an object into primitive constituents (if possible) overcomes this, at the cost of convoluted code (as we shall soon see).

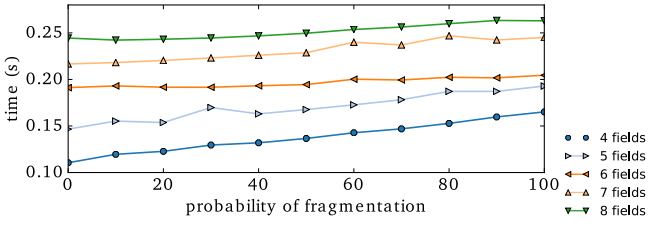


Figure 1. Impact of fragmentation on iteration over objects of different sizes. Smaller is better.

Even with support for arrays storing objects, splitting is useful to address cache pollution — fetching unused data into cache. Programmers commonly transform arrays of structs (objects) into structs of arrays, deconstructing the objects of an array into smaller parts, stored separately but used together. This optimisation brings even more complexity than using arrays to hold entire objects, and is good for iteration but bad for random access.

1.1 The SHAPES Concept

SHAPES has been designed as an extension for high-level, imperative, object-oriented programming languages and aims to facilitate the development of cache friendly code, while keeping important high-level properties, such as:

- type safety and memory safety,
- separation of layout and program logic,
- usual reasoning about pointer-based data structures,
- code reusability, and
- notions of object and object identity.

The goal of SHAPES is to make it possible to write a data structure once, and then tune it — by changing how objects are placed and laid out internally — to fit different usage scenarios and deployment onto different hardware. Changing from one layout to another should be as simple as changing a layout specification, written separately from the algorithm, and then a quick recompile, delegating to the compiler the hard work of updating all accesses to the objects correctly.

In order to achieve this, SHAPES extends the host language with *pools*, *layouts*, and class parameterization. Pools are contiguous memory regions into which objects can be grouped. Layouts describe how objects are organized in these pools, and in particular whether and how they are split. Classes take pool parameters, which determine how their instances will be allocated to pools. Moreover, SHAPES relies on the existence of a runtime that supports pools as described above, and compacting garbage collection, which will be used to reduce fragmentation of pools, often caused by dropping references to objects.

Contributions In this paper we outline a core SHAPES language, we describe a prototype library which supports the SHAPES features, we demonstrate the benefits of the use of the SHAPES library in terms of some short programs, and finally we discuss how SHAPES would fit into a full programming language, and how the original SHAPES remit can be expanded.

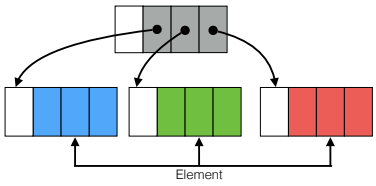
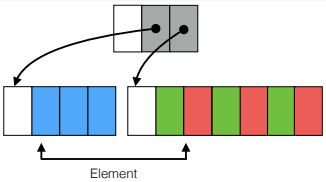
2 Core SHAPES

Managed languages abstract memory away, voiding the need for programmers to worry about allocation and reclamation,

Table 1. Different layouts for the same data structure. Code and representation in memory. The `init` method of each class represents the class constructor. White rectangles represent meta-data and coloured rectangles represent the "objects" values.

Array of Structs (AoS)	**Struct of Arrays (SoA)**	**Struct of Arrays of Structs (SoAoS)**

`class Element`
`f1: int`
`f2: int`
`f3: bool`
 	
`var aos = new Element[N]`
	`class Elements`
	`f1: int[]`
	`f2: int[]`
	`f3: bool[]`
	
	`var soa = new Elements(N)`
		`class Elements`
		`f1: int[]`
		`f2f3: SubElement[]`
		
		`class SubElement`
		`f2: int`
		`f3: bool`
		
		`var soaos = new Elements(N)`

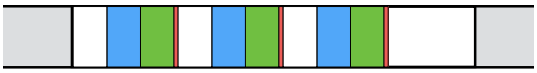
`def init(x: int, y: int,`
`  z: bool): void`
`  this.f1 = x`
`  this.f2 = y`
`  this.f3 = z`
	`def init(N: int): void`
	`  soa.f1 = new int[N]`
	`  soa.f2 = new int[N]`
	`  soa.f3 = new bool[N]`
	`// requires f1!=null && f2!=null`
	`  // && f3!=null`
	`def initElem(i: int, x: int, y: int,`
	`  z: int): void`
	`  this.f1[i] = x`
	`  this.f2[i] = y`
	`  this.f3[i] = z`
		`class Elements`
		`def init(N: int): void`
		`  this.f1 = new int[N]`
		`  this.f2f3 = new SubElement[N]`
		`// requires f1 != null && f2f3 != null`
		`def initElem(i: int, x: int, y: int,`
		`  z: int): void`
		`  this.f1[i] = x`
		`  this.f2f3[i].f2 = y`
		`  this.f2f3[i].f3 = z`

`def m(e: Element): void`
`  e.f3 = true`
`def main(): void`
`  var aos = new Element[N]`
`  for i ←[0 .. N] do`
`  aos[i] =`
`  new Element(i, 0, false)`
`  for i ←[0 .. N] do`
`  m(aos[i])`
	`def m(es: Elements, i: int): void`
	`  es.f3[i] = true`
	`def main(): void`
	`  var soa = new Elements(N)`
	`  for i ←[0 .. N] do`
	`  soa.initElem(i, i, 0, false)`
	`  for i ←[0 .. N] do`
	`  m(soa, i)`
		`def m(es: Elements, i: int): void`
		`  es.f2f3[i].f3 = true`
		`def main(): void`
		`  var soaos = new Elements(N)`
		`  for i ←[0 .. N] do`
		`  soaos.initElem(i, i, 0, false)`
		`  for i ←[0 .. N] do`
		`  m(soaos, i)`

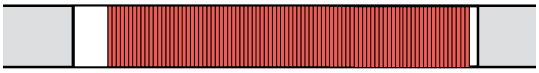
and excluding errors like double-free and dangling pointers. The downside of this abstraction is that a programmer has less (or no) control over how a program's data is placed in memory. In unmanaged languages like C and C++, a programmer can `malloc` a chunk of memory, allocate several objects inside that space, and access them either using pointers, or as if accessing elements of an array. This type of flexibility enables layout optimisations, but at the cost of more brittle software (*e.g.* memory unsafety) and higher maintenance costs. When writing code in high-level languages, the programmer often has no other option than using arrays of values to control object layout.

In this Section, we discuss the advantages of a good memory layout, we survey some of the difficulties in writing code that exploits cache locality with arrays, and explain how we address them in SHAPES. We use a small running example of a collection of objects (`Elements`). Table 1 shows three possible layouts for this data structure using arrays of values to ensure that all `Elements` are allocated contiguously. This program creates an array of `Elements` and iterates over it twice. First, initialising all the `Elements`, and then calling some method `m` on each object. The allocation of `Elements` in an array of objects will guarantee that they will be contiguous in memory. As discussed before, this is advantageous because when reading the first `Element` (`aos[0]`), more adjacent `Elements` will be fetched to cache.

Figure 2a shows a cache line being fetched to cache, when the processor first touches the `aos` data of Table 1, invoking the `m` method. We assume an architecture with cache lines of 64 bytes, as in the machine we used for our experiments,



(a) Contents of cache line, upon iteration on `aos`, when `i=1`



(b) Contents of cache line, upon iteration on `soa`, when `i=1`.

Figure 2. Cache lines for different layouts.

and we depict which data is fetched to cache. Note that we expect an `int` to fit in 8 bytes and a `bool` to fit in 1 (ignoring padding for simplicity). The objects in `aos[0]`, `aos[1]` and `aos[2]` will be fetched. Thus, while the first loop iteration is expensive, due to a cache miss, the next two loop iterations are almost for free, due to cache hits. For a large number of objects, `N`, this will bring performance improvements.

Even more advantageous for this code in particular, is the layout in the second column of Table 1. Once the processor touches the array f_3 for the first time — and note this is the only one needed for this part of the program — it can fetch many more boolean values. It fetches only the "f_3 fields", resulting in many more cache hits in the next loop iterations.

2.1 Why Arrays Are Insufficient

Absence of Value Semantics Many high-level languages, such as Java or Scala, do not support arrays with value semantics, *i.e.*, arrays contain *pointers* to objects, rather than the objects themselves. This means that none of the layouts depicted in Table 1 are possible, and that the programmer has no control on data placement — the location of objects in memory will depend on the allocator, or on the garbage collector.[1]

Unsafe Splitting Once an object is split into more than one array, it becomes possible to inadvertently reassemble the wrong parts of an object, thus creating an object out-of-thin-air. For example, the creation of a `Frankenstein` object is created by reassembling: `aos.`f_1`[0]`, `aos.`f_2`[0]` and `aos.`f_3 `[1]`.

Pointer-Based Data Structures Mapped to Arrays Some of the most common data structures, *e.g.* `Lists`, `Trees`, and `Stacks` rely on the existence of pointers. Mapping such data structures to arrays is non-trivial [Vollmer et al. 2017], given that a pointer to an object is the array(s) in which the object is allocated, and an offset. Even in low-level languages, such mappings require some non-trivial pointer management. Consider, for example, the SoA data structure from Table 1. Adding a fourth field with a pointer to another `Element` is tricky if such an element is also split over several different arrays. Additional complications arise when reorganising objects in data structures (*e.g.* sorting, inserting or deleting objects). Reordering objects in an array is also complex if they can only be accessed through the array (using an index).

Cost of Growing an Array Holding objects in an array for locality raises the issue of capacity. When an array reaches its maximum capacity, it must be re-sized — typically, by creating a new and bigger array, and copying all the data from an array to the other — an $O(n)$ operation on n elements, especially if it involves additional indirection inside of the objects.

Element Objects with Different Lifetimes Objects belonging to the same data structure commonly have different lifetimes. Arrays holding unreachable objects cause fragmentation of memory, unless the programmer manually shifts all the data once each object becomes unreachable. Fragmentation causes cache misses, and manual memory management is known to be error prone and unsafe [Jones et al. 2016].

SHAPES We strive to have it all: keep memory abstract (*e.g.* no need to worry about how many bytes to allocate, or when and where to deallocate), without giving up on layout control — simple development and efficient code. We believe in the run-time doing most of the hard work, such as providing memory management that keeps relevant parts of related objects allocated contiguously in memory without expensive copying of elements and without compromising memory safety. Programmers can keep a pointer-based abstraction, but project it differently onto array-like structures for different usage scenarios, with minimal effort — without changing the logic of the program. Compaction and reordering must be supported for locality and low fragmentation. To this end, we introduce a notion of *pools* which are abstract contiguous regions in which objects can be grouped.

Pools are chunks of memory that allow contiguous allocation of objects. The rationale behind pools is that objects with high access affinity (frequently used together) should be placed together for better cache utilisation. In prior work, pooling separated objects in separate memory pools depending on their type, time or place of allocation, so that objects with high access affinity are placed together for better cache utilisation [Calder et al. 1998; Lattner and Adve 2003]. We do not intend for SHAPES to use any complex heuristics on which objects go into each pool. Instead, we require that pools hold objects of the same type, and we use the knowledge of the programmer to decide for each object which pool it should be in. Later, we will discuss how we can provide memory safety, and prevent instances of the same class but with different layout and placement from being confused.

A SHAPES pool is reminiscent of an array of objects (not pointers), but also supports object splitting as well as garbage collection, both to reorder its contents (*e.g.* to align it with access patterns) and reduce fragmentation. Figure 3 shows

[1] A moving collector *might* improve locality — or make it worse.

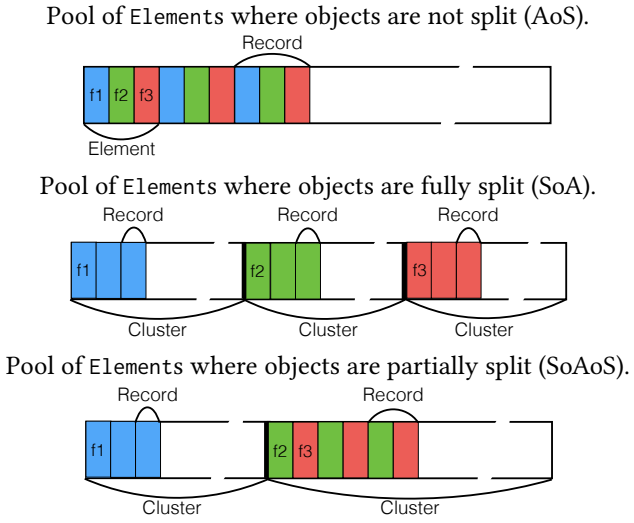
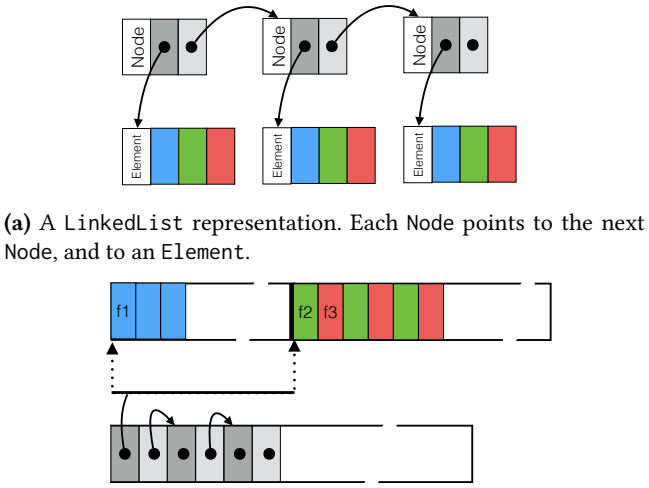


Figure 3. The layouts from Table 1 transformed into pools.



(a) A `LinkedList` representation. Each `Node` points to the next `Node`, and to an `Element`.

(b) A `LinkedList` of `Nodes`, allocated in a pool of `Nodes` without any splitting, that point to `Elements`, allocated in a pool of `Elements` split into two clusters.

Figure 4. Different layouts for a `LinkedList`.

three different pools, with three different splitting strategies: AoS, SoA, and SoAoS from Table 1.

A SHAPES pool consists of *clusters*, which contain sequences of *records*. Records contain sequences of values. A cluster contains the same data as the arrays of Table 1: the pool in the AoS column contains a single cluster with all the `Elements` allocated contiguously; the pool in the SoA column contains three clusters, one for each field, one for each array; and the pool in the SoAoS column contains two clusters, one for each array, where the first cluster contains all the values pointed by field f_1, and the second cluster all the values pointed by fields f_2 and f_3. A record represents a part of, or an entire object. For example, an object that is not split, as in the first pool, is a record consisting of fields f_1, f_2, and f_3. In the third pool, an object is split into two and thus consists of two records: one in the first cluster, containing the fields f_1, and another in the second cluster, containing the fields f_2 and f_3. An important property that we want to ensure is that records are aligned, that is, parts of the same object must have the same *offset* within their clusters.

The pooling mechanism brings a number of advantages, which help us deal with the problems mentioned above. Namely, an object is split within the same pool, rather than in multiple arrays. This property combined with record alignment allows us to have references to pooled objects such that a single value allows calculating all field locations. A reference contains the address of the pool and the offset of the object N — the different records of the object can be found in the offset N of all the pool clusters. This means that the exact location of a field of a given object at offset N can always be obtained through simple pointer arithmetic. Naturally, these references will not be visible to the programmer of the high-level language, in the same way that a memory pointer in not visible to a Scala/Java programmer.

Being able to reference a split, pooled object without additional indirection simplifies the mapping of pointer-based data structures onto arrays, such as the one in Figure 4a, to pools, such as in Figure 4b. From the programmer's view, object allocation can take place "somewhere in memory" (like normal allocation in *e.g.* Java), or in a pool. In our example, we allocate `Nodes` and `Elements` in a pool; and rather than using a "regular" pointer (as in offsets from start of global memory) from `Nodes` to `Elements`, we use a pooling–splitting aware SHAPES reference, so that splitting is taken into account. Later we will show how SHAPES can be used in a high-level language to implement such a data structure.

Pools allow pointers and aliases to point to individual pooled objects, in contrast to the way most languages allocate objects in arrays. For example, in Rust one can allocate several objects in an array with value semantics, but accessing these objects through a mutable reference is rather complex: While multiple references to immutable structures are possible, for a mutable reference, one needs to "borrow" a reference from the array, making the array temporarily inaccessible. Moreover, while arrays are typically of fixed size, pools can grow dynamically, allowing addition in constant time. We explain how we achieve this in Section 3.

Pools are more high-level, powerful and flexible than arrays with value semantics (arrays of objects). To address reordering of objects in data-structures, changing of access order in the program, and because of problems with fragmentation, we will use a moving garbage collector, which compacts and reorders pools. This is discussed in Section 4.1.

2.2 Layout and Program Logic Are Entangled

Whether writing code in a high-level or low-level language, if the programmer needs to manually change the layout of data in memory, then she needs to change the code that uses (reads and writes) this data. This is visible in Table 1, when converting an AoS into a Soa and a SoAoS. It changes the type of the class fields, and the logic of the code that uses them, as we can see in the `init` and `m` methods. Moreover, in order to split an object of three fields into two records, one needs to write another class (`SubElement`). This makes programming complicated, as the usage of a data structure can be spread over thousands of lines of code. This also leads to repeated code, if one needs to use multiple instances of the same data structure with different layouts. Since layout optimisation can be platform dependent (*e.g.* varying cache configurations), many versions may need to be maintained differing slightly on layout but with the same program logic. Because of layout differences, one bug-fixing patch may not be applicable across versions.

To overcome this issue, SHAPES uses classes parametric with pools, pools belonging to layouts, and layout declarations describing object splitting. The idea is that decisions regarding layout are made at class instantiation, and not class declaration. Thus, the data structure's code is oblivious to layout and location. For example, the class `Element` can be annotated as follows:

```
class Element⟨p: [Element]⟩
  f1: int
  f2: int
  f3: bool
  def init(x: int, y: int, z: bool): void
    this.f1 = x; this.f2 = y; this.f3 = z
```

Classes are parameterized over pools. The first parameter binds the location of instances of the class. The remaining parameters bind other locations of objects pointed by instances of the class. For example, the class `Element` takes a single parameter, denoting the location of its instances, annotated as a pool holding `Elements` (`[Element]`). A location is either a pool, or the "mixed heap" (like a normal Java heap). The annotation of the first parameter is optional, as it is the same as the class itself. An example of a class that takes multiple parameters is as follows.

```
class Container⟨loc, elemLoc: [Element]⟩
  elem: Element⟨elemLoc⟩
  def setField(n: int): void
    this.elem.f1 = n
```

Each container instance will reside in a location it internally refers to as `loc`, and will point to `Elements` allocated in a location bound by `elemLoc`. More examples will be given later, but note that this is similar to how object ownership is propagated in ownership type systems [Clarke et al. 1998].

In contrast to ownership contexts, pools do dot enforce a hierarchical decomposition of the object graph.

The `Element` class is oblivious as to whether its instances will be allocated in pools, or not, and as to which (if any) splitting will be used. Thus, its reads and writes to its fields are unaware of layout decisions. This separation between layout concerns and the program logic allows the programmer to use this class declaration and change its layout as many times as needed without changing the code of its methods — the `init` method remains exactly the same.

When instantiating an object, the programmer must specify where it shall be allocated. Naturally, the location of objects is reflected in their types. Object types consist of a class name followed by a sequence of pool arguments, where the first argument denotes the pool where the object will be allocated, and the remaining, the pools that object may reach. Objects can be allocated anywhere in memory, using the keyword `heap`, or in pools, using the name of a particular pool. This means that objects may only be created in pools that were previously created, according to some splitting strategy. SHAPES requires the host language to be extended with layout declarations, which define how instances of a given class can be split.

For example, the programmer can specify different splitting strategies for `Elements`:

Minimal split. All the object fields are allocated together.

```
layout MinElt: [Element] = rec{f1, f2, f3}
```

Maximal split. All the object fields are allocated separately.

```
layout MaxElt: [Element] = rec{f1} + rec{f2} + rec{f3}
```

Custom split Fields f_1 and f_2 together; field f_3 separately.

```
layout CustElt: [Element] = rec{f1, f2} + rec{f3}
```

Layout declarations define how/if objects in pools are split, and impose any splitting on all objects in a pool into different *records*. A layout declaration consists of a type/layout identifier (`MaxElt`), the class of objects it holds (`[Element]`), and record declarations (`rec`s) defining which fields go together. For example, a pool of type `MaxElt` will organize its objects with maximal split . This pool will be split into three different clusters, where each cluster contains a sequence of records, and each record contains a single field. For example, records of the first cluster contain values of field f_1. Creating pools of different layouts, and allocating an object with customized split and another with minimal split is done so:

```
pool elements : CustElement
pool elements' : MaxElt
... new Element⟨elements⟩
... new Element⟨elements'⟩
```

The function `m` can be written in SHAPES as follows. This function is oblivious to the layout of the parameter `e`, even though it is applicable to objects with different layouts.

```
⟨p⟩ def m(e: Element⟨p⟩): void
  e.f3 = true
def main(): void
  pool elems : MinElt
  var es = new Element⟨elems⟩[N]
  for i ←[0 .. N] do
    es[i] = new Element⟨elems⟩(i, 0, false)
  for i ←[0 .. N] do
    m(es[i])
```

2.3 Declaration of Pointer-Based Data Structures

As we have discussed previously, mapping pointer-based data structures to arrays is quite complex. In this Section, we show how a linked list could be annotated to support pool allocation. To show how code can be kept clean, we use typedef-like synonyms. The code looks as follows.

```
class List⟨x1, x2: [Node], x3: [Element]⟩
  synonym PooledNode = Node⟨x2, x3⟩
  synonym PooledElement = Element⟨x3⟩
  head: PooledNode
  def addFirst(elem: PooledElement): void
    this.head = new PooledNode(elem, this.head)
  def get(index: int): PooledElement
    var cur = this.head
    var i = 0
    while (cur != null && i < index)
      cur = cur.next; i++
    return (cur != null ? cur.elem : null)
  def removeFirst(): void
    this.head = this.head.next

class Node⟨x1, x2: [Element]⟩
  elem: Element⟨x2⟩
  next: Node⟨x1, x2⟩
  def init(elem: Element⟨x2⟩, next: Node⟨x1, x2⟩): void
    this.elem = elem; this.next = next
```

Note that nowhere in the `List` class declaration is there a reference to pool layout. The developer of a data structure does not need to consider, at all, where and how constituent objects will be allocated[2]. As discussed before, this is due to class parametricity and because class parameters are annotated with abstract types, delegating layout decisions to use-site, and instantiation-time. The only extra labour for the programmer is at the type level. The type information can be used at compile time, or dynamically, to generate layout aware code. We will give more details about it later. SHAPES code for other data structures, such as, a `Stack`, `Queue` or `ArrayList` can be found in Appendix B.

[2]Note that the type [C] means that it *may* be allocated in a pool of C objects.

2.4 Instantiation and Usage of Data Structures

SHAPES does not change code that reads and writes to data in memory. The "internal" code of a data structure is mostly the same as if there were no SHAPES annotations. But how does it look for the data-structure's client code? Let us consider the following code that creates an instance of `List`, adds a new `Element` to it, and then gets a pointer to the first element in the `list`.

```
pool nodePool : heap
pool elemPool : heap
var list = new List⟨heap, nodePool, elemPool⟩
list.add(new Element⟨elemPool⟩(0, 0, false))
var elem = list.get(0)
```

This code creates a list with the layout of Figure 5a: `Nodes` and `Elements` are not allocated in pools, and objects are not split. To obtain the layout from Figure 5b, we only need to replace the second line with:

```
pool elemPool : MinElt
```

Similarly, assuming the existence of a layout declarations `MinNode` which does not split objects, `MaxNode` which splits each `Node` into two clusters, and `MixElem` which stores f_2 and f_3 in one cluster, and f_1 in another, the layout of Figure 5c is obtained by replacing the first two lines with:

```
pool nodePool : MinNode
pool elemPool : MixElt
```

And using the `list` still goes as before:

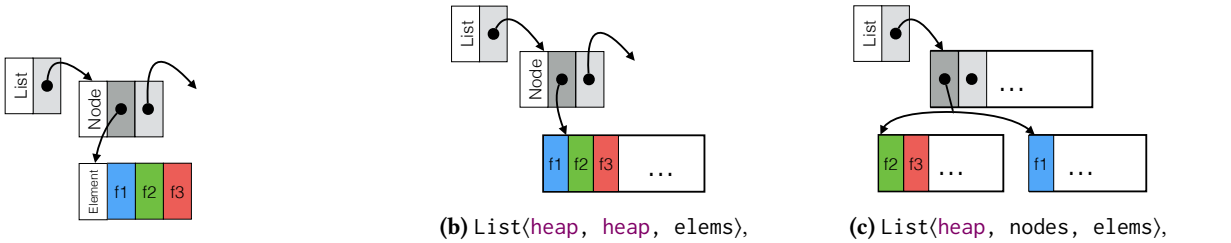
```
list.add(new Element⟨elemPool⟩(0, 0, false))
var elem = list.get(0)
```

2.5 SHAPES is Type Safe

Once SHAPES is integrated in a high-level language, the programmer will be able to enjoy all the advantages of such a language: abstract memory, automatic memory management, safey, etc. Type safety ensures the absence of errors which are possible when implementing memory optimisations manually. We now describe three such errors.

ERR1: Allocation of Objects in the Wrong Pool Allocating an `Element` in a pool of `Nodes` could be problematic: since they have difference sizes, access patterns may be disrupted (irregular strides); since they have different structure, they cannot be split easily using the same clusters. Thus, the following code would be forbidden by the type system, because `nodePool` is not a pool for `Elements`.

```
pool nodePool : MinNode
... new Element⟨nodePool⟩ -- ERR
```



(a) `List⟨heap, heap, heap⟩`

(b) `List⟨heap, heap, elems⟩`, where `elems:MinElt`

(c) `List⟨heap, nodes, elems⟩`, where `nodes:MinNode`, `elems:MixElt`

Figure 5. Three different layouts for the same data structure.

ERR2: Creation of Objects Out-Of-Thin-Air As we discussed before, SHAPES guarantees correctness of object reassembly. This is made possible through pool-aware references containing pool addresses and object offsets, and types which carry layout declarations that contain enough information to calculate offsets of fields.

Moreover, the type system should forbid incomplete, or overlapping layout declarations, such as the ones below.

```
layout Missing: [Element] = rec{f1} + rec{f2} -- ERR
layout Many: [Element] =
                    rec{f1, f2} + rec{f2, f3} -- ERR
```

ERR3: Mixing pools. Objects allocated in the same pool must have the same layout. For example, below, `n.next` expects objects laid out as in `MinNode`, while we store an object organized as in `MaxNode`

```
pool nodes1 : MinNode
pool nodes2 : MaxNode
var n = new Node⟨nodes1, heap⟩
n.next = new Node⟨nodes2, heap⟩
```

Neither do we support pools with the same layout sharing objects:

```
pool nodes3 : MinNode
n.next = new Node⟨nodes3, heap⟩
```

The type system can prevent such errors by extending techniques found, *e.g.*, in the ownership types literature [Clarke and Drossopoulou 2002; Clarke et al. 2013].

2.6 Formal Model

We have developed a formal model for the core SHAPES language, its type system, a low level language incorporating pools, and the compiler. We have proven that the type system is sound, and that compilation is meaning preserving. The formal model is not part of the current submission.

2.7 Let Objects be Objects. SHAPES is OO.

While objects may be deconstructed internally, SHAPES preserves the object concept at the surface-level. Given that the splitting strategies are external to class declarations, the programmer can still keep programming using the same object-oriented mindset. Objects encapsulate fields and methods that use those fields, independently of their layout. With SHAPES fully integrated in an object-oriented language, it will be possible to keep the same design principles of the object-oriented paradigm (modulo field visibility, unless the pools are created inside the objects themselves).

2.8 Design Decisions

Here we briefly discuss the design iterations and the considerations for the core language.

Pools and Class Parameterization In our initial attempts we described object splitting in the class definitions – but this required different class definitions for different layouts of the same class. We then tried to obtain object splitting through different pool parameters, whereby clusters of fields are annotated with pools indicating where they will be stored – but then we faced the problem that a pool could contain clusters originating from objects in different pools.

This led to the idea of layouts as separate entities, describing how objects of a given class are organized within a pool. Thus, pools are instances of layouts, and classes are parameterized by pools. Moreover, the fields of an object may point to objects located in other pools. This requires that a class would allow for several pool parameters.

Because we want to be able to access fields from pooled objects, and to iterate over pools safely and efficiently, we need to ensure that pools contain objects of the same class. To ensure this, we initially tried to infer that the use of class arguments was consistent, but then found it simpler to give types to class parameters.

Subtyping For now, SHAPES pools are monomorphic, in the sense that it is not possible to store objects of type T', s.t. $T' <: T$ in a pool P containing T values. We imposed this restriction in order to avoid having to handle any extra fields in T' that are not in T? As we already mentioned, storing objects of different sizes makes it harder to construct regular access patterns. If P is maximally split, we could deal uniformly with the T (sub)structure, and deal with any extensions separately.

We will evaluate the design decision of monomorphic pools when we have a working compiler.

Polymorphism Polymorphism is a corner stone of object-oriented programming. In conjunction with dynamic binding, polymorphism allows writing code that is future-proof, *e.g.* is able to invoke a method that has not yet been written. This is important for object-oriented reuse. Again, performance and flexibility are usually at odds. For example, in a study of refactoring a moderately sized C++ program in the Scientific Computing realm to use a more object-oriented design (*e.g.* SOLID [Chidamber and Kemerer 1994]), dynamic dispatch alone incurred an 4–8× slow-down because of negative inline effects [Källén et al. 2014]. A JIT compiler might be able to mitigate such effects.

When parameterising types over layout information, unless there is rudimentary support for polymorphism, programmers will be forced to duplicate semantically equivalent code that perform the same operations on differently laid-out objects. (This is what happens when optimisations like these are applied manually.) Parameterising classes over types, which in turn are parameterised by layout parameters is essentially the same problem.

Polymorphic code in shapes can be compiled into either: a single unified binary blob which inspects the types at runtime to calculate the addresses for loads and stores; many versions of the same code specialised for a particular (combination of) type(s). The former is less efficient than the latter, but causes less binary bloat. This problem is not unique to SHAPES (*c.f.* C++ templates) but exacerbated by additional parameters to types. Exploring using a JIT or trace compiler to produce specialised versions for hot paths in a program might prove a solid middle ground.

3 Implementation of Core SHAPES

We plan to integrate SHAPES into Encore[Brandauer et al. 2015], an object-oriented, statically-typed, actor-based language, which uses class declarations to define objects. Encore compiles to C code, thus the integration of SHAPES into Encore should be straightforward. Class declarations will be extended with pool parameters, types with pool arguments, and expressions with primitives for pool creation. The compiler will be extended with rules that use types to generate code that "knows" how to allocate objects into pools, and the Encore runtime will be extended with object pooling and compacting garbage collection.

As a proof of concept, we have developed a SHAPES prototype API in C that features pooling and splitting, and is designed to be linked with a managed run-time[3]. It provides a set of functions and structures to be used by the compiler to handle pools and pooled objects. Our API will be used for object pooling and pool compaction, and the existing runtime for allocation of non-pooled objects (the ones allocated in the `heap`). We have also evaluated it through a set of small benchmarks, to better understand if our implementation can indeed improve a program's performance. Our prototype and evaluation can be found at: https://github.com/jupvfranco/shapes. More in Appendix A.

In this Section, we briefly describe our implementation.

3.1 Object Pooling

Pools are chunks of memory that allow for contiguous allocation of objects, with or without splitting, such as described before. Pools are constructed from sub-pools, each with space for 4096 same-typed objects; whenever a sub-pool is full, another one is created. Our prototype aligns pools on a 4GB boundary[4], so that they have space to grow. In the unlikely event of 4GB not being enough for a data-structure, one can move the whole data structure to somewhere else in memory. The number of objects in a sub-pool is constant, but its size is variable. Given that no object can be smaller than 1 byte, the choice of 4096 objects in each subpool guarantees they will be at least of size 4KB, ensuring page alignment which is important for prefetching.

Whenever a pool of `T` objects is created, SHAPES reserves space for 4096 × `sizeof(T)`, and uses the layout information provided by the programmer to *logically* split this (sub)pool into several clusters. This means that the information found in layout declarations is kept during runtime.

In order to do this, our library must keep a table for all the types of the program; it stores in this table both *record types* and *pool types*. Record types represent blobs of memory that cannot be split, e.g. `rec {`f_1`, `f_2`}` of `CustElement`. Of course, these types can be primitive types, e.g. `int`, representing single-field records. Pool types are directly mapped from layout declarations and contain the number of clusters in a given pool, and the record types allocated in each cluster — remember that each cluster allocates a sequence of records with the same type. Table 2 shows the different record and pool types obtained from the `Element` class and respective layouts. This information can then be used to create pools and to allocate objects in pools. For example, when creating a pool of `Elements` with maximal split, SHAPES uses a `mmap` to reserve 4096 × 17 bytes: the first 4096 × 8 bytes will be used to allocate values in f_1 fields, the second 4096 × 8 bytes to allocate values in f_2 fields, and so on.

3.2 References to Pooled Objects

Our API does not exclude the usage of ordinary C pointers to objects and fields within pools. However, when an object is split across different clusters, a single pointer is not enough to re-assemble the whole object[5]. We have implemented

[3]Once a front-end implementation exists, the library will be linked with the run-time of the Encore programming language [Brandauer et al. 2015]. The library is already usable from C, but uses idioms expecting code generation and is therefore somewhat verbose and clunky.

[4]We have implemented our SHAPES API for a 64-bit address space, which gives us space for a large number of pools.

[5]Naturally, one could use more than a pointer, however this is expensive, and potentially unsafe.

Table 2. Layout information of `Elements` at runtime.

Record type	Size
`int`	8
`Element_f1_f2`	16

Record type	Size
`bool`	1
`Element_f1_f2_f3`	17

Pool types	#clusters	Clusters
`MinElement`	1	`Element_f1_f2_f3`
`MaxElement`	3	`int` | `int` | `bool`
`CustElement`	2	`Element_f1_f2` | `bool`

a structure for references, named `global_references`, that contains all the information required to refer to an object in a pool, whether split or not. Each reference is a 64 bit number — the size of a C pointer — containing:

- `type_id` (16 bits) – a type identifier,
- `pool` (16 bits) – the location of the pool,
- `subpool_id` (16 bits) – the sub-pool where the object lives,
- `index` – the offset of the object within the subpool (12 bits), which can be some number between 0 and 4096 (the number of objects allocated in a pool), and
- and 4 extra bits used for pointer compression and garbage collection purposes.

Keeping the type identifier of the pool that the reference is pointing to is useful for a number of reasons: it can be used during garbage collection to find the fields to be traced, or to calculate the field offsets on the fly[6]. The rest of the information is used to access independent object fields, or to re-assemble an object. The following calculation shows how to calculate the memory offset of each field of an `Element`, which is maximally split.

$$pool + 4096 \times 17 \times subpool\_id + \begin{cases} index \times 8 \ (\texttt{f}_1) \\ 8 * 4096 + index \times 8 \ (\texttt{f}_2) \\ 16 * 4096 + index \times 1 \ (\texttt{f}_3) \end{cases}$$

We know that every subpool holds 4096 objects and that the fields of every `Element` fit in 17 bytes. We can use this information, with the `pool` and `subpool_id` to find the starting address of the sub-pool, in which the object is allocated. Moreover, we know the size of each field, (8, 8 and 1), and that each subpool is split into three clusters. We can use this information to calculate the starting address of each cluster. For example, the third cluster has offset 16 * 4096 (8 * 4096 bytes occupied by the first cluster and 8 * 4096 occupied by the second). Using the field size, and the `index` of the object, we can find exactly the location pointed to by the field.

Compressed References. The system optionally allows for pointer compression by using object-relative addressing instead of global references. Since each subpool contains 4096 elements, object-relative addressing within a subpool fits in 12 bits, plus 1 bit to enable references into related subpools and global addresses via a table indirection. Object-relative addressing allows fast copying or relocation of entire subpools because pointer addresses are stable accross copies. This optimisation is orthogonal to the SHAPES goals, and thus not disussed in further detail.

[6]this is particularly important if using an interpreted language

3.3 Compacting Garbage Collection

We now discuss how we have implemented the last ingredient of SHAPES: a compacting garbage collector that reduces fragmentation within pools. The current version of the prototype requires a pause in the execution of the application. However, moving and copying garbage collectors are well-studied topics [Jones et al. 2016], and it should be possible to use our techniques with parallel or concurrent collectors. Our garbage collector collects and compacts pools individually. For each pool being managed, it creates a new pool of the same type, and copies live object by live object to the new pool. It then destroys the original pool. Given that the new pool only contains live objects, after the GC has run no garbage will be fetched to cache.

External references into a pool can be managed in several ways: system-wide garbage collection can update them, or external references can go via proxies which can be updated by the garbage collector. The latter design requires means to distinguishing between references within a pool and references into a pool.

3.4 The SHAPES Library

Functions provided:

Pool creation. Creation of a new memory pool with splitting strategy defined by `type_id`. The result is a pool reference that points to a pool in which allocation can happen.

```
pool_reference pool_create(uint16_t type_id)
```

Pool destruction. Destruction of a memory pool. All the objects currently allocated in the pool are freed.

```
int pool_destroy(pool_reference *pool)
```

Pool allocation. Allocation of an object in the pool `pool`, leaving all fields uninitialised. The result is a reference to the newly created object.

```
global_reference pool_alloc(pool_reference *pool)
```

Pool grow. Allocation of memory in the pool `*pool` for `n` more objects. When combined with an iterator, this function can create and initialise many objects at once.

```
int pool_grow(pool_reference *pool, const size_t n)
```

Pool shrink. Shrinking of the pool `*pool` by `n` elements.

```
int pool_shrink(pool_reference *pool, const size_t n)
```

Pool read. Reading a field of a pooled object referred by `reference`. In order to find the field, this function is given the cluster offset, the size of the record where the field is, and the field offset within the record.

```
void* pool_read(const global_reference reference,
                const size_t cluster_offset,
                const size_t record_size,
                const size_t field_offset);
```

For example, if the field being read is f_3, from an `Element` that is maximally split, then we have: `cluster_offset = 16, record_size=1, field_offset=0`.

Pool write. Writing a field of a pooled object referred by `reference`. Similar to the `pool_read` operation, but copying `data` to the location of the field.

```
void pool_write(const global_reference reference,
                const size_t cluster_offset,
                const size_t record_size,
                const size_t field_offset,
                const size_t field_size,
                const void* data)
```

3.5 Iteration

Iteration on pointer-based data structures that involve dereferencing all `global_references` may be expensive, due to all the calculations needed. However this too can be optimised. For example, assuming no fragmentation in a pool, if there are no constraints on the order of objects being accessed, one can just iterate over all objects in the pool by moving the index. For that, our library provides a function that iterates over the first `N` objects (of size `size`) in the pool, following their allocation order, and applies some function `fun` to each one of them.

```
void pool_iterate(const pool_reference pool,
                  const size_t size,
                  const size_t N,
                  const pool_apply fun);
```

We have also implemented another iteration function that calculates the location of the pool only in the beggining of the iteration and the location of subpools only when following a pointer from an object in a different sub-pool. The rationale behind this iteration is that the `index` of the object should be enough for most objects. One can implement other types of iteration. For example, if only one field is used, a similar iteration over all records of a cluster.

3.6 Compiling and Executing SHAPES Code

As we have mentioned before, the compilation process of SHAPES will be guided by the program's types. Given that types will have enough information regarding location and layout of objects, it will be possible to generate code using the SHAPES library. Given that the same data structure can be instantiated with different layouts, our first approach will be to generate all the possible code versions, for all the declared layouts. One could write a more optimised compiler, or even use Just-in-Time compilation, in order to avoid large binaries. However, this is orthogonal to SHAPES, and therefore we leave it for future work.

In order to demonstrate, how the SHAPES compiler will function, in this Section, we "manually" compile some code, considering two different layouts. We consider the layout information for the class and layouts described previously, and the following code:

```
pool elems = new MinElement
var elem = new Element⟨elems⟩
elem.f3 = false
```

It creates a pool of `Elements` with minimal split, it allocates a new `Element` in this pool, and writes the f_3 field of this object. This code shall be compiled to the following C code.

```
pool_reference elems = pool_create(MIN_ELEM_TYPE_ID);
global_reference elem = pool_alloc(&elems);
boolean b = false;
pool_write(elem, 0, 17, 16, 1, &b);
```

However, if the `Elementss` were maximally split, we would need to pass a different type identifier to the pool, and use different offsets when reading the field:

```
pool_reference elems = pool_create(MAX_ELEM_TYPE_ID);
// ... same as before
pool_write(elem, 16, 1, 0, 1, &b);
```

The code above is only possible when using a compiler that knows how to calculate the sizes and offsets of object fields. However we believe our proposed syntax contains enough information for such calculation. Note that there are several dynamic languages, such as Python, that do not require compilation. In this case, SHAPES would not be as efficient, however it could still be used, given that the layout structures contain enough information regarding the fields of each type, and their sizes.

4 Extended SHAPES

In this section we outline the promise of SHAPES, in particular what possibilities structuring parts of memory into pools unlock, how pools can be combined to create more complicated object structures, and how we intend to integrate garbage collection with SHAPES not only for reclaiming garbage objects, but to improve locality.

4.1 Garbage Collection and Compaction

Libraries that use pooling and splitting commonly do so in unmanaged languages. When manually transforming arrays of structures to structures of arrays in managed languages like Java and C#, programmers are giving up on

automatic garbage collection by tying the lifetime of a (deconstructed) object to its containing array. In SHAPES, we integrate garbage collection into the approach so that

- programmers do not need to give up garbage collection;
- garbage collection can be used as an *advantage* — in particular relying on moving and compacting for locality

Because structures are pointer-based in SHAPES, normal tracing garbage collection (for example) can be used to determine liveness. Thus, removing an object split over N clusters is still a single unlinking, regardless of the value of N.

Pools are Boundaries A pool can be thought of as a logical subheap nested in a bigger heap, where objects naturally belong together. Thus, when a garbage collector should decide how objects are placed in memory, it should do so based on pool membership. Thus, pools naturally give groupings to objects to help a collector do its job, and also gives a natural boundary for when to *stop* tracing.

The latter theoretically allows running garbage collection in a small subpart of the heap, for example to compact all links of a linked list, or change their order to list order post sorting. Since garbage collection can be localised to a pool (or even a subpool), garbage collection can be run more frequently, or even triggered by the programmer before a lengthy operation, similar to how programmers today transform matrices prior to big operations to better align access order with memory placement [Ureche et al. 2015].

For localised garbage collection to be possible, we can either rely on alias restrictions (*e.g.* like ownership types [Clarke et al. 1998], essentially control the visibility of pools) to exclude the possibility of incoming pointers or dynamically track incoming references (for example like [Clebsch et al. 2015]) to either fix aliased objects in place, or handle such pointers using indirections that allow aliased objects to move.

Tuning Placement Strategy Hirzel experiments with different placement strategies for garbage collection [Hirzel 2007]. He finds that every placement strategy (allocation order, depth-first, breadth-first, etc.) performs best on certain benchmarks and worst on others. By allowing the heap to be logically partitioned into pools, SHAPES will unlock the possibility to tune placement strategy on a per-data structure-basis. This stresses the ownership types-like [Clarke et al. 1998] nature of SHAPES, but instead of prohibiting certain pointers, we simply pay less attention to them when garbage collecting (*i.e.*a pointer from outside a pool to an object in the pool should not control its placement in the pool).

Inter-Pool Object Affinity So far, we have discussed affinity of objects in the same pool, but in real-world scenarios it is common to construct linked structures built from different objects. For example, we may have a container with customers `Records`; where each `Record` has a field for `AccountInfo`. this case, we want to store the `Record` entries in their pool in an order compatible[7] with the `AccountInfo` entries (in their pool). If the orderings are thus aligned, accesses to ages to access grades will enjoy cache locality and efficient prefetching on batch operations, *e.g.* on iteration. This would allow having two lists sharing the same set of grade object, which is not easily possible if grade objects must be embedded in the links.

To this end, SHAPES must be able to express relationships between pools. For example, in the above example, the `AccountInfo` pool is *dependent* on the `Record` pool. Dependencies between pools introduces a bijection between objects. For example, we could say that a `AccountInfo` object corresponds to a `Record` object if it is pointed to by a field in the same `Record`. This should mean, for example, that a garbage collection cycle in the latter pool also triggers on in the former, and that pointers from the `Record` pool to `AccountInfo` pool decides the order in the `AccountInfo` pool.

This could be encoded *e.g.* by nesting pool declarations:

```
layout records: [Record] = ... {
  layout accounts: [AccountInfo] = ... // dependent
}
```

This is easily tractable, but also static. An alternative approach is to allow dependencies to change dynamically (since they will be enforced dynamically by garbage collection).

4.2 Concurrency & Parallelism

Introducing concurrency into SHAPES brings in all known issues to do with avoiding data races in the source code, and also for the garbage collector. However, SHAPES does not introduce any new problems: any data-race-free SHAPES program will be compiled to data-race-free binary code: Even though object identity of pooled objects is represented through tuples (*c.f.* Section 3), these tuples will be used atomically.

Because of how pools are structured, parallelism can naturally happen on a subpool or cluster boundary. For example, summarising all `Elements` in a collection thus:

```
if f3 then f1 + f2 else f1
```

can be broken up into two parallel operations:

```
if f3 then f2 else 0 // Op 1          f1 // Op 2
```

which are performed by separate threads on each cluster (both task and data parallel) and then reduced. If a pool is free from garbage, or if it is possible to efficiently determine if an entry is live or not, it is not necessary to traverse the pointer-based spine of the data structure. *This means that compact pools are suitable for vectorisation.* Thus, it will be possible to compile some operations on collections of objects to vector operations, for improved efficiency.

[7] In the sense that access patterns for both objects arise similar to as if they were perfectly interleaved in the same pool (which we don't want to do in the general case because of poor cache utilisation).

4.3 Dynamically Changing Field Affinity

While certain pool layout might be advantageous to a certain access pattern of the objects in the pool, the access pattern might change during program execution. For example, at the beginning of a program, health and age data are accessed together, while later on, health and addresses are being accessed together. Transforming a pool from one layout to another is relatively straightforward, and can be reflected by a type change, meaning it can be readily expressed in code. We could for example use techniques as the reclassification of Fickle [Drossopoulou et al. 2001] or type-state related mechanisms [Strom and Yemini 1986].

Naturally, changing an object's type requires considering who can witness such a type change. One possibility is to piggyback on garbage collection to find all references, and combine with code update, which has been done leveraging ownership-like annotations [Boyapati et al. 2003], to make sure that subsequent accesses have the proper types. An alternative approach is to rely on dynamic lookup, which only requires changing the type of the target(s), but is less performant.

When to apply layout changes is an interesting question. As a first approach SHAPES will leave this to the programmer to decide. The ultimate goal is to have pool layouts be determined by a combination of clever compiler and run-time as a result of monitoring the execution of the program.

4.4 Value Semantics

Low-level languages like C and C++ (and also some high-level languages like C#, Rust and Eiffel) distinguish between reference (pointer) semantics and value semantics, where the former passes values by reference (*i.e.* sharing, or aliasing) and the latter by passing around copies of values. Like storing values in arrays avoid indirection, so can embedding values in objects. Choosing between reference semantics and value semantics is an important part of layout as it allows bringing in data (value semantics) and avoiding cache pollution (reference semantics).

SHAPES uses pointer semantics by default. However, when *e.g.* iterating over our linked list, if the `elem` field is always read, it makes sense to embed it in the `Node` that points to it. This would avoid dereferencing and it would fetch useful data (`Element`) to cache in advance (when fetching a `Node`). We will extend SHAPES with value semantics. Ideally, SHAPES can allow this optimisation while keeping it separated from the program's logic — keeping classes parametric with their layouts, and extending layout declarations with `embedding` information:

```
layout valNode: [Node] = rec { next, embed elem }
```

When dealing with value semantics, it is hard to keep the code completely unware of the layout. Challenges will arise with aliasing for example, and usual solutions involve copying of objects. Following C and C++, class declarations could be annotated to control embedding:

```
class Node⟨loc⟩
 elem: embed Element
 next: Node⟨loc⟩
```

Note the class no longer takes the location of the `Element` as parameter. Thus, value semantics reduces pool parameters.

5 Related Work

Object Pooling and Splitting Pooling and splitting are two techniques often used to improve a program's performance through better data layout. The idea of data placement to reduce cache misses was first introduced by Calder et al. [Calder et al. 1998], where the authors apply profiling techniques to find temporal relationships among objects. This work was then followed up by Lattner et al. [Lattner and Adve 2003, 2005] where rather than relying on profiling, static analysis of C and C++ programs finds what layout to use. Huang et al. [Huang et al. 2004] explore pool allocation in the context of Java. Object Splitting was introduced by Franz and Kistler [Franz and Kistler 1998], where fields are classified as hot (accessed frequently) and cold (accessed less frequently); this classification is used to decide how to split objects. Since then splitting has been combined with pooling [Chilimbi and Shaham 2006; Curial et al. 2008; van der Spek et al. 2010; Wang et al. 2010, 2012].

Garbage Collection for Program Locality Hirzel [Hirzel 2007] uses a moving garbage collector in order to implement several data layouts of object oriented programs and evaluate which layout presents the best performance.

Heap Partitioning and Regions Loci [Wrigstad et al. 2009] split the heap into per-thread subheaps plus a shared space. This division is only *conceptually*, and does not aim to affect representation in memory. Some languages split the heap into several sub-heaps in order to simplify garbage collection or parallelism. Examples of these languages are Pony [Clebsch et al. 2015; Clebsch and Drossopoulou 2013] and Erlang [Armstrong 2007; Armstrong et al. 1993]. None of these languages share direct goals with SHAPES, in the sense that they do not try to improve data locality. Particularly in Pony and Erlang, the programmer does not have any control on how to divide the heap.

Jaber and Kulkarni presented a notion of heap partitioning that allow different data structures to be allocated in different partitions [Jaber and Kulkarni 2017]. They use static analysis to infer ownership relations between objects, and a preprocessor that adds ownership information to the code without changing its meaning. They seek to integrate this heap partitioning in computation offloading tools, region-based memory management, and with techniques for memory locality optimisations. This work relates to SHAPES in that it aims

at improving memory locality. However, it does not given any layout control to the programmer, neither does it give any guarantees of contiguous allocation, or means for object splitting.

Tofte and Talpin introduced the concept of region-based memory management [Tofte and Talpin 1994, 1997]. They use region types, which divide memory in regions, in an ML language, where allocation and deallocation are inferred from type and effect analysis. This idea was used in the Cyclone language [Hicks et al. 2004], which is concerned with safety of C-like languages. Deterministic Parallel Java also provides means to split data in the heap: Java code is annotated with regions information used to calculate the effects of reading and writing to data [Bocchino et al. 2009].

Pools are similar to regions (or Rust's lifetimes) in that they both guarantee that objects are allocated in the same part of memory, but they differ in the following salient points:

- Objects of the same region have the same lifetime, while objects from the same SHAPES pool may have different lifetimes, meaning less floating garbage. The inability to GC within a region can lead to substantial memory consumption [Berger et al. 2002]. In SHAPES, when an object within a pool becomes unreachable, it will be possible to collect it.
- As regions are nested objects, they may only point to objects with shorter lifetimes. No such restriction in SHAPES.
- Regions do not support object splitting — all fields of an object are stored together, and thus within the object's region. Rust's multiple lifetime parameters to a value, only allow the determination of the region where the field's target is stored, but not where the field itself is stored.
- Regions do not offer any guarantees as to whether objects are stored contiguously, or as to which objects are stored within them. Therefore, regions cannot support efficient iteration, whereby pointers are incremented by a small offset, such as in our iterators (*c.f.* Section 3).

On the other hand, it is possible to develop in Rust collections which store split objects: Using Rust traits, one can impement data structures whose interface hides the splitting from the client of the structure, but internally is layout-aware. All field accesses are represented as function calls, which are however inlined and thus incur no penalties. Nevertheless, it is necessary to write these data structures by hand, while in SHAPES we only require annotations. Moreover, one cannot have more than one mutable alias to any of the elements of the data strcure. We show the list-of-elements example encoded with Rust traits in Appendix C.

Location Abstractions in Programming Languages In the context of NUMA systems, Franco and Drossopoulou use annotations to describe in which NUMA nodes the objects should be placed [Franco and Drossopoulou 2015],with the aim to improve program performance by reducing memory accesses to remote nodes, ignoring any possible in-cache data accesses.

Programming Languages Supporting Locality Vollmer et al. propose an approach to compiling tree-shaped data structures into a more efficient, packed representation [Vollmer et al. 2017]. De Wael et al. propose "Just-in-Time Data Structures" that adapt their internal workings to fit a current usage scenario [De Wael 2015]. It would be interesting to use SHAPES to implement such data structures. In the context of dynamic languages, Bolz et al. use storage strategies to dynamically optimise collections containing instances of the same primitive type [Bolz et al. 2013]. Ureche et al. introduced an extension to Scala that allows for safe and automatic changes in data layouts. The programmer uses scopes to define transformations of data layout, and the compiler uses this information to generate the code that accesses the data [Ureche et al. 2015]. This work is very similar to the SHAPES ideas in the sense that the programmer needs not to change the code accordingly to the layout, and in that it uses types for correctness guarantees. Its goals are different, and does not provide programmers new tools to control data layout and placement — for example, it is not possible to express that an array should hold pointers to objects consecutive in memory, like in SHAPES. Furthermore, when providing array-of-structs and struct-of-arrays layout transformations, a programmer must specify both representations separately, and how to access them. In a sense, this work is complimentary to SHAPES and exploring how they can be combined is an interesting direction for future work.

6 Conclusions

We have introduced SHAPES, an extension of high-level managed languages that allows the programmer to shape data in memory using object pooling and splitting. SHAPES was conceived for a managed language, and relies on garbage collection for compaction and aligning objects across pools, but is useful even without these features in an unmanaged setting. SHAPES adds to languages class declarations parametric with layouts, postponing layout decisions from declaration to creation time. This makes it possible to write code layout unaware and keeping memory abstract, while enjoying the performance of good memory layouts. We have introduced the core ideas of SHAPES, described our implementation, and discussed ideas for potential optimisations.

A Preliminary Evaluation

We evaluate our implementation through a set of sequential micro-benchmarks. We now describe our evaluation methodology and results.

Methodology. Our benchmarks are written in C, Encore [Brandauer et al. 2015], and Java. We compare SHAPES code manually compiled into C code that uses our library against:

- Malloc: C code using `malloc` and `free` functions for data (de)allocation;
- Malloc_opt: Optimised version of Malloc code, which uses a single `malloc` at the beginning of the program with space for all the data allocated and data manually placed consecutively in that segment with even strides. Assuming the same logic as in the malloc version of a benchmark, the Malloc_opt code gives us a lower bound on optimal number of cache misses and execution time.
- Encore: Encore is an actor-based, object-oriented programming language that compiles to C, and allocation groups objects of the same size together. Encore is also a managed language and uses the Pony garbage collector and allocator [Clebsch et al. 2015];
- Java: A managed, and well-known Object-Oriented language.

Encore and Java are different in that Java used a moving collector, whereas Encore does not. SHAPES, Encore and Java all use `mmap` on Linux to request memory pages.

We ran our benchmarks on a machine equipped with an Intel Core i7-3770 CPU @ 3.40GHz, with 8 cores on a single socket, and 16GB of memory. L1, L2 and L3 caches was 32KB, 256KB, and 8192KB, respectively and cache lines are 64 bytes. L1 and L2 were 8-way associative and L3 is 16-way associative. The operating system was Ubuntu 14.04.5 LTS. The C code was compiled with GCC 6.2.0 and using level `O3` of optimisations; and we used Java 1.7. We used `perf` [Perf 2017], a lightweight profiler that uses the CPU performance counters of Linux. We used `perf stat -e cache-misses -r <repetition> <benchmark>`, which reports the average number of cache misses and execution times. We repeated each benchmark 30 times and report averages.

Benchmarks and Results We now discuss the benchmarks used and the results obtained. We first study the benefits and overhead of object pooling, which are presented in Figure 6.

pool_array Compares a SHAPES pool against a `malloc`ed array. It creates a pool, and an array (a single allocation for the entire array and its objects), with `size` objects, each one with two fields using minimal split. It then iterates over this pool twice to initialise all the objects' fields, and then to read all the objects' values. Note that the malloc and malloc_opt versions are the same for this example. Moreover, we do not present Java and Encore results for this benchmark, as these languages only support arrays of pointers, and it is not possible to have an array of objects with value semantics (although value semantics seem to be coming in Java 10). Results are in Figure 6a. Naturally, given that the malloc version is as optimal as it can be, it shows better behaviour than SHAPES. Although malloc and SHAPES are comparable in terms of cache misses, SHAPES is slower than malloc. This overhead is due to the extra calculations needed for dereferencing pointers to pooled objects. We could further optimise the SHAPES code for this benchmark, as those calculations can be avoided: if objects have no split, a C pointer would be enough for this this program. We leave this to further work.

list_rand Allocates a linked list of 10^6 nodes, interleaved with the allocation of unrelated objects (of random sizes of 1–16 bytes), allocated every `N` nodes created. The program then iterates over all the nodes of this data structure. In the SHAPES version, all the nodes are allocated in a pool, with minimal split. Results are in Figure 6b. As expected, given that unrelated objects are not allocated in the pool of nodes, they do not create any fragmentation. Thus, they do not affect performance, both in terms of cache misses and execution time. The interleaved allocation does affect malloc, Encore and Java though.

iterate Allocates a pool (in SHAPES) or an array (in C) with space for `size` pointers, initialises as many objects, stores them in the array and iterates over all of them. Even in the malloc version, there will not be any fragmentation, as nodes are allocated one after the other. Results are in Figure 6c. It is interesting to see that SHAPES has comparable performance with malloc_opt in terms of cache misses, and is better than malloc in both cache misses and execution time, even though dereferencing of references to pooled objects is more expensive than dereferencing of C pointers. We believe this is associated with extra space for meta-data required by malloc, and with the allocation of nodes, which is faster with SHAPES.

rand_access Similar to **iterate**, but rather than iterating over all objects in allocation order, it randomly picks `size` indices of objects to be dereferenced. Results are in Figure 6d. Although SHAPES is not accessing objects in order of allocation, it still has better performance than the others.

We now discuss the impact of different splitting strategies and report results in Figure 7.

max_min Creates a pool with space for `size` objects with three fields and it iterates twice over all the objects: first writing to one of the fields, and then to another field. One of the fields is never used. We measure SHAPES's performance using maximal and minimal split of objects. The maximal split presents better performance. This happens because the unused field is never loaded to cache.

conditional Similar to **max_min**, but in the second iteration we only read the three fields if a certain condition is satisfied. We report performance for different probabilities of accessing the full object using maximal and minimal split, and we see that for smaller probabilities of access the maximal split is better than the minimal one. However, these two layouts get similar behaviours for bigger probabilities.

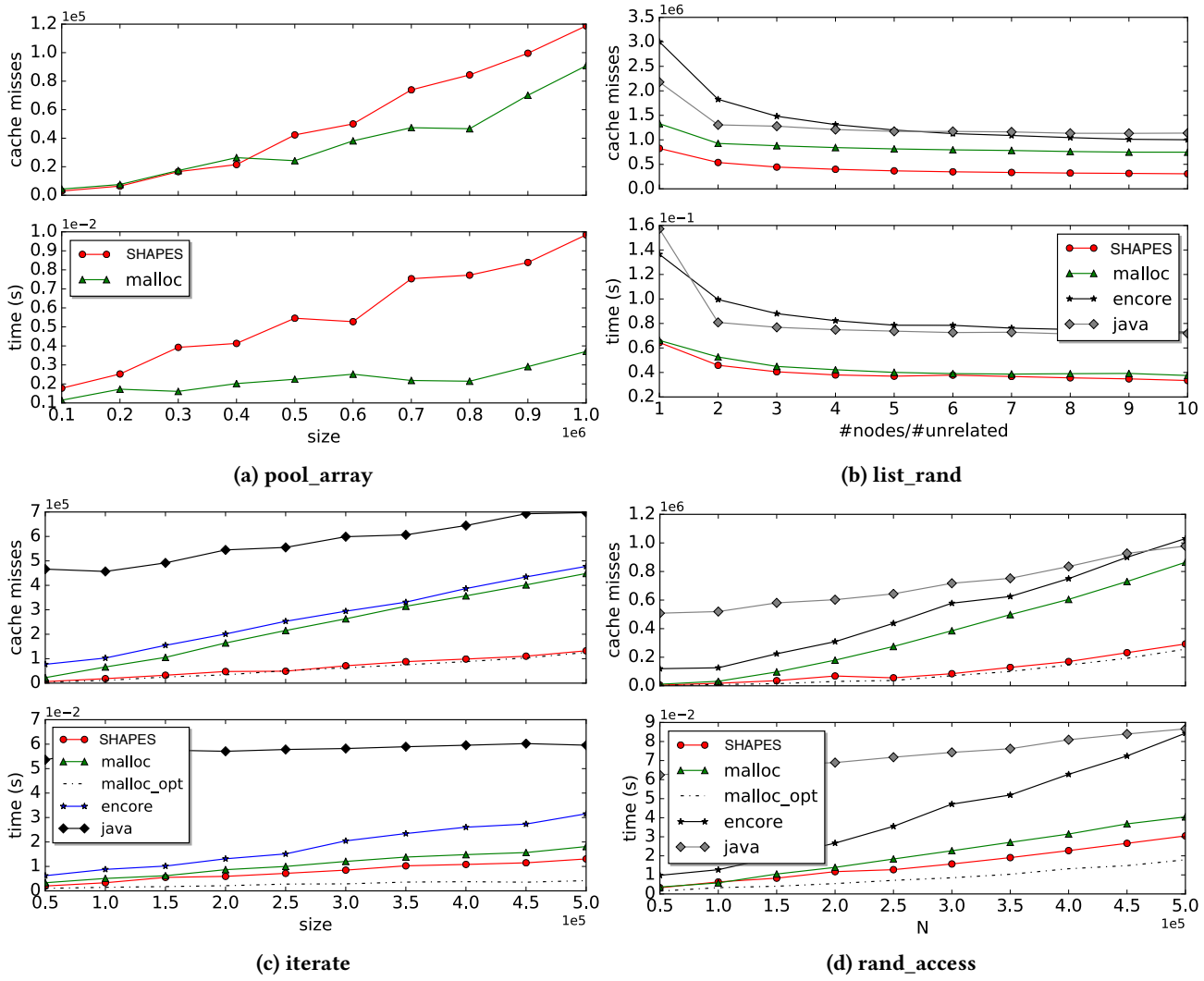
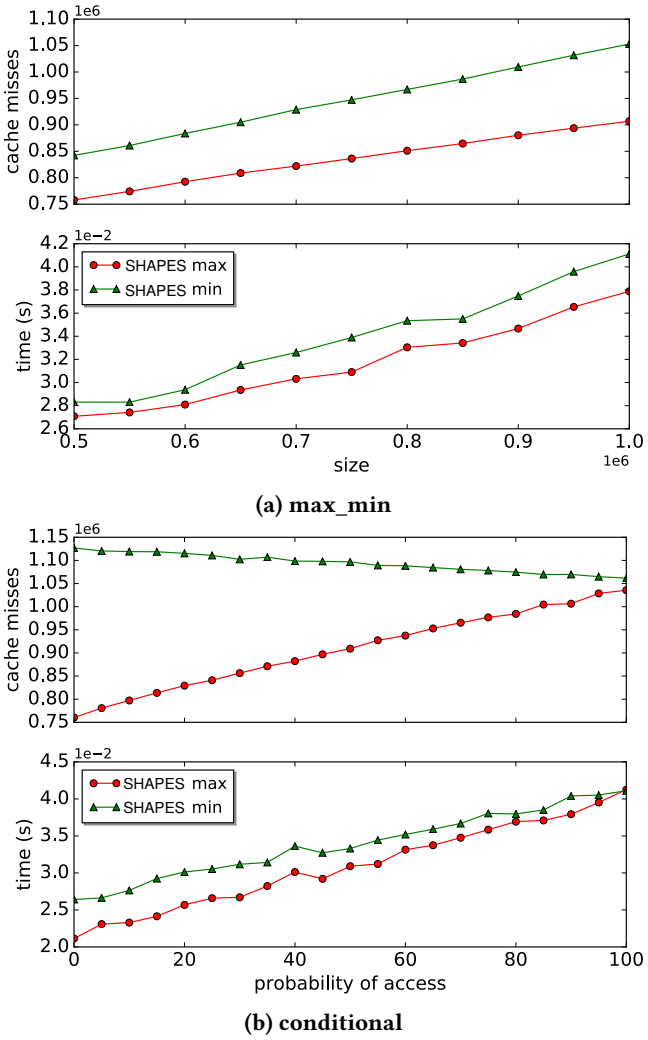


Figure 6. Comparison of pools against arrays and pointer-based data structures. Smaller is better in all the plots. Cache misses and time plots share the same x-axis.

We now discuss the impact of data structure re-organization and report results in Figure 8. Namely, throughout a program's execution, data structures may be modified: new objects are added, other objects become unreachable, data structures are sorted or reorganized. The SHAPES prototype does not move objects in pools, when objects are added or become unreachable, or when data structures are sorted. For example, when a `Node` is deleted from `VideoList`, even if it becomes unreachable, it is not removed from the pool; when a `Node` is inserted, it is appended to the end of the pool, independently from where it is logically added. This means that the order of `VideoList` nodes may be different from their pool order, and that the pool may contain useless `Nodes`, leading to deteriorating cache locality. To overcome this issue we intend to use a moving and compacting garbage collector, which will reorganise pools so that their objects are allocated in the same order they are used. We leave the implementation of such a collector for future work, and we now evaluate the impact of fragmentation, and the benefits of compaction, when simulating the lifetime of a data structure.

lifetime Creates a linked list of 10^6 nodes, and performs $5 * 10^5$ operations. These operations can be: the *insertion* of a node in a given index; the *deletion* of a node in a given index; and the *finding* of a node in a given index. When deleting an object, we use an explicit `free` in the malloc version of the code, but no explicit deallocation in the SHAPES version[8]. In Java and Encore, garbage collection is not triggered. All these operations cost $O(n)$. From all the operations performed, a

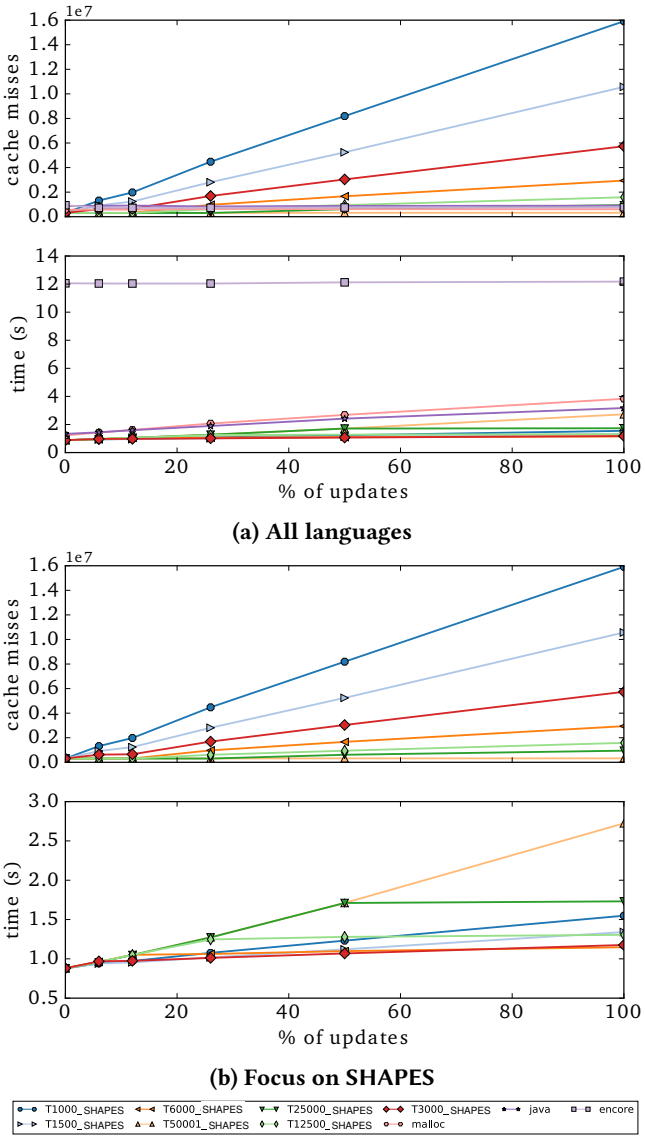
[8]In SHAPES, removed objects in a pool are freed automatically on compaction.



(a) max_min

(b) conditional

Figure 7. Comparison of different splitting strategies. Smaller is better in all the plots. Cache misses and time plots share the same x-axis.



(a) All languages

(b) Focus on SHAPES

Figure 8. Cache misses and execution time of Java, Encore, malloc, and SHAPES.

given percentage P (% of updates in the plot) will be updates ($\frac{P}{2}$% of operations are insertions and $\frac{P}{2}$% deletions).

The SHAPES version applies some compaction and reordering to the pool where the linked list is allocated — it creates a new pool with the same layout and it copies all the objects from one pool to the other following the iteration order. The source pool is then deallocated and the destination pool is used for the rest of the program's execution. Compaction and sorting is triggered after T updates on the data structure. We vary the percentage of updates, and we measure cache misses and execution time of Java, Encore, malloc, and SHAPES (for different Ts). The results are in Figure 8.

In this benchmark, contrary to the previous results, the number of cache misses does not affect the execution time. For instance, SHAPES behaves worse than the others in terms of cache misses and better in term of execution times. This happens because the whole (or most of) data structure fits in the last level cache, meaning that only accesses to new objects will result in cache misses. However, when using compaction, SHAPES copies all the reachable objects into a new pool.

With different levels of fragmentation, SHAPES shows better execution time (not cache misses) than the rest of the languages. This is expected, as we already saw that even when reading random objects of a pool, SHAPES presents better behaviour (c.f. Figure 6).

Although, SHAPES performs better than Java, Encore and malloc, in terms of execution time, it is interesting to observe that with the right update threshold, SHAPES' performance

can be even better, as we can see in Figure 8b. If no compaction happens ($T = 50001$) throughout the program's execution, then we can see that fragmentation impacts the program's performance: it is directly proportional to the probability of updates. If compaction does not happen often enough ($T = 25000$), we can see that once half of the updates were performed, and the pool was compacted, the execution time becomes constant. However, if compaction happens to often, it also hurts performance, as compaction itself is a $O(n)$ operation[9]. For this *specific* benchmark, an optimal threshold, with respect to execution time is 3000 updates to the data structure before it is compacted.

It should be noted that the current implementation of SHAPES is still in a prototype stage and therefore not yet optimised. In part, this is due to its library implementation, that makes many optimisations difficult that a compiler could easily do — in particular hiding costs (*e.g.* through careful instruction order optimisation) of and caching results of address calculations for object accesses in the presence of splitting. This explains why the significantly reduced cache misses have less impact on run-time than one might expect.

B Minimal API Implementation

B.1 Stack

```
class Stack⟨thisLoc, elemsLoc: [Element]⟩
  nodesPool: MinimalNodes
  top: Node⟨nodesPool, elemsLoc⟩

  def peek(): Element⟨elemsLoc⟩
    this.top.elem

  def pop(): Element⟨elemsLoc⟩
    let cur = this.top
    this.top.next = cur.next
    cur.elem

  def push(e: Element⟨elemsLoc⟩): void
    var newTop = new Node⟨this.nodesPool, elemsLoc⟩
    newTop.elem = e
    newTop.next = this.top
    this.top = newTop
```

B.2 ArrayList

```
class ArrayList⟨thisLoc, elemsLoc: [Element]⟩
  elems: Array[Element⟨elemsLoc⟩]
  size: int

  def init(cap: int): void
    this.elems = new Array[Element⟨elemsLoc⟩](cap)
    this.size = 0

  def add(e: Element⟨elemsLoc⟩): void
    if this.size + 1 == this.elems.length() then
      this.resize()
    end
    this.elems[this.size] = e
    this.size += 1

  def resize(): void
    var tmp = new Array[Element⟨elemsLoc⟩](this.size *
        2)
    for i ←[0 .. this.size] do
      tmp[i] = this.elems[i]
    end
    this.elems = tmp

  def remove(i: int): void
    this.shiftLeft(i+1)
    this.size -= 1

  def shiftLeft(start: int): void
    if start > 0 then
      for i ←[start .. this.size - 1] do
        this.elems[i] = this.elems[i+1]
      end
    end
```

B.2.1 Queue

```
class Queue⟨thisLoc, elemsLoc[Element]⟩
  elems: Array[Element⟨elemsLoc⟩]
  start: int
  end: int
  initCap: int = ...

  def init(): void
    this.elems = new Array[Element⟨elemsLoc⟩](initCap)
    this.start = 0
    this.end = 0

  def push(e: Element⟨elemsLoc⟩): void
    if this.end + 1 == this.elems.length() then
      if this.start > 0 then
        this.shift(this.start, 0) end
      else this.resize() end
    end
    this.elems[this.end] = e
    this.end += 1

  def pop(i: int): Element⟨elemsLoc⟩
    var e = this.elems[this.start]
    this.start += 1
    return e

  def resize(): void // as in the ArrayList
```

[9] We have not implemented external pointer reassignment, as a moving garbage collector would.

```
def shift(from: int, to: int): void // similar to
    the ArrayList
```

C Rust Version

Rust does not support object splitting. Nevertheless, using traits, one can provide implementations for the various data layouts, allow the client to choose which layout they wanted to use, and at the same time avoid overhead stemming from dynamic dispatch. The client code can be layout unaware. Still, this requires a lot of programming: the transformation from Array_of_Structs to Struct_of_Arrays is exposed in each of the traits, each of the traits has to provide methods for accessing the fields. Moreover, mutable references to different elements of the data structure are impossible.

Below we show how traits can be used to program the List-of-Elements example from our paper, just for two layouts. The two implementations of `Element` provide functions `f1`, `f2` and `f3` which are layout aware, and know how to access the individual fields. We also require implementations of lists-of-elements which are layout-aware. These functions are inline expanded, and therefore do not incur performance penalties. However, writing this code is unnecessary in SHAPES. The client code, eg the function `iterate`, is layout-unaware, and can access the field `f1` in the normal way.

Moreover, it is very difficult to obtain more than one mutable reference into the collection. We discuss this at the very end of the section

Object Splitting in Rust, Immutable References We show how we can split objects, using traits and immutable references.

```
trait Element {
  fn f1(self) → usize;
  fn f2(self) → usize;
  fn f3(self) → bool; }

trait ElementList<E: Element>{
  fn get(&self, i:usize) → E;
  fn length(&self) → usize;
  fn iterate(&self) {
    let n = self.length();
    for i in 0..n {
      let element = self.get(i);
      println!("whatever {}", element.f1()) }}}

struct MinimalSplit {
  elements: Vec<(usize,usize,bool)> }

impl <'e> Element for &'e (usize,usize,bool) {
  fn f1(self) → usize { self.0 }
  fn f2(self) → usize {  self.1 }
  fn f3(self) → bool   {  self.2 } }

impl <'e> ElementList<&'e (usize,usize,bool)> for &
    'e MinimalSplit {
  fn get(&self, i:usize) → &'e (usize,usize,bool)
      {  &self.elements[i] }
  fn length(&self) → usize {
      self.elements.len() }}


struct MaximalSplit {
  f1s : Vec<usize>,
  f2s : Vec<usize>,
  f3s : Vec<bool>  }

struct MaximalSplitRef<'e> {
  m: &'e MaximalSplit, i: usize }

impl <'e> Element for  MaximalSplitRef<'e> {
  fn f1(self) → usize { self.m.f1s[self.i] }
  fn f2(self) → usize {  self.m.f2s[self.i] }
  fn f3(self) → bool   { self.m.f3s[self.i] } }

impl <'e> ElementList<MaximalSplitRef<'e⟩ for &'e
    MaximalSplit {
  fn get(&self, i:usize) → MaximalSplitRef<'e> {
    MaximalSplitRef { m: self, i: i } }
  fn length(&self) → usize {
    self.f1s.len() }}

fn main() {
  println!("Hello, world!");
  let minsplit = MinimalSplit {
      elements: vec![(1,2,true), (10,20,false)]};
   (&minsplit).iterate();
   let maxsplit = MaximalSplit {
     f1s: vec![1,10],
     f2s: vec![2,20],
     f3s: vec![true,false]};
   (&maxsplit).iterate(); }
```

Object Splitting in Rust, Mutable References Extending the code from above to support object splitting as well as several mutable references, is difficult. Adding the setter method `setf1` to the `Element` trait as per below gives a compiler error, because the tuple is immutable, and we cannot mutably borrow an immutable field.

```
trait Element {
  fn setf1(self,j:usize) → ();    ...  }

impl <'e> Element for &'e (usize,usize) {
 fn setf1(self,j:usize) → () {
    self.0=j    // COMPILER ERROR
  }      ... }
```

On the other hand, implementing the Element trait for a *mutable* reference (i.e., `& mut 'e`) will make it very difficult to pass mutable references to the outside.

Acknowledgments

We would like to thank Stephan Brandauer for the discussions about Rust and Kim-Ahn Tran for the suggestions on the evaluation, as well as the anonymous referees for their comments. This project received funding from the FP7 project UPSCALE, the Swedish Research council through the grant Structured Aliasing, the UPMARC Linneaus Centre of Excellence, and EPSRC (grant EP/K011715/1).

References

Joe Armstrong. 2007. A History of Erlang. In *HOPL*. ACM, 6–1.

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1993. Concurrent Programming in ERLANG. (1993).

Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation. *SIGPLAN Not.* 37 (2002), 1–12. DOI: http://dx.doi.org/10.1145/583854.582421

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey L. Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*. 97–116.

Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *OOPSLA*. ACM, 167–182. DOI: http://dx.doi.org/10.1145/2509136.2509531

Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. 2003. Lazy Modular Upgrades in Persistent Object Stores. In *OOPSLA'03*. ACM, 403–417.

Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, EinarBroch Johnsen, KaI. Pun, S.LizethTapia Tarifa, Tobias Wrigstad, and AlbertMingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming*. Springer, 1–56.

Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-Conscious Data Placement. In *ASPLOS VIII*. ACM, 139–149.

Shyam R Chidamber and Chris F Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (1994).

Trishul M. Chilimbi and Ran Shaham. 2006. Cache-Conscious Coallocation of Hot Data Streams. In *PLDI '06*. ACM, 252–262.

Dave Clarke and Sophia Drossopoulou. 2002. Ownership, Encapsulation and the Disjointness of Type and Effect. *SIGPLAN Not.* 37, 11 (2002), 292–310. DOI: http://dx.doi.org/10.1145/583854.582447

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. LNCS, Vol. 7850. Springer, 15–58. DOI: http://dx.doi.org/10.1007/978-3-642-36946-9_3

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA '98*. ACM, 48–64. DOI: http://dx.doi.org/10.1145/286936.286947

Sylvan Clebsch, Sebastian Blessing, Juliana Franco, and Sophia Drossopoulou. 2015. Ownership and Reference Counting based Garbage Collection in the Actor World. In *ICOOOLPS'2015*. ACM.

Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully Concurrent Garbage Collection of Actors on Many-Core Machines. In *OOPSLA'2013*. ACM.

Stephen Curial, Peng Zhao, Jose Nelson Amaral, Yaoqing Gao, Shimin Cui, Raul Silvera, and Roch Archambault. 2008. MPADS: Memory-Pooling-Assisted Data Splitting. In *ISMM '08*. ACM, 101–110.

Mattias De Wael. 2015. Just-in-time Data Structures: Towards Declarative Swap Rules. In *WODA 2015*. ACM, 33–34. DOI: http://dx.doi.org/10.1145/2823363.2823371

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2001. Fickle: Dynamic Object Re-classification. In *ECOOP'01*. Springer, 130–149.

Juliana Franco and Sophia Drossopoulou. 2015. Behavioural Types for Non-Uniform Memory Accesses. In *PLACES 2015*. 109–120. DOI: http://dx.doi.org/10.4204/EPTCS.203.9

Michael Franz and Thomas Kistler. 1998. *Splitting Data Objects to Increase Cache Utilization*. Technical Report. University of California, Irvine.

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with Safe Manual Memory-Management In Cyclone. In *ISMM'04*. ACM, 73–84. DOI: http://dx.doi.org/10.1145/1029873.1029883

Martin Hirzel. 2007. Data Layouts for Object-Oriented Programs. In *ICMMCS*. ACM, 265–276.

Xianglong Huang, Stephen M Blackburn, Kathryn S Mckinley, J Eliot, B Moss, Zhenlin Wang, and Perry Cheng. 2004. The Garbage Collection Advantage: Improving Program Locality. In *OOPSLA*. ACM.

IvyBridge 2016. Intel Ivy Bridge. http://www.7-cpu.com/cpu/IvyBridge.html. (2016).

Nouraldin Jaber and Milind Kulkarni. 2017. Data Structure-Aware Heap Partitioning. In *CC'2017*. ACM, 109–119. DOI: http://dx.doi.org/10.1145/3033019.3033030

Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management*. CRC Press.

Malin Källén, Sverker Holmgren, and Ebba Hvannberg. 2014. Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application. In *SCAM'2014*. IEEE, 125–134.

Chris Lattner and Vikram Adve. 2003. *Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis*. Technical Report. U. of Illinois.

Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI '05*. ACM, 129–142.

Perf 2017. Perf Wiki. https://perf.wiki.kernel.org/index.php. (2017).

Robert E Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* 1 (1986), 157–171.

Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value λ-calculus Using a Stack of Regions. In *POPL '94*. ACM, 188–201. DOI: http://dx.doi.org/10.1145/174675.177855

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (1997), 109–176. DOI: http://dx.doi.org/10.1006/inco.1996.2613

Vlad Ureche, Aggelos Biboudis, Yannis Smaragdakis, and Martin Odersky. 2015. Automating Ad Hoc Data Representation Transformations. In *OOPSLA'15*. ACM, 801–820. DOI: http://dx.doi.org/10.1145/2814270.2814271

Harmen L. A. van der Spek, C. W. Mattias Holm, and Harry A. G. Wijshoff. 2010. Automatic Restructuring of Linked Data Structures. In *LCPC'09*. Springer, 263–277.

Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *ECOOP 2017 (LIPIcs)*. Schloss Dagstuhl. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2017.26

Zhenjiang Wang, Chenggang Wu, and Pen-Chung Yew. 2010. On Improving Heap Memory Layout by Dynamic Pool Allocation. In *CGO '10*. ACM.

Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Di Xu. 2012. On-the-fly Structure Splitting for Heap Objects. *ACM TACO* 8, 4 (2012), 26:1–26:20. DOI: http://dx.doi.org/10.1145/2086696.2086705

Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. 2009. Loci: Simple Thread-Locality for Java. In *ECOOP 2009 (LNCS)*. Springer, 445–469. DOI: http://dx.doi.org/10.1007/978-3-642-03013-0_21

Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.