University of London
Imperial College of Science, Technology and Medicine
Department of Computing

SHAPES: Easy and high-level memory layouts

Alexandros Tasos

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, 7th October 2022

Abstract

CPU speeds have vastly exceeded those of RAM. As such, developers who aim to achieve high performance on modern architectures will most likely need to consider how to use CPU caches effectively, hence they will need to consider how to place data in memory so as to exploit spatial locality and achieve high memory bandwidth.

Performing such manual memory optimisations usually sacrifices readability, maintainability, memory safety, and object abstraction. This is further exacerbated in managed languages, such as Java and C#, where the runtime abstracts away the memory from the developer and such optimisations are, therefore, almost impossible.

To that extent, we present in this thesis a language extension called SHAPES. SHAPES aims to offer developers more fine-grained control over the placement of data, without sacrificing memory safety or object abstraction, hence retaining the expressiveness and familiarity of OOP. SHAPES introduces the concepts of pools and layouts; programmers group related objects into pools, and specify how objects are laid out in these pools. Classes and types are annotated by pool parameters, which allow placement aspects to be changed orthogonally to how the business logic operates on the objects in the pool. These design decisions disentangle business logic and memory concerns.

We provide a formal model of SHAPES, present its type and memory safety model, and its translation into a low-level language. We present our reasoning as to why we can expect SHAPES to be compiled in an efficient manner in terms of the runtime representation of objects and the access to their fields.

Moreover, we present SHAPES-z, an implementation of SHAPES as an embeddable language, and `shapeszc`, the compiler for SHAPES-z. We provide our our design and implementation considerations for SHAPES-z and `shapeszc`. Finally, we evaluate the performance of SHAPES and SHAPES-z through case studies.

Statement of Originality

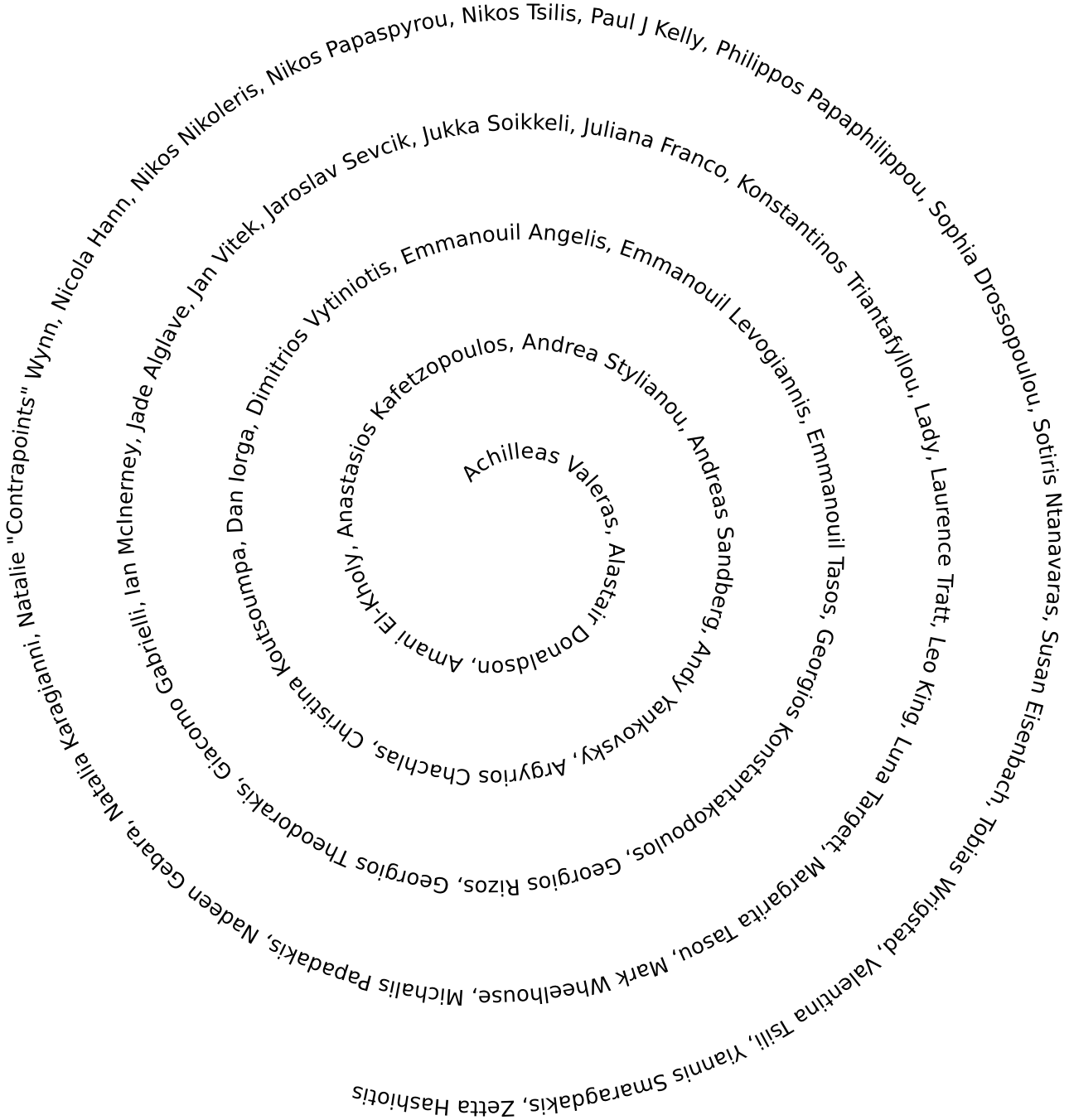
This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes. I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Alexandros Tasos

Copyright Declaration

Acknowledgements

I would like to express my gratitude to the following people:



This work has been supported by an EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS) Grant (Reference EP/L016796/1).

'Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious.'

Fred Brooks, The Mythical Man Month (1975)

Contents

List of Figures and Tables

List of Source Code Listings

Chapter 1

Introduction

Moore's law has resulted in significant improvements in the execution speed of CPUs in the last decades. DRAM speed, on the other hand, has certainly increased as well, but at a much slower rate than that of CPUs. A common belief that seems to be still held even today is that accessing main memory is "cheap"; this belief did have merit on older machines, where the speed of memory access used to rival that of the CPU, but it no longer necessarily holds. Nowadays, a fetch from main memory can be expected to take hundreds of CPU cycles to execute.

To mitigate this gap between CPU and DRAM speed, CPU architects have introduced *caches*: Caches aim to improve average memory access time; the more frequently data is served from the cache instead of DRAM, the fewer clock cycles will be spent on memory accesses on average [HP11].

This positive impact of caches on performance can be significant; fetching data from the cache can be an order of magnitude faster compared to fetching data from main memory. As such, algorithms that aim to be performant can be expected to need to pay more attention to how they layout and access data in memory; "cache-unfriendly" algorithm implementations can be expected to perform suboptimally compared to their equivalent "cache-friendly" counterparts.

However, changing the business logic so as to better utilise the cache can require signifi-

Level	Size	Latency	Line size	Associativity
L1d cache	32 KB	4 cycles	64 bytes	8-way set assoc.
L2 cache	1024 KB	14 cycles	64 bytes	16-way set assoc.
L3 cache	11 264 KB	68 cycles	64 bytes	11-way set assoc.
DRAM	—	259 cycles [2]	—	—

Table 1.1: Descriptions of the cache levels of an Intel Haswell i7-7820X 3.6 GHz CPU; numbers taken from [Pav14].

cant changes to the way data is laid out in memory; such changes usually are at odds with conventional approaches, such as object-oriented programming. To that extent, this thesis presents SHAPES, a language extension intended to allow developers to more easily accommodate changes data layout to make better use of the cache without having to deviate from the spirit of OO programming.

1.1 CPU caches

A cache is a smaller, faster memory that resides physically close to or within a CPU core and holds copies of data from portions of main memory that are deemed to be accessed frequently. The judgement on what data is to be kept in the cache at any point in time is performed in a manner transparent to the developer[1]. As stated earlier, the objective of introducing a cache is to provide data from the cache as frequently as possible (thus increasing the *cache hit* rate), rather than resorting to accessing DRAM.

A CPU design need not be limited to just one cache; a common approach used by CPU designers in order to further increase cache hit rates is to introduce multiple caches (*e.g.,* separate caches for data and instructions) or hierarchies of caches, with each level having a gradually larger size and a higher access latency; Figure 1.1 presents the specifications and execution latency of all cache levels of an Intel Skylake i7-7820X CPU: Indeed, the CPU has 3 cache levels, with the L1 data cache being the smallest yet fastest and the L3 data cache being the largest yet slowest.

[1] Quite a lot of CPU instruction sets provide instructions that provide *prefetching hints* to the CPU, but, given that they are hints, they need not be respected.

[2] Reported as "79 cycles + 50 ns", calculation into cycles assumes a 3.6 GHz operating frequency.

A noteworthy observation to make is that accessing the cache, even at the "outermost" level, can be orders of magnitude faster compared to accessing DRAM. For instance, on the CPU of Figure 1.1, a fetch from DRAM costs 259 cycles, whereas a fetch from L3 costs 68 cycles, which is at least a 3.5x improvement. As such, effective use of the cache can be extremely vital as a means of improving performance.

Spatial and temporal locality Two observations about most programs is that they exhibit *spatial* and *temporal locality*.

- *Spatial locality* refers to the observation that for a recently accessed memory location X, it is likely that memory locations within close proximity to X are also likely to be accessed in the near future.
- *Temporal locality* refers to the observation that data in recently accessed memory locations is likely to be accessed again in the near future.

Caches aim to exploit these observations in order to improve performance.

Cache organisation A cache contains multiple *cache entries* that may be valid or invalid (*i.e.,* data is present or absent, respectively) at any given time. Each (valid) cache entry stores the data of a contiguous block of memory; this is known as a *cache line*. All cache lines have a fixed size; the size of a cache line for the CPU of Figure 1.1, for instance, is 64 bytes for all levels of the cache hierarchy.

The presence of the concept of cache lines aids in exploiting *spatial locality*: If we are fetching a byte at address a, the entire cache line that a resides into will be kept in the cache as well, which will consist of data within proximity to a. Therefore, if we were to fetch the byte at the adjacent addresses $a-1$ or $a+1$, we can expect the respective values for these addresses to be most likely be present in the cache as well.

Because caches have a limited size, a *cache eviction policy* to determine what cache lines need to be kept in the cache at any time is necessary. Given the observation of *temporal locality*, *i.e.,*

data accessed recently can be expected to be also accessed in the near future, eviction policies such as *Least Recently Used* (LRU) seem quite appealing.

Indeed, a common algorithm often used for in CPU caches is N-way set associativity, which aims to approximate an LRU eviction policy, whilst being much cheaper to implement in hardware. In an N-way associative cache, a cache line will be placed in N potential entries, depending on its address; if all N entries are currently occupied, one of them is evicted beforehand.

1.2 Caches and Data Layouts

Writing cache-friendly programs can be expected to be more straightforward in unmanaged languages than in managed ones, as programmers have more control over data placement. For example, a programmer can allocate a large chunk of contiguous memory as a *pool* from which to "sub-allocate" objects that should be close in memory. For improved cache utilisation when iterating over many objects, programmers sometimes *split* a single array of objects into multiple arrays, each holding the values of a specific field of the objects; this is commonly referred to as an *Array-of-Structs* (AoS) to *Struct-of-Arrays* (SoA) transformation. Depending on which fields are being accessed together frequently, efficiency can be improved further by *clustering* values of several object fields together in one of the split arrays. Figure 1.2 presents an example of these concepts.

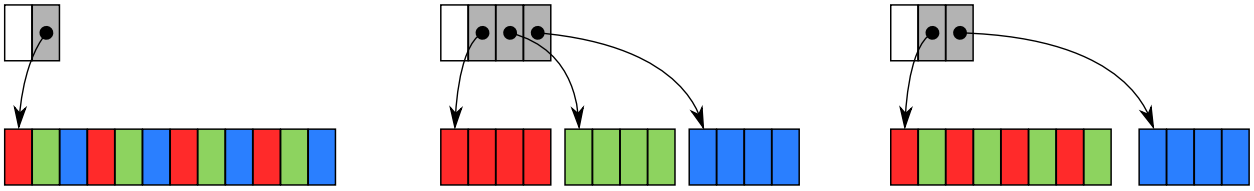
Applying these techniques in managed languages can be *difficult and not always possible*; in Java, for instance, where object types do not currently have value semantics, the memory allocator and garbage collector are not obliged to place objects in an array sequentially in memory. Moreover, splitting an array of objects into several arrays of fields *destroys object integrity and identity* (meaning it is no longer possible to have a pointer to the object, and referring to it must be done in an ad hoc manner), is *memory unsafe* (non-existing values can be created "out of thin air" by combining fields of different objects), and *loses automatic garbage* collection of individual objects. This affects managed and unmanaged languages alike: The authors of the WAVE++ particle simulator (1990), for instance, express their desire for

```
class Student {
  name: String;
  age: int;
  supervisor: Professor;
}
def avgAge(arr:Student[]){
  var sum = 0;
  for (i=0 .. arr.len())
    sum += arr[i].age;
  return sum / arr.len();
}
```

```
class StudentsSoA {
  name: String[];
  age: int[];
  supervisor: Professor[];
}
def avgAge(arr:StudentsSoA){
  var sum = 0;
  for (i=0 .. arr.age.len())
    sum += arr.age[i];
  return sum / arr.age.len();
}
```

```
class StudentsMixed {
  clu1: StudentsCluster1[];
  clu2: Professor[];
}

def avgAge(arr:StudentsMixed){
  var sum = 0;
  for (i=0 .. arr.clu1.len())
    sum += arr.clu1[i].age;
  return sum / arr.clu1.len();
}
class StudentsCluster1 {
  name: String[];
  age: int[];
}
```



(a) AoS representation (b) SoA representation (c) Mixed representation

Figure 1.2: Language-based field clustering: In-memory representation of class `Student`, AoS, SoA, and mixed

the ability to automatically apply such techniques in C++ without having to abandon OO programming [FWF+90].

To that extent, we present SHAPES, a language extension that aims to support efficient cache use in both managed and unmanaged languages by achieving such memory optimisations more easily, whilst also allowing the developers to write straight-forward OO code and without having to abandon key OO concepts.

SHAPES uses a type-based approach to enable these memory optimisations without having to sacrifice object integrity, memory safety, or garbage collection.

1.3 Thesis outline

This thesis focuses on SHAPES, the design behind its concepts, and an implementation of it. The outline of this thesis is as follows:

- § 2 presents SHAPES and how it implements memory optimisations via *pooling* and *clustering*.
- § 3 presents already existing work with respect to the concepts that SHAPES covers; we also compare the rationale for their design to that of SHAPES.
- § 4 formalises SHAPES; the formalism is done in terms of SHAPES^h, a high-level, user-facing calculus with *pooling* and *clustering*.
- § 5 presents SHAPES$_\ell$, a low-level intermediate representation, how SHAPES^h is translated into SHAPES$_\ell$, and the design decisions concerning SHAPES$_\ell$, such as the fact that multiple *specialised* SHAPES$_\ell$ functions need to be generated for each SHAPES^h function. We also justify why we expect these design decisions to allow translated SHAPES^h code to be performant. § 5 also presents meta-theoretic results for SHAPES^h and SHAPES$_\ell$: Type soundness, "memory safety", and bisimulation.
- § 6 presents SHAPES-z, our implementation of SHAPES as an embeddable domain-specific language and presents the design and implementation of `shapeszc`, the SHAPES-z compiler (§ 6.2).
- § 7 justifies the design of SHAPES through a sequence of case studies where we evaluate claims regarding performance and code readability.
- § 8 presents our conclusions and potential points of future work.

Chapter 2

SHAPES: Memory layouts

2.1 Getting into SHAPES

We now give an introduction to the design of the language extension SHAPES, using a simple running example: A class `Student`, with fields `name`, `age` and `supervisor` (which points to a `Professor`, the student's `supervisor`), as in Figure 1.2a. Assume that a method needs to access the `Students`' `ages` consecutively, as in `avgAge()`. To improve cache performance, we can perform a manual transformation from what is called an *Array-of-Structs* representation (AoS), shown in Figure 1.2a, into what is called a *Struct-of-Arrays* representation (SoA), shown in Figure 1.2b: Instead of an array of `Students`, we group the students' `names`, `ages`, and `supervisors` (Lines 2–4), each into their own array. The in-memory representation of the two (AoS and SoA) is depicted in Figure 1.2a and Figure 1.2b, respectively.

The `StudentsSoA` transformation shown in Figure 1.2b, however, is *a leaky and error-prone abstraction*. It sacrifices readability, maintainability and abstraction for performance:

- The look-and-feel of OO is lost: We are now effectively processing arrays of primitives instead of objects, *e.g.,* `arr.age[i]`. We may accidentally fetch the wrong parts of an object due to an off-by-one error (*e.g.,* evaluating `arr.name[i++]` and then `arr.age[i++]`), thus inadvertently "mixing" various unrelated object parts into one. References to objects

belonging to a pool have to be explicitly represented as an index and not as a regular object reference.

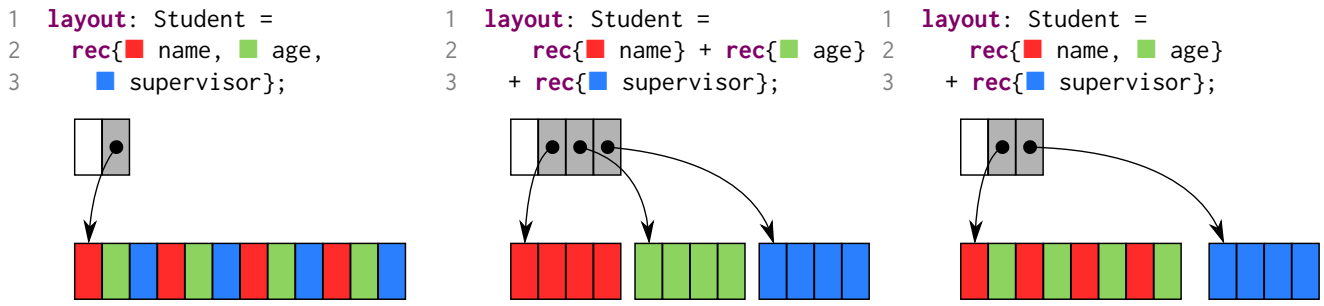
- Switching to different layouts is tedious and error-prone (*e.g.,* from Figure 1.2a to Figure 1.2b). Moreover, it is sometimes beneficial to use *mixed layouts* (§ 7), *e.g.,* we might want to group the values for `name` and `age` in one *cluster* (consisting of a chunk of allocated memory), and the values for `supervisor` in another such *cluster* (Figure 1.2c). This would require additional boilerplate code (Lines 12–15 of Figure 1.2c).

- There is no concept of automatic garbage collection of individual students in `StudentsSoA` and `StudentsMixed`, as we now have arrays of integers or pointers.

We can retain the OO look-and-feel and achieve automatic layout changes with a library. However, the state of the art of such libraries may require the introduction of syntactical extensions that do not compose elegantly with the rest of the underlying language and/or the sacrificial of core OO concepts such as encapsulation and object identity. Moreover, any flexibility with respect to automatic layout switching will be limited (*e.g.,* layout switching can only be performed on static arrays and/or will be limited to merely AoS vs SoA). Moreover, the issue of automatic garbage collection of pooled objects will still persist (more in § 3). As such, we propose a language extension-based solution to these issues with SHAPES.

SHAPES aims to support efficient cache use whilst enabling the programmer to write straightforward OO code and without having to abandon key OO concepts. SHAPES programmers add pool parameters to class definitions which allow objects to be flexibly placed in different pools; the business logic of classes is thus oblivious to the layouts being used and imposing a specific layout is not an onerous task. Programmers, who are aware of how they access the relevant data, write layout annotations and declare pools of specific layouts to achieve the best possible cache usage for the business logic in question.

We now give a gradual introduction to SHAPES, in six stages. Each stage extends and refines the previous one.



(a) AoS representation (b) SoA representation (c) Mixed representation

Figure 2.1: Stage 1: Language-based field clustering

2.2 Stage 1: Language-based field clustering

As a first approximation, we place all class instances inside one unique, implicit pool for that class. That is, in Stage 1, constructing a `Student` object will allocate it inside the unique, implicit pool corresponding to class `Student`.

An optional **`layout`** declaration specifies how class instances are laid out inside that implicit pool. A **`layout`** declaration splits the class' fields into *clusters*. Each cluster specifies the fields' values to be stored together and in what order (so as to *e.g.,* eliminate any padding introduced due to alignment). Omitting a layout declaration implies an *Array-of-Structs (AoS)* layout. Thus, using the "standard" code from Figure 1.2a and choosing one of the layouts from Figure 2.1 we can obtain any of the respective representations as in Figure 2.1a, Figure 2.1b, or Figure 2.1c.

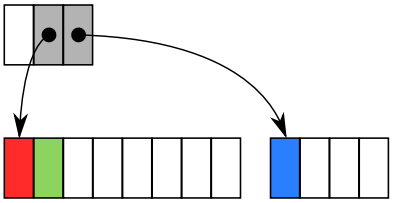
2.3 Stage 2: Use as many pools as needed

Not *all* objects of one class have to be placed in the *same* pool: For example, the nodes of two different binary trees would be better placed in different pools, and sometimes it is beneficial to use different memory layouts for objects of the same class (*cf., Currency* case study in § 7.4).

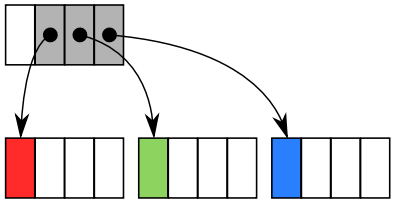
We add explicit declarations for **`pools`** and allow many *named* **`layout`** declarations for each class. As an example, in Figure 2.2 (which builds on class `Student` from Figure 1.2a), Lines 1–5 declare two layouts for `Student`: `StudentL1` clusters fields `name` and `age` together and places `supervisor` in its own cluster; `StudentL2` is a *Struct-of-Arrays (SoA)* layout.

```
1  layout StudentL1: Student =
2    rec{■ name, ■ age} + rec{■ supervisor};
3
4  layout StudentL2: Student =
5    rec{■ name} + rec{■ age}
6      + rec{■ supervisor};
```

```
7    ...
8    pools pStu1: StudentL1, pStu2: StudentL2;
9
10   s1 = new Student: pStu1;
11   s2 = new Student: pStu2;
12   ...
```



(a) s1 at pStu1



(b) s2 at pStu2

Figure 2.2: Stage 2: As many pools as needed

Pools are created at runtime—Line 8 creates two new pools: `pStu1` with layout `StudentL1`, and `pStu2` with layout `StudentL2`. We must now *specify* the pool in which to place a newly created object; Lines 10–11 construct two new `Student` objects, `s1` and `s2` and place them inside `pStu1` and `pStu2`, respectively, in accordance with their respective pools' layouts. Execution of Lines 8–11 will result in the memory layout shown in Figures 2.2a and 2.2b.

Notice that SHAPES supports reference semantics and not copy semantics, thus the concept of object identity is preserved. Even though pools contain the fields of the objects they contain, all object identifiers are treated as references. For example, `s1` and `s2` (Lines 10–11) are *references* to the newly constructed objects in pools `pStu1` and `pStu2`. Similar to languages such as Java [GJS+14], it is possible for two variables to alias to the same object. Additionally, objects can only be placed into a pool when they are constructed, but are not copied or moved into and out of a pool. Implicit copy construction/assignment (à la *e.g.,* C++ [ISO12]) could be added as an extension to SHAPES.

2.4 Stage 3: Use pools only when you need them

In some cases, there is no incentive for using pools (*e.g.,* rarely executed, non-performance-critical code). In most OO languages, objects are allocated on the heap, with no placement guarantees. SHAPES supports this through the "special" pool **none**, which can contain objects

```
1 s3 = new Student: none;
```

Figure 2.3: Stage 3: Pools only when needed

of any class, and no layout is applied to the objects. This allows gradual introduction of pooling and clustering into a project. For example, in the code of Figure 2.3, `Student s3` is created "inside" the **none** pool, hence it will be placed on the heap.

2.5 Stage 4: Flexible object placement

If we want to allow the creation of binary trees whose `Nodes` are placed in per-tree pools, we will need to provide a way so that functions manipulating these `Nodes` know the pool where to place newly generated `Nodes`. To support this in SHAPES, we supply classes with pool parameters.

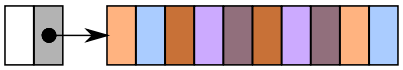
Consider, for example, binary trees of `Professors`; the relevant definitions appear in Lines 1 and 2 in Listing 2.1. Class `Professor` has one pool parameter, `pProf`, which stands for the pool which contains the corresponding `Professor`. Class `Node` has two pool parameters: `pNode` is the pool of the corresponding `Node`, and `pProf` is the pool which contains the `Professors`. That is, the first pool parameter always corresponds to the pool where **this** is located (pool `pNode` for `Nodes` and `pProf` for `Professors`, respectively, in our case).

```
1  class Professor<pProf> { ... }
2  class Node<pNode, pProf> {
3    ...
4    def addLeft( ... ) {
5      ...
6      p = new Professor<pProf>;
7      this.left = new Node<pNode, ...>;
8      ...
9    }
10 }
```

Listing 2.1: Stage 4: Flexible Object Placement

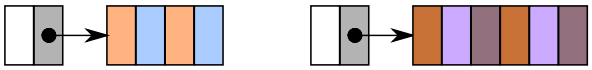
Method `addLeft()` (Line 4 in Listing 2.1) now knows that the new `Professor` (Line 6) is to be placed in pool `pProf`, and the new `Node` is to be placed in pool `pNode` (Line 7). This allows the developer to ensure that all `Nodes` and all `Professors` in one tree are placed in exactly one `Node` and one `Professor` pool, respectively.

```
1 class Node<pNode, pProf>{
2   def addLeft( ... ) {
3     ... new Professor<pProf> ... ; // OK!
4     ... new Node<pProf, ...> ... ; // BUG
5   }
6 }
```



(a) Non-uniform pool

```
1 class Node<pNode:[Node],pProf:[Professor]>{
2   def addLeft( ... ) {
3     ... new Professor<pProf> ... ; // OK!
4     ... new Node<pProf, ...> ... ; // ERR
5   }
6 }
```



(b) Uniform pool

Figure 2.4: Stage 5: Pool bounds

2.6 Stage 5: Make it safe with uniform pools

Allowing code to specify the pool where objects are placed in can have unintended consequences, because objects of different types can be potentially placed inside the same pool. This is undesireable, because the layout of a pool corresponds to only one object type.

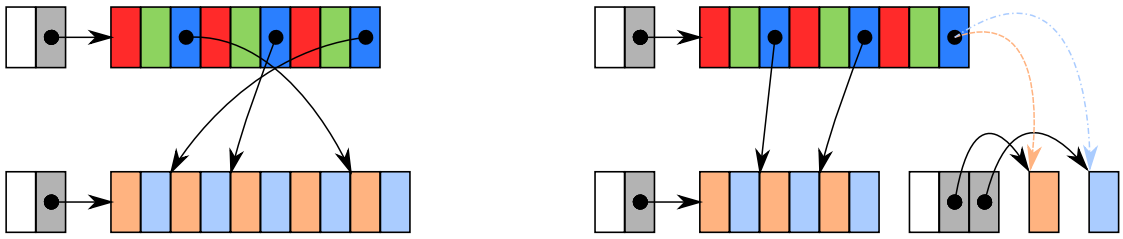
As such, we need to require pools to be *uniform*. We enforce *pool uniformity* by introducing the concept of *pool bounds*, which specify the type of objects a pool can contain.

As an example, in Figure 2.4a, uniformity of `pProf` would be violated after running Lines 3–4. This is because a `Professor` and a `Node` would be placed inside the same pool and the layout of `pProf` can only accommodate objects of type `Professor`, thus violating any semblance of type safety. We prevent this from occurring by adding bounds in Line 1 of Figure 2.4b. These bounds specify that `pProf` can only contain instances of `Professors`, hence we deduce that Line 4 in Figure 2.4b is erroneous.

Pools created inside methods must always specify a layout, hence their bound can be easily deduced.

2.7 Stage 6: Make it fast with homogeneous pools

So far, we argue all of our design decisions have been uncontentious: They are either concerned with providing a reasonable feature set to the developer or with preventing an unsound situation from arising. Our decision to enforce the concept of *pool homogeneity*, however, is done as a



(a) Homogeneous `Student` pool (b) Heterogeneous `Student` pool

Figure 2.5: Homogeneous and heterogeneous `Student` pools

trade-off: We further restrict what constitutes a valid SHAPES program on the expectation of gaining additional performance guarantees thanks to these new restrictions.

A uniform pool is *homogeneous* if the corresponding fields of all its objects point to objects in the same pool; that is, a pool `p` is *homogeneous* if for any objects `o1` and `o2` belonging to `p` and for any field `f`, it holds that `o1.f` and `o2.f` are either both in the same pool or on the heap (*i.e.*, in pool `none`). A pool that is not *homogeneous* is *heterogeneous*.

In Figure 2.5a, we see a homogeneous pool of `Students` whose `supervisor` fields all point to objects in the same pool. In Figure 2.5b, we see a heterogeneous pool of `Students`: The `supervisors` of the first two `Students` point to `Professors` in a different pool to that of the last `Student`.

Such heterogeneity could be caused by the following code, wherein we have a `Student` pool `pStu1` and two `Professor` pools `pProf1` and `pProf2`:

```
1  s1 = new Student<pStu1, ...>;
2  s2 = new Student<pStu1, ...>;
3  s1.supervisor = new Professor<pProf1>;
4  s2.supervisor = new Professor<pProf2>;
```

If we were to support heterogeneous pools, we would have to make the following design decisions regarding the runtime and *suffer a performance penalty*:

- In a naive implementation, a reference to a pooled object would consist of a reference to the pool containing the object and a reference to the object inside the pool itself. This would be rather wasteful on RAM and cache. In § 5.5, we show that, by exploiting homogeneity, a reference to a pooled object can have the same size as that of a pointer.

```
1  class Professor<pProf: [Professor<pProf>]> {
2      name: String;
3      ssn: String;
4  }
5  class Student<
6      pStu: [Student<pStu, pProf>],
7      pProf: [Professor<pProf>]> {
8      name: String;
9      age: int;
10     supervisor: Professor<pProf>;
11 }
12 layout ProfL: Professor = ...;
13 layout StudentL: Student = ...;
14 ...
15 pools pStu1: StudentL<pStu1, pProf1>,
16       pProf1: ProfL<pProf1>;
17       pProf2: ProfL<pProf2>;
18 s1 = new Student<pStu1, pProf1>;// OK!
19 s2 = new Student<pStu1, pProf1>;// OK!
20 s3 = new Student<pStu1, pProf2>;// ERR
21 p1 = new Professor<pProf1>;
22 p2 = new Professor<pProf2>;
23 s1.supervisor = p1; // OK!
24 s2.supervisor = p2; // ERR
25 ...
```

Listing 2.2: Stage 6: Enforcing uniformity and homogeneity via pool bounds

- Since the layout of the corresponding pool for a pooled object cannot be necessarily deduced at compile time, heterogeneity would imply dynamically looking up the layout information of that specific pool; this hampers performance and requires the developer to not assume that `x.f` is a "cheap" operation. In § 5.5, we show that the layout of a pool can be statically known thanks to homogeneity, hence such dynamic lookups are unnecessary.

To enforce pool homogeneity, we adapt ideas from C++ templates [ISO12], Java Generics [GJS+14], and Ownership types [CÖSW13], as follows:

- As in Stages 4 and 5, classes have several *formal pool parameters*. These correspond to the pools containing the objects pointed to by (some of) their fields. The first pool parameter also corresponds to the pool where **this** is stored.

- Object types, class instantiations, pool bounds, and pool creations must supply a pool argument per formal pool parameter of their respective class. Just like formal pool parameters, the first pool parameter specifies the object the pool is allocated into; during pool creation, the first pool parameter is the pool itself being created.

- If a pool `p1` has a bound of the form `[C<p1, ..., pn>]`, then all objects residing in `p1` *must also* have the type `C<p1, ..., pn>`. In § 4.3.1, we show how we use this restriction to enforce homogeneity in a static manner.

As an example, consider the code of Listing 2.2. Similar to Listing 2.1, class `Professor` has one pool parameter, `pProf`, and class `Student` has two pool parameters, `pStu` and `pProf`. The bounds of these classes (Lines 6–7 for class `Student` and Line 1 for class `Professor`) are now decorated with pool arguments. Pool arguments are supplied at pool creation (Lines 15–17), as well as at object creation (Lines 18–22).

For Listing 2.2, the pool bounds help enforce pool homogeneity as follows: Line 10 mandates that the `supervisor` *must belong* to the pool referenced by the formal pool parameter `pProf`. Thus, when constructing pool `pStu1` (Line 15), we substitute the formal pool parameters of `Student` with `pStu1`, `pProf1` and deduce that the `supervisor` of any `Student` placed in `pStu1` must have type `Professor<pProf1>`, *i.e.,* the supervisor *must* reside in pool `pProf1`. However, Line 20 specifies that the `supervisor` of `s3` has type `Professor<pProf2>`, hence it must reside in `pProf2`. This would break homogeneity and is flagged as an error by our type system.

By using similar reasoning, we deduce that Line 24 would also break homogeneity, given that `p2` was constructed in pool `pProf2` and we mandate that the `supervisor` of `s2` be located inside pool `pProf1`.

2.8 Conclusions

We have reached a design which is sound (§ 4), supports pools and goes beyond AoS/SoA, and we argue that it is flexible and transparent.

Pool parametricity is, as mentioned, similar to Java Generics [GJS+14] in that pool parameters have bounds, and these bounds are types which may contain pool arguments. It differs from Java Generics in that the pool arguments are not types; instead, they are entities generated at runtime. It also differs as SHAPES supports three kinds of types: *Object types* determine the class of the object, and the pools of the object and its fields; *layout types* determine the class of objects stored in pools of that layout and how the objects in the pool are split into *clusters*; *pool bounds* characterise pool parameters to classes and specify the pool an object referenced by field f will reside in (if any). Finally, SHAPES enforces homogeneity, a concept that is not

found in Generics. For the rest of this thesis, when we talk about pools, we will be referring to homogeneous pools.

Chapter 3

Related work

We will now present the existing work that has been performed with respect to pooling and clustering and evaluate the design of SHAPES (§ 2) against this body of work. We will also compare the design of SHAPES against each such piece of work where possible and state the benefits SHAPES brings to the table, as well as the parts where SHAPES may be lacking and/or can be incorporated as possible future work.

3.1 Frameworks and libraries

3.1.1 AoS to SoA

The C++ world is rich with libraries that transform an array of an AoS to an SoA layout. Almost all of these libraries operate on fixed size arrays. We present some of these libraries:

ASX ASX [Str11] is a C++ library that lets a developer switch between an AoS and an SoA format for an array whose size is fixed at compilation. Listing 3.1 presents an ASX example. To use ASX, the developer needs to modify the definitions of the class/struct fields in a manner similar to that of Lines 1–4 for `Vec3`.

```
1  static constexpr size_t SIZE = 1024;
2  template<ASX::ID t_id = ASX::ID_value>
3  struct Vec3 {
4      typedef ASX::ASAGroup<Type1, t_id> ASX_ASA;
5      union { float x; ASX_ASA dummy1; };
6      union { float y; ASX_ASA dummy2; };
7      union { float z; ASX_ASA dummy3; };
8  };
9
10 ASX::Array<Vec3, SIZE, ASX::SOA> vecs;
11 vecs[5].x = 0.5;
```

Listing 3.1: ASX example

Line 10 defines `vecs` as an ASX array that contains `SIZE` elements of type `Vec3` and whose fields will be laid out in an SoA layout. Accessing and modifying the fields of the `Vec3` objects within `vecs` (Line 11) is performed in a uniform manner regardless of whether an AoS or an SoA layout is used.

Due to its implementation, ASX imposes the constraint that the types of the fields must have the same size and alignment. If this constraint cannot be satisfied, ASX suggests a "workaround" wherein multiple fields are "grouped" within a nested `struct` type; this resembles the concept of *clustering* in SHAPES (§ 2.1).

ASX also defines `ASA::Vector`, which is a dynamically resizeable array. An `ASA::Vector` does not correspond to a container that has an SoA layout similar to that of SHAPES (§ 2.1); rather it is a hybrid layout between AoS and SoA, which bears resemblance to what is called as an *Array-of-Structs-of-Arrays* (AoSoA) layout: In an AoSoA layout, container elements are divided into equisized chunks of size M (with the exception of a chunk that holds the remaining elements); the chunks are laid out sequentially in memory and the elements within each chunk are laid out in an SoA manner.

When comparing ASX to SHAPES, we observe that SHAPES layouts are not bound by the limitations imposed by ASX in terms of size and alignment; in a SHAPES layout declaration, fields of a type are permitted to be grouped and ordered in any arbitrary manner.

Additionally, SHAPES does not require modifications to the fields of a class type. However,

```
1  SOAX_ATTRIBUTE(id, 'N');
2  SOAX_ATTRIBUTE(pos, 'P');
3  SOAX_ATTRIBUTE(vel, 'V');
4  SOAX_ATTRIBUTE(mass, 'M');
5
6  typedef std::tuple<
7      id<int, 1>,
8      pos<double, 3>,
9      vel<double, 3>,
10     mass<float, 1>
11 > Particle;
12
13 Soax<Particle> soax(42);
14
15 soax.id(23) = 0;
16 soax.pos(23, 0) = 3.14;
17 soax.posArr(0) =
18     soax.velArr(1) - soax.velArr(2);
19 soax.resize(100);
20
21 auto particle = soax.getElement(7);
22 particle.id() = 42;
23 particle.pos(0) = 3.14;
24 soax.push_back(particle);
```

Listing 3.2: SoAx example [HL18]

classes do need to be enriched with pool parameters. This raises a possible future direction for SHAPES wherein pool parameters become optional to an extent, so that pools and layouts can be added to an existing codebase in a gradual manner.

Another noteworthy point of extension for SHAPES would be the addition of support for AoSoA layouts. AoSoA layouts are certainly an interesting approach to layouts and a potentially noteworthy addition, as only one cluster needs to be allocated, thus making memory allocation simpler.

SoAx SoAx [HL18] is a C++ library intended for HPC code. It lets the developer declare a structure type via a template metaprogramming scheme and then construct an SoA "array".

Listing 3.2 presents an SoAx example. The developer initially declares the attributes that are to be used with SoAx (Lines 1–4). To define a structure type (*e.g.,* `Particle` in Line 11), the developer specifies what properties the type uses, their underlying type, as well as their arity (Lines 7–10). The `pos` attribute in `Particle`, for instance (Line 8) is defined to consist of three values of type `double`.

An array of `Particles` where the `Particles`' properties are laid out in an SoA layout is defined in Line 13. The size of this array is not fixed upon construction; array `soax` can be resized to accommodate more elements (Line 19) or a new object can be inserted into it (Line 24).

SoAx enriches its arrays accessor functions to access and modify the fields of objects (Lines 15–

```
1  class Car;
2  class Cell;
3  class ProducerCell;
4
5  using AllocatorT = SoaAllocator<
6    kNumObjects,
7    Car,
8    Cell,
9    ProducerCell
10 >;
11 class Cell: public AllocatorT::Base {
12 public:
13   declare_field_types(
14     Cell,
15     // incoming_
16     DeviceArray<Cell*, kMaxDegree>,
17     // outgoing_
18     DeviceArray<Cell*, kMaxDegree>,
19     Car*, // car_
20     int, // max_velocity_
21     int, // current_max_velocity_
22     int, // num_incoming_
23     int, // num_outgoing_
24     float, // x_
25     float, // y_
26     bool // is_target_
27   )
28
29 private:
30   SoaField<Cell, 0> incoming_;
31   SoaField<Cell, 1> outgoing_;
32   SoaField<Cell, 2> car_;
33   SoaField<Cell, 3> max_velocity_;
34   SoaField<Cell, 4> current_max_velocity_;
35   SoaField<Cell, 5> num_incoming_;
36   SoaField<Cell, 6> num_outgoing_;
37   SoaField<Cell, 7> x_;
38   SoaField<Cell, 8> y_;
39   SoaField<Cell, 9> is_target_;
40 };
```

Listing 3.3: DynaSOAr example class

16). It also allows developers to access the objects themselves (Line 21), albeit via value semantics (*i.e.,* a copy of the object is returned). Additionally, the fields of objects stored in an SoAx array can be accessed collectively and operations can be applied on them in a collective manner (Lines 17–18); this bears resemblance to `std::valarray` in C++ [ISO12], as well as Intel's Array Building Blocks (Intel ArBB) project [NSL+11].

Compared to SoAx, fields can be grouped and ordered arbitrarily in a SHAPES layout declaration, whereas SoAx only supports SoA layouts. Additionally, the design of SHAPES makes accessor methods unnecessary, since fields of an object can be accessed in an identical manner, regardless of whether the object belongs to a pool or not.

With respect to points of extension, support for collective operations in a SHAPES implementation is a feature that we argue is worth considering.

Ikra-Cpp and DynaSOAr Ikra-Cpp [SM18] is a C++/CUDA library that allows switching between AoS and SoA for static arrays. It provides some object-oriented capabilities (constructors, object-specific methods); the developer needs to annotate their classes and use Ikra-Cpp

specific primitive types.

DynaSOAr [SM19] builds on top of Ikra-Cpp; it implements *dynamic object sets* that use an SoA layout; like SHAPES pools, arbitrarily many objects can now be allocated inside these object sets.

Listing 3.3 presents an example of a DynaSOAr class definition[1]. We initially specify an allocator type (Line 5) that will correspond to our class and then make our class inherit from a type associated to that allocator type (Line 11). In our allocator type definition, we also specify the maximum size of the pool (Line 6), as well as what types of objects class `Cell` will be referencing (Lines 7–9).

Within the body of class `Cell`, we need to specify the types of the fields to DynaSOAr (Lines 16–27) and then declare *proxy fields* that correspond to the class' actual fields (Lines 30–39).

Compared to Ikra-Cpp, SHAPES does not require the use of nonstandard primitive types. Additionally, the duplication of field information in DynaSOAr (compare Lines 16–27 and Lines 30–39) is a non-issue for SHAPES, as pooling and clustering are incorporated into the language.

A major benefit of DynaSOAr that should definitely be considered for future work in SHAPES is the addition of multithreaded allocation/garbage collection.

Julia package A package [Kor15] for the Julia language allows transformation of a fixed size (at runtime) array into an SoA structure via metaprogramming. It only supports `isbits` types, *i.e.,* immutable scalar types with no references to other objects; SHAPES does not impose such limitations in its design.

Python column-based databases In the dynamic language world, [MHR+15] describe and implement an object layout for column-based databases intended to be easily optimisable by the PyPy JIT. Access and traversal of the objects in the database is achieved through an iterator

[1] Taken from https://github.com/prg-titech/dynasoar/blob/a4f956cd13160eb566e8a0caef6aad1e59c26da9/example/traffic/dynasoar/traffic.h, licensed under the MIT License.

```
View node {
    Field {x: d}
    Field {y: d}
    Field {z: d}
}
```

Listing 3.4: TALC field specification file

interface. Supporting such an iteration scheme can be achieved in SHAPES via extensions (*e.g.,* our implementation of SHAPES called SHAPES-z, *cf.,* § 6). Moreover, as we will see in § 5.5, SHAPES makes dependency on a JIT redundant.

3.1.2 Clustering

OP2 OP2 [GMS+11] is a C++ library intended for computations on unstructured grids and is mainly focused on easing parallelisation of such applications (via, *e.g.,* MPI [for15], OpenMP [Boa18], CUDA [NC22]). OP2 mainly attempts to tackle the issue of executing a kernel over a set of data in parallel in a declarative manner. OP2 also allows the developer to perform a limited form of clustering (only fields of the same type can be clustered together) whereas the SHAPES design has no such constraint.

Additionally, OP2 features execution plans: The developer specifies what fields will be accessed in a kernel and how. Then, during execution of a computational kernel, an execution plan will partition objects so that when we run the kernel in question over two objects residing in the same partition, then there will be no data races. SHAPES offers no such feature, considering its general purpose nature. However, concurrency is certainly a noteworthy point for future extensions.

We discuss OP2 in more detail in § 7.2.

TALC TALC [KJQ08] is a C language extension that allows the clustering of static, fixed size arrays of `struct` types. The developer writes object schemas (similar in nature to layout declarations) and their business logic in TALC. Then, by specifying which schema to use, a

TALC compiler generates C code in accordance to the schema specified. This C code has to be manually generated anew when switching to a different layout. Later work [SKK+15, SKK+13] extends TALC with automatic selection of the most efficient layout via a greedy algorithm.

Listing 3.4 presents a TALC schema file. Lines 2–4 specify the arrays in the C source code that must be clustered together (`x`, `y`, and `z`), as well as the underlying type of the array (in this case, `d` corresponds to the type `double`). Given Listing 3.4, TALC will modify a given piece of code so as to perform clustering on the arrays specified.

While SHAPES currently offers no automatic layout selection, it is not constrained to fixed size arrays.

3.1.3 Object representation transformations

For Scala, [UBSO15] proposed an extension for automatic changes to the data layout where a developer defines transformations and the compiler applies the transformation during code generation.

3.2 Automatic program transformation

3.2.1 Pooling

[CKJA98] attempt to reduce cache misses by automatically partitioning objects into pools of popular and unpopular objects. Later work by [LA03, LA05] for C and C++ leverages static analysis instead of profiling to partition objects into pools.

3.2.2 Clustering

Part of the work done by [HBM+04] consists of automatically determining "hot" and "cold" fields of an object and clustering such objects in such a manner that "hot" fields are placed in

the same cluster. A SHAPES developer will have to explicitly annotate their code; an analysis tool can then determine the "hot" and "cold" fields and then produce the relevant layout (which the developer can then use).

[PPSdM01] present a greedy algorithm for determining an optimal clustering strategy tuned for embedded applications. Such automatic clustering is performed only on arrays of structures with a size that is fixed at compile time. Clustering in SHAPES has to be performed manually, but, unlike fixed size arrays, the sizes of pools is not fixed.

3.3 Programming languages

3.3.1 Class parameterisation

SHAPES types have been influenced by Ownership types [CÖSW13], using pool parameters instead of ownership contexts. Unlike Ownership types, our type system allows cycles between pools.

The concept of bounds and well-formed types is drawn from Featherweight Generic Java [IPW01], although our formalism does not have any concepts of polymorphism.

Similar to pooling, Petersen et al. [PHCP03] describe a model that uses ordered type theory to allow a runtime to coalesce multiple calls to the allocator.

Class parameterisation has also been used in the context of region based memory management, such as Cyclone [GMJ+02], the Rust language [KN18], where types are permitted to be parameterised over lifetimes, and Pony [CDBM15], where types are permitted to be parameterised over reference capabilities.

```
1 struct Vec3 {
2     float x, y, z;
3 };
4 typedef soa<4> Vec3 Vec3x4;
```

Listing 3.5: ISPC SoA feature

```
1 struct Vec3x4 {
2     float x[4], y[4], z[4];
3 };
4
```

Listing 3.6: Equivalent C++ declaration of Listing 3.5

3.3.2 ISPC

The Intel ISPC (Implicit SPMD Program Counter) language [PM12] aims to make it easier for developers to write code that can take advantage of SIMD instruction sets found in today's architectures (*cf.*, § A).

ISPC provides (partial) support for SoA layouts, by providing a keyword called `soa<n>`. This keyword transforms a structure type into a structure type where each member is an array of size `n` with the same type as the respective member in the existing definition.

Listing 3.5 presents an example of the `soa` feature of ISPC applied to struct `Vec3` (Line 4); Listing 3.6 presents the equivalent type definition in C++.

Notice that an `soa` ISPC type does not correspond to a "true" SoA layout; rather, it corresponds to an *Array-of-Structs-of-Arrays* (AoSoA) layout.

ISPC is presented in more detail in § A.3.

3.3.3 Sierra

Like ISPC (§ 3.3.2), Sierra [LHH14] also aims to allow developers to write code that utilises SIMD in an efficient, high-level manner, albeit as an extension to C++ (rather than as a "standalone" language).

Sierra enriches the C++ type system with vector types, which are dubbed in Sierra as `varying` types. A `varying` type consists of an underlying type T and the number n of elements in the vector. For instance, `float varying(4)` represents a vector type that consists of 4 `float`s.

```
1 struct Vec3f {
2     float x;
3     float y;
4     float z;
5 };
6 typedef Vec3f varying(4) Vec3fx4;
```

Listing 3.7: Sierra `varying` modifier on a `struct` type

```
1 struct Vec3fx4 {
2     float varying(4) x;
3     float varying(4) y;
4     float varying(4) z;
5 };
6
```

Listing 3.8: Equivalent definition for Listing 3.7

In the case where the underlying type T is a primitive or pointer type, the corresponding type `T varying(n)` is represented as an n-tuple of elements of type T at runtime. In the case where T is a struct type, however, the derived type is a struct type where the `varying` keyword is recursively applied to all of the struct type's members.

Listing 3.7 presents an application of the `varying` keyword on type `Vec3f` (Line 6); Listing 3.8 presents its equivalent definition. This type transformation resembles the concept of *mapped types* present in Typescript [Cor22]. This allows Sierra to express similar layout transformations as ones provided by ISPC's `soa` keyword.

Sierra is presented in more detail in § A.4.

3.4 Heap partitioning

IBM's X10 language [SBP+12] partitions the object heap into *places*, which are intended to assist the developer in taking better advantage of memory locality, as well as provide future support for distributed and heterogeneous computing. In X10, the current continuation is associated with a place and it can access objects from that place only. Objects can be copied between places through a *place-shifting* operation, which, given a set of roots, it copies a subset of the object graph into the designated place via serialisation.

Similar ideas are presented in Offload [CDD+10], which aims to allow developers to allow execution of C++ code on heterogeneous cores, such as accelerators. The main use case of Offload is to better utilise the SPE (Synergistic Processing Elements) cores of the Cell Broadband Engine

CPU [Hof05] with relatively few changes to an existing codebase.

In the realm of Ownership Types [CÖSW13], some works have permitted splitting data in the heap *conceptually* (hence they do affect in-memory representation), to calculate the effects of reading and writing to data [BAD+09] or reason about thread-local data [WPM+09].

Inference has been used successfully in this context *e.g.,* by Jaber et al. [JK17] for ownership-based heap partitioning.

Franco and Drossopoulou use annotations to control placement on a NUMA node granularity [FD15] with the aim of improving program performance.

3.5 Our earlier work

The SHAPES design builds on and extends prior work [FWD16, FHW+17, FTD+18, TFW+18, TFD+20]:

OHMM [FWD16] is similar to *Stage 4* (§ 2.5). That is, pools are not uniform: Objects of a specific type are placed in the class-specific *subpool* of that pool. Moreover, similar to *Stage 1* (§ 2.2), each class can have up to one layout; all subpools corresponding to that class will adhere to the class' layout.

SHAPES ideas were presented in [FHW+17], which corresponds to *Stage 5* (§ 2.6). The paper contains neither the complete language design, nor a formal model. Pools are not homogeneous, hence runtime type information is necessary for field access and object construction.

[TFW+18] presents extensions to the SHAPES model to add support for dynamically allocated pool-backed arrays with value semantics and a SIMD environment similar to that of [PM12, LHH14]; we further expand on SIMD in § A).

3.6 Conclusion

In this chapter, we have presented existing work that has been done in terms of pooling and clustering. Additionally, we compared this body of work to SHAPES and observed that even with the partial overlap of SHAPES with it, SHAPES is still capable of bringing additional benefits to the table, especially in terms of usability. As such, we argue that examining and delving into the concepts of SHAPES is worth pursuing.

Additionally, we have also indicated points for future extension. These include but are not limited to support for additional layouts (*e.g.,* AoSoA), support for concurrency (*e.g.,* performing concurrent allocations within a pool), and automatic layout selection (via *e.g.,* heuristics or machine learning). We hope that a future version of SHAPES will incorporate some of the above features.

Chapter 4

SHAPES^h: A formalism of SHAPES

In § 2, we presented high level language constructs that simultaneously allow a high-level business logic and easier fine grained control over data placement.

To provide these constructs and ensure they can be implemented in a manner we expect to be efficient, we developed SHAPES^h, a high-level calculus that provides the appropriate constructs (§ 4), and SHAPES$_\ell$, a low-level intermediate representation designed with efficiency in mind for implementing these constructs (§ 5). § 5.5 presents the translation of SHAPES^h code into SHAPES$_\ell$.

SHAPES^h is a minimal object-oriented calculus with no inheritance, and augmented with pools. The most striking feature compared to other OO languages is that SHAPES^h types are parameterised with pool variables as parameters (§ 4.3); it is thanks to these pool parameters that the SHAPES^h type system can statically enforce pool *uniformity* and *homogeneity* (§ 2.6, § 2.7).

Representation of entities in SHAPES^h also deviates from a typical OO calculus: Objects carry ghost information regarding pools, but the pools themselves do not affect object placement or perform any sort of clustering. This simplifies SHAPES^h and demonstrates the argument that regardless of the pooling and clustering scheme being used, SHAPES allows developers to write their business logic with a simpler, object-oriented mental model in mind, where all objects (both standalone and pooled) are treated uniformly. Moreover, we only use SHAPES^h

$$
\begin{array}{rcll}
\mathit{prog} & ::= & \mathit{cd}^*\ \mathit{ld}^* & \mathit{Program} \\
\mathit{cd} & ::= & \textbf{class}\ C\langle(p\colon [C\langle p^+\rangle])^+\rangle\ \ \{\ \mathit{fd}^*\,\mathit{md}^*\ \} & \mathit{ClassDecl} \\
\mathit{fd} & ::= & f\colon t\ ; & \mathit{FieldDecl} \\
\mathit{md} & ::= & \textbf{def}\ m(x\colon t)\colon t\ \{\mathit{localPools}\ ;\ \mathit{localVars}\ ;\ \mathit{stmts}\} & \mathit{MethodDecl} \\
\mathit{localPools} & ::= & \textbf{pools}\ (p\colon L\langle ps\rangle)^*; & \mathit{LocalPools} \\
\mathit{localVars} & ::= & \textbf{vars}\ (x\colon t)^* & \mathit{LocalVars} \\
\mathit{stmts} & ::= & e \mid e\ ;\ \mathit{stmts} & \mathit{Statements} \\
e & ::= & \textbf{null} \mid x \mid \textbf{this} \mid \textbf{new}\ t \mid x.m(x) \mid x.f \mid x.f = x \mid x = e & \mathit{Expression} \\
t & ::= & C\langle ps\rangle & \mathit{ObjectType} \\
\mathit{ld} & ::= & \textbf{layout}\ L\colon [C] = (\textbf{rec}\ \{f^+\};)^+ & \mathit{LayoutDecl} \\
\mathit{np} & ::= & p \mid \textbf{none} & \mathit{PoolVariableOrNone} \\
ps & ::= & \mathit{np} \mid \mathit{np} \cdot ps & \mathit{PoolVariables}
\end{array}
$$

Figure 4.1: Syntax of SHAPES^h where $p \in \mathit{PoolVariableId}$, $x \in \mathit{LocalVariableId}$, $C \in \mathit{ClassId}$, $f \in \mathit{FieldId}$, $m \in \mathit{MethodId}$, and $L \in \mathit{LayoutId}$. Differences from standard OO languages in *highlight*.

as a formalism and not as a runtime target, hence we do not need to be concerned with such inefficiencies; in § 5, we will demonstrate how SHAPES$_\ell$ places objects in the same pool close to each other in memory and clusters them according to a layout.

We now present the syntax, type system, and operational semantics of SHAPES^h.

4.1 The SHAPES^h language

Figure 4.1 presents the syntax of SHAPES^h; highlighted entities represent the syntactic entities that are novel with respect to other object-oriented languages. Classes in SHAPES^h are parameterised with pools. As usual, classes contain field and method definitions. Field definitions consist of their identifier and type. Method definitions consist of their identifier, a parameter and its type, a return type, and a method body. Method bodies consist of a preamble and statements. A preamble consists of declarations for local pools and local variables. Statements and expressions are as usual with respect to OO.

A layout declaration has the form **layout** L : $C =$ **rec** $\{fs_1\}$; .. **rec** $\{fs_n\}$. It introduces a new layout L, which describes how objects of class C residing in a pool with layout L are split into n clusters. All of the fields of C must be present in a layout declaration with no duplication. The first cluster will contain fields fs_1, the second cluster will contain fields fs_2, and so on.

$$
\begin{array}{rcl}
\mathcal{X} \in \mathit{Heap} & = & \mathit{Address} \rightharpoonup (\mathit{Object} \cup \mathit{Pool}) \\
\mathit{Address} & = & \mathit{ObjectAddress} \uplus \mathit{PoolAddress} \\
\mathit{Object} & = & \mathit{ClassId} \times \mathit{PoolArg}^+ \times \mathit{Record} \\
\mathit{Pool} & = & \mathit{LayoutId} \times \mathit{PoolArg}^+ \\
\pi \in \mathit{PoolArg} & = & \mathit{PoolAddress} \cup \{\mathbf{none}\} \\
\rho \in \mathit{Record} & = & \mathit{FieldId} \rightarrow \mathit{Value} \\
\beta \in \mathit{Value} & = & \mathit{ObjectAddress} \cup \{\mathbf{null}\} \\
\Phi \in \mathit{SFrame} & = & \mathit{VariableId} \rightharpoonup (\mathit{Value} \cup \mathit{PoolArg}) \cup (\{\mathbf{none}\} \rightarrow \{\mathbf{none}\}) \\
\Sigma \in \mathit{Stack} & = & \mathit{SFrame}^*
\end{array}
$$

Figure 4.2: Dynamic Entities of SHAPES^h where $\omega \in \mathit{ObjectAddress}$.

Object types consist of the name of a class followed by a sequence of pool arguments, some of which may be **none**. An object of type $C\langle p \cdot ps\rangle$ will belong to class C, reside in pool p, and its fields will point to objects whose placement is determined by the fields' declarations and the pool arguments $p \cdot ps$. An object of type $C\langle \mathbf{none} \cdot ps\rangle$ will belong to class C, will not reside in a pool, and its fields will point to objects whose placement is determined by the fields' declarations and the pool arguments $\mathbf{none} \cdot ps$. Because the first pool parameter specifies which pool an object will be allocated into, the first pool parameter of a class definition corresponds to the pool **this** is allocated into.

Pools are created dynamically upon execution of a method's preamble. A pool p created inside the preamble, *e.g.,* via **pools** .. $p : L\langle p \cdot ps\rangle$, contains objects of type $C\langle p \cdot ps\rangle$, which are organised according to layout L, where C is the class definition the layout L corresponds to. Objects are allocated inside a pool p by executing the expression **new** $C\langle p \cdot ps\rangle$.

Notation We will be using the following notation throughout the rest of this thesis:

We use several shorthand syntaxes in order to define maps. The syntax $[x_1 \,..\, x_n \mapsto y_1 \,..\, y_n]$ is a shorthand for $[x_1 \mapsto y_1, \,..\, , x_n \mapsto y_n]$. Also, $[x_1 \,..\, x_n \mapsto a^n]$ is a shorthand for $[x_1 \mapsto a, \,..\, , x_n \mapsto a]$. Similarly, if $x_s \leq x_e$, then the syntax $[x_s, \,..\, , x_e \mapsto a]$ is a shorthand for $[x_s \mapsto a, x_s + 1 \mapsto a, \,..\, , x_e - 1 \mapsto a, x_e \mapsto a]$.

We append s to names to indicate sequences: xs is a sequence of x-s. We use $\cdot$ for list concatenation: $p \cdot ps$ is a list where we prepend p into ps.

We use the notation p to indicate a pool variable, and np to indicate a pool variable or **none**. For ease of notation, we use ps to indicate a sequence of np-s, *i.e.,* any elements in ps may be **none**.

We use the syntax $F(x_1 \cdot \ .. \ \cdot x_n)$ as a shorthand for $F(x_1) \cdot \ .. \ \cdot F(x_n)$.

4.2 Execution of SHAPES^h Programs

The execution of SHAPES^h corresponds to the execution of a typical OO language when not taking the *highlighted* parts into account. The SHAPES^h heap adds pool entities to standard OO; these pool entities consist of the layout L they adhere to. To express the correspondence between SHAPES^h and SHAPES$_\ell$, we enrich the SHAPES^h semantics with ghost information; this ghost information consists of all the pool arguments passed into an object or pool at the time of its creation (*e.g.,* an object created through **new** $C\langle p_1 \ .. \ p_n\rangle$ will also contain the addresses of the pools p_1 to p_n).

The SHAPES^h operational semantics is given in terms of large steps semantics, and has the form $\mathcal{X}, \Sigma, stmts \rightsquigarrow \mathcal{X}', \Sigma', \beta$. That is, a heap $\mathcal{X}$, a stack of frames Σ, and a sequence of statements $stmts$ are reduced to a new heap $\mathcal{X}'$, a new stack Σ', and a value β.

Figure 4.2 presents the definitions of the SHAPES^h runtime entities. To illustrate them and aid in their presentation, we will use the code presented in Listing 4.1 (written in SHAPES), which builds on the definitions of classes `Professor` and `Student` (Lines 1 and 5, respectively, in Listing 2.2). Figure 4.3 depicts the SHAPES^h configuration (stack and heap) after execution of the code in Listing 4.1.

SHAPES^h runtime configurations consist of a stack (Σ) of frames (Φ) mapping identifiers to values, and heaps ($\mathcal{X}$) mapping object and pool addresses to objects (ω) or pools (π), respectively. In Figure 4.3, the stack consists of one frame, with variables `pStu`, `at`, `ec`, `jf`, `sd`; the heap consists of the objects with addresses π_1, ω_1, ω_2, ω_3, ω_4. For convenience, if $\Sigma = \Phi \cdot \Sigma'$ we use $\Sigma(x)$ and $\Sigma[x \mapsto \beta]$ as shorthands for $\Phi(x)$ and $\Phi[x \mapsto \beta] \cdot \Sigma'$, respectively. That is,

```
1  layout StudentL: Student =
2      rec{■name, ■age} + rec{■supervisor};
3  ...
4  pools pStu: StudentL<pStu, none>;
5  at = new Student<pStu, none>;
6  ec = new Student<pStu, none>;
7  jf = new Student<pStu, none>;
8
9  sd = new Professor<none>;
10 jf.supervisor = sd;
```

Listing 4.1: SHAPES example used in § 4 and § 5

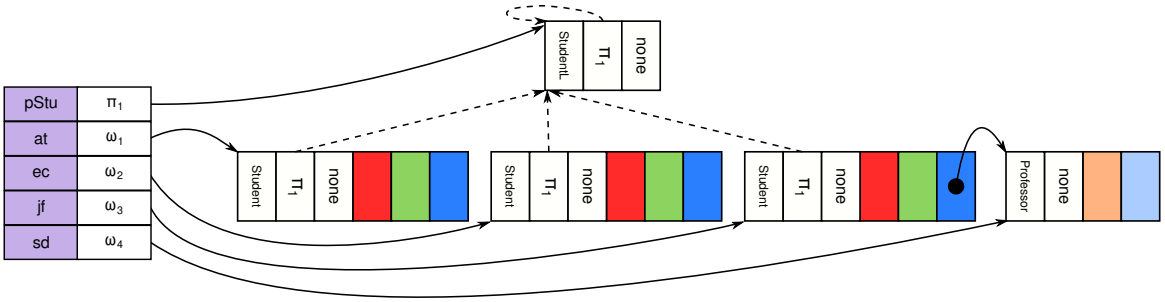


Figure 4.3: SHAPES^h stack and heap representation for Listing 4.1

accessing and modifying a variable through a stack only takes the top frame into account.

Objects consist of a class identifier C (determining its type), a sequence of pool arguments (*i.e.*, pool addresses, some of which may be **none**), and a record. A standalone object has **none** as its first pool parameter (*e.g.*, `Professor` at address ω_4 in Figure 4.3); an object stored in a pool π has a pool address π as its first parameter (*e.g.*, `Students` at addresses ω_1, ω_2, ω_3 belong to pool with address π_1. The fields' values are stored as a record (ρ), which maps the object's fields to values.

Pools consist of a layout identifier L, and a sequence of pool arguments. The layout identifier determines how the objects inside the pool are laid out and Pools can only store instances of the class corresponding to layout identifier. As mentioned, pools in SHAPES^h do not control the placement or layout of objects belonging to them; standalone and pooled objects have the exact same representation, hence they are treated uniformly.

Values are either object addresses, or **null**. Because SHAPES^h allows pools to be referenced by variables, stack frames (Φ) map variables to either values, pool addresses or **none**. SHAPES^h uses sequences of stack frames (Σ). We require for convenience that any frame maps **none** to **none**.

The operational semantics of SHAPES^h (Figure 4.4) deviate from that of typical OO in two

[Value]
$$\frac{}{\mathcal{X}, \Sigma, \mathbf{null} \rightsquigarrow \mathcal{X}, \Sigma, \mathbf{null}}$$

[Variable]
$$\frac{}{\mathcal{X}, \Sigma, x \rightsquigarrow \mathcal{X}, \Sigma, \Sigma(x)}$$

[Assignment]
$$\frac{\mathcal{X}, \Sigma, e \rightsquigarrow \mathcal{X}', \Sigma', \beta}{\mathcal{X}, \Sigma, x = e \rightsquigarrow \mathcal{X}', \Sigma'[x \mapsto \beta], \beta}$$

[Statement sequence]
$$\frac{\mathcal{X}, \Sigma, e \rightsquigarrow \mathcal{X}', \Sigma', \_ \quad \mathcal{X}', \Sigma', \mathit{stmts} \rightsquigarrow \mathcal{X}'', \Sigma'', \beta}{\mathcal{X}, \Sigma, e\ ;\ \mathit{stmts} \rightsquigarrow \mathcal{X}'', \Sigma'', \beta}$$

[New Object]
$$\frac{\Sigma(ps) = \pi s \quad \omega \notin \mathcal{X} \quad fs = \mathcal{F}s(C) \quad \mathcal{X}' = \mathcal{X}[\omega \mapsto (C, \pi s, [fs \mapsto \mathbf{null}^{|fs|}])]}{\mathcal{X}, \Sigma, \mathbf{new}\ C\langle ps \rangle \rightsquigarrow \mathcal{X}', \Sigma, \omega}$$

[Object Read]
$$\frac{\Sigma(x) = \omega \quad \mathcal{X}(\omega) = (C, \_, \rho)}{\mathcal{X}, \Sigma, x.f \rightsquigarrow \mathcal{X}, \Sigma, \rho(f)}$$

[Object Write]
$$\frac{\Sigma(x) = \omega \quad \Sigma(x') = \omega' \quad \mathcal{X}(\omega) = (C, \pi s, \rho) \quad \mathcal{X}' = \mathcal{X}[\omega \mapsto (C, \pi s, \rho[f \mapsto \omega'])]}{\mathcal{X}, \Sigma, x.f = x' \rightsquigarrow \mathcal{X}', \Sigma, \omega'}$$

Figure 4.4: Operational semantics for pool-agnostic operations.

ways: Firstly, we change the semantics so as to store our ghost information (*i.e.,* addresses of the pool parameters *ps* provided by the **new** $C\langle ps \rangle$ expression) into objects. Secondly, Rule MethodCall is now also tasked with the construction of the method-local pools, as well as the passing of references to pools from the caller to the callee. Nevertheless, the syntax for constructing objects and accessing/mutating their fields is the same, regardless of whether the object is standalone or pooled (*i.e.,* these rules are *pool-agnostic*).

Method call The operational semantics for method call are presented in Figure 4.5, in two different rules. The first, Rule MethodCall, constructs the stack frame corresponding to the method that is about to be called, and returns the value yielded from evaluation of the method body back to its caller. Passing the value of the implicit **this** parameter and the method parameter is done in the same manner as in a typical OO calculus. It is also necessary, however, to set the values of the class parameters. This is because the pool parameters of the class can be used as parameters in type declarations inside a method body (more specifically in **new** statements). We pass the pool addresses stored into the object the method is invoked against and store them into the frame.

Evaluation of a method body is defined in Rule MethodBody. Here, we must initialise the local

[MethodCall]

$$\frac{\begin{array}{c}\Sigma(x) = \omega \quad \mathcal{X}(\omega) = (C, \pi s, \_) \\ \mathcal{M}(C, m) = (\_, x' : \_, \mathit{localPools}; \mathit{localVars}\ ;\ \mathit{stmts}) \\ \Sigma' = [\mathbf{this} \mapsto \omega, x' \mapsto \Sigma(x''), \mathcal{P}s(C) \mapsto \pi s] \cdot \Sigma \\ \mathcal{X}, \Sigma', \mathit{localPools}; \mathit{localVars}\ ;\ \mathit{stmts} \rightsquigarrow \mathcal{X}', \_, \beta\end{array}}{\mathcal{X}, \Sigma, x.m(x'') \rightsquigarrow \mathcal{X}', \Sigma, \beta}$$

[MethodBody]

$$\frac{\begin{array}{c}\mathit{localPools} = \mathbf{pools}\ p_1\colon L_1\langle ps_1\rangle\ ..\ p_n\colon L_n\langle ps_n\rangle \\ \mathit{localVars} = \mathbf{vars}\ x_1\colon \_\ ..\ x_m\colon \_ \\ \pi_1, \ .. \ , \pi_n \notin \mathcal{X} \quad \forall i, j.\ [i \neq j \rightarrow \pi_i \neq \pi_j] \\ \Sigma' = \Sigma[p_1\ ..\ p_n \mapsto \pi_1\ ..\ \pi_n][x_1\ ..\ x_m \mapsto \mathbf{null}^m] \\ \mathcal{X}' = \mathcal{X}[\pi_1 \mapsto (L_1, \Sigma'(ps_1)), \ .. \ , \pi_n \mapsto (L_n, \Sigma'(ps_n))] \\ \mathcal{X}', \Sigma', \mathit{stmts} \rightsquigarrow \mathcal{X}'', \Sigma'', \beta\end{array}}{\mathcal{X}, \Sigma, \mathit{localPools}; \mathit{localVars}; \mathit{stmts} \rightsquigarrow \mathcal{X}'', \Sigma'', \beta}$$

Figure 4.5: Operational semantics for method call.

variables defined in the method's preamble. Object variables are initialised to **null**. For pool variables, new (empty) pools are constructed in a two-step manner: The pools are first reserved on the heap and then they are actually constructed, along with the stack frame. This allows us to have cycles among pools.

4.3 Type System

The type system has the remit of ensuring that at runtime:

A1 Objects' fields point to objects of the appropriate class (as usual).

A2 Objects are allocated in the appropriate pools and adhere to the layout of that pool (hence ensuring memory safety).

A3 Pool homogeneity is preserved.

We will be using the lookup functions defined in Figure 4.6. We define the following lookup functions. For simplicity, we implicitly assume that layout and class identifiers are unique within the same program, and field and method identifiers are unique within the same class.

$$
\begin{aligned}
\mathcal{C}(C) &\triangleq (pds\ fds\ mds)\ \textit{iff}\ (\textbf{class}\ C\langle pds\rangle\{fds\ mds\}) \in prog[0] \\
\mathcal{P}s(C) &\triangleq p_1\ ..\ p_n\ \textit{iff}\ \mathcal{C}(C)[0] = (p_1 \colon \_\ , ...,\ p_n \colon \_) \\
\mathcal{P}b(C) &\triangleq pbd_1\ ..\ pbd_n\ \textit{iff}\ \mathcal{C}(C)[0] = (\_ \colon pbd_1\ , ...,\ \_ \colon pbd_n) \\
\mathcal{B}(C,p) &\triangleq pbd\ \textit{iff}\ (p : pbd) \in \mathcal{C}(C)[0] \\
\mathcal{M}(C,m) &\triangleq (t, x : t', localPools; localVars, stmts)\ \textit{iff} \\
&\qquad (\textbf{def}\ m(x : t') : t\ \{localPools; localVars; stmts\}) \in \mathcal{C}(C)[2] \\
\mathcal{F}(C,f) &\triangleq t\ \textit{iff}\ (f : t) \in \mathcal{C}(C)[1] \\
\mathcal{F}s(C) &\triangleq f_1\ ..\ f_n\ \textit{iff}\ \mathcal{C}(C)[1] = (f_1 : \_\ ..\ f_n : \_) \\
\mathcal{L}(L) &\triangleq (C, fs_1\ ..\ fs_n)\ \textit{iff} \\
&\qquad (\textbf{layout}\ L : C = \textbf{rec}\{fs_1\};\ ..\ \textbf{rec}\{fs_n\}) \in prog[1] \\
\mathcal{O}(L,f) &\triangleq (i,j)\ \textit{iff}\ \mathcal{L}(L) = (C, fss)\ \wedge\ fss[i,j] = f \\
\mathcal{O}(C,f) &\triangleq i\ \textit{iff}\ \mathcal{F}s(C)[i] = f \\
\mathcal{C}l(L) &\triangleq \mathcal{L}(L)[0] \\
\mathcal{R}s(L) &\triangleq \mathcal{L}(L)[1]
\end{aligned}
$$

Figure 4.6: SHAPES^h lookup functions

Typing takes place in the context of an environment Γ, which maps object variables to object types (t), and pool variables to pool types (pt) or bounds (pb). We define a typing environment as follows:

Definition 4.1 (Environment)**.**

$$
\begin{aligned}
\Gamma \in \mathit{TypingContext} &::= x \colon t, \Gamma \mid p \colon u, \Gamma \mid \epsilon \\
u \in \mathit{PoolTypeOrPoolBound} &::= pt \mid pb \\
pt \in \mathit{PoolType} &::= L\langle ps\rangle \\
pb \in \mathit{PoolBound} &::= [C\langle ps\rangle] \mid \textbf{None} \\
T &::= t \mid u
\end{aligned}
$$

We distinguish three kinds of types:

Object Types ($C\langle ps\rangle$), where C is a class and ps are pool arguments, some of which may be **none**. They specify objects of class C, and the arguments ps specify the pools containing

$$\frac{\Gamma \vdash C\langle ps\rangle}{\Gamma \vdash \textbf{null} : C\langle ps\rangle}\ [\textsc{Value}] \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ [\textsc{Variable}] \qquad \frac{\Gamma \vdash x : t \qquad \Gamma \vdash e : t}{\Gamma \vdash x = e : t}\ [\textsc{Assignment}]$$

$$\frac{\Gamma \vdash e : t' \qquad \Gamma \vdash stmts : t}{\Gamma \vdash e; stmts : t}\ [\textsc{Statements}] \qquad \frac{\Gamma \vdash C\langle ps\rangle}{\Gamma \vdash \textbf{new}\ C\langle ps\rangle : C\langle ps\rangle}\ [\textsc{NewObject}]$$

$$\frac{\Gamma \vdash x : C\langle ps\rangle \qquad t = \mathcal{F}(C, f)[\mathcal{P}s(C)/ps]}{\Gamma \vdash x.f : t}\ [\textsc{FieldRead}] \qquad \frac{\Gamma \vdash x.f : t \qquad \Gamma \vdash x' : t}{\Gamma \vdash x.f = x' : t}\ [\textsc{FieldWrite}]$$

$$\frac{\Gamma \vdash x : C\langle ps\rangle \qquad \mathcal{M}(C, m) = (t, \_ : t', \_, \_) \qquad \Gamma \vdash x'' : t'[\mathcal{P}s(C)/ps]}{\Gamma \vdash x.m(x'') : t[\mathcal{P}s(C)/ps]}\ [\textsc{MethodCall}]$$

Figure 4.7: Typing Expressions and statements.

the object itself and the pools containing the objects pointed by that object's fields.

Pool Types ($L\langle ps\rangle$) describe pools which store objects of type C and are organised according to layout L. The arguments ps specify which pools contain the objects pointed by the fields of the objects stored in this pool. Pool types characterise pool values, *i.e.,* pools allocated dynamically through execution of a method's preamble.

Pool Bounds ($[C\langle ps\rangle]$ and **None**) Pool bounds characterise both formal pool parameters (whose layout is not necessarily known at that scope) and pools instantiated inside a method (whose layout is explicitly specified). The type **None** is only needed when translating SHAPES^h into SHAPES_ℓ, specifically during method specialisation (§ 5.5).

Expression and Statement Types Typing has the standard form $\Gamma \vdash e : t$ and $\Gamma \vdash stmts : t$. Notice that the type rules only return object types. Pool types and pool bounds are only used in ascertaining that types are well-formed. The type rules are presented in Figure 4.7. These are the type rules which ensure that Objectives A1–A3 hold.

The first five rules in Figure 4.7 are standard. **null** can have any well-formed object type (Rule Value). The type of a variable x is looked-up in Γ (Rule Variable). Assignment to a local variable is valid if both the variable and the right-hand-side expression have the same type (Rule Assignment). Notice that we do not model inheritance or subtyping. A sequence of expressions is well-typed if all expressions in it are well-typed (Rule Statements). Creation of

$$\text{[ObjTypeWF]}\quad \frac{\forall i.\ \Gamma \vdash ps[i] :: \mathcal{B}(C, \mathcal{P}s(C)[i])[\mathcal{P}s(C)/ps]}{\Gamma \vdash C\langle ps\rangle} \qquad \text{[BndWF]}\quad \frac{\Gamma \vdash C\langle ps\rangle}{\Gamma \vdash [C\langle ps\rangle]}$$

$$\text{[PoolTypeWF]}\quad \frac{\Gamma \vdash C\langle ps\rangle \quad \mathcal{C}l(L) = C}{\Gamma \vdash L\langle ps\rangle} \qquad \text{[PoolVar]}\quad \frac{}{\Gamma \vdash p :: \Gamma(p)} \qquad \text{[PoolBnd]}\quad \frac{\Gamma \vdash p :: L\langle ps\rangle \quad \mathcal{C}l(L) = C}{\Gamma \vdash p :: [C\langle ps\rangle]}$$

$$\text{[None1]}\quad \frac{}{\Gamma \vdash \textbf{none} :: \textbf{None}} \qquad \text{[None2]}\quad \frac{\Gamma \vdash [C\langle ps\rangle]}{\Gamma \vdash \textbf{none} :: [C\langle ps\rangle]}$$

Figure 4.8: Well-formed Types

a new object is valid and has type $C\langle ps\rangle$ if $C\langle ps\rangle$ is a valid type (Rule NewObject).

The following three rules are concerned with pool arguments, These rules are similar to those in Featherweight Java [IPW01], or Ownership Types [CÖSW13] (in the sense that classes are parameterised), the difference being that in SHAPES^h, class parameters are pools instead of types (as in Java) or objects (as in Ownership Types).

Rule FieldRead looks up a field f from a receiver x of type $C\langle ps\rangle$; the function $\mathcal{F}(C, f)$ looks up the definition of f as found in class C. The formal parameters from class C must be substituted by the pool arguments in ps, hence the substitution $[\mathcal{P}s(C)/ps]$. For example, the term `s1.supervisor` in Line 18 of Listing 2.2 has type `Professor<pProf1>`.

Similar substitutions are applied in Rules FieldWrite and MethodCall to translate between the internal names of the class parameters and the arguments used at the call site. With these rules, the assignment `s1.supervisor = new Professor<pProf1>` would be legal, while, given an `otherStudent` of type `Student<none, none>`, the assignment `otherStudent.supervisor = new Professor<pProf1>` would be illegal.

Well-formed Types Figure 4.8 describes well-formedness of types, which is the way we statically enforce homogeneity in SHAPES^h. Rule ObjTypeWF mandates that the type $C\langle ps\rangle$ is well formed, if each of the arguments $ps[i]$ adheres to the bound of the i-th formal parameter

of C, (*i.e.,* $\mathcal{B}(C, \mathcal{P}s(C)[i])$) when we have substituted all formal parameters of C with ps.

The judgement for pool variables adhering to pool types and bounds has the format $\Gamma \vdash np :: u$. By Rule PoolBnd, a pool variable adheres to its bound as given in Γ, and by Rule PoolVar, a pool variable which adheres to $L\langle ps\rangle$ also adheres to $[C\langle ps\rangle]$, where C is the class of the layout L.

Thus, in Line 18 of Listing 2.2, the type `Student<pStu1, pProf1>` is well-formed (as `pStu1` adheres to the bound `[Student<pStu1, pProf1>]`, and `pProf1` adheres to the bound `[Professor<pProf1>]`). However, in Line 20, the type `Student<pStu1, pProf2>` is badly formed; for it to be well-formed, we would need for `pStu1` to adhere to the bound `[Student<pStu1, pProf2>]`, but this cannot be since the bound of `pStu1` is `[Student<pStu1, pProf1>]`. Therefore, Line 20 is rejected with a typing error.

Pools are not first class objects Pools are dynamic entities, as they are created upon entry to a method preamble. However, pools are not first class entities, as they cannot be the outcome of an execution, cannot be stored in fields, and the same pool variable cannot be re-assigned within execution of the same scope.

All these restrictions are necessary because pool variables are used within types. For instance, consider an object o with type `C<p1, p2>`; if we were to reassign pool `p1`, we would "invalidate" the type `C<p1, p2>` of o, as well as any other type or bound that references `p1`. This would be further complicated if, for instance, we permitted conditional reassignment of pools (*e.g.,* within an `if` statement). As such, we have decided to not treat pools as first class entities.

4.3.1 Homogeneity and Type Safety

We will now discuss how the type system achieves homogeneity. We will see that homogeneity leads to the introduction of some novel constraints on pool bounds.

As we said in section § 2.1, homogeneity requires that for any two objects o_1 and o_2 allocated in the same pool π, and any sequence of field reads $f_1, \cdots, f_n$, if $o_1.f_1.\ldots.f_n$ and $o_2.f_1.\ldots.f_n$

are defined and not **null**, then they must reside in the same pool π'. Through an inductive argument, we can convince ourselves that homogeneity is equivalent to *immediate homogeneity*, where the latter requires that for any pool π any objects o_1 and o_2 allocated in π, and any field f, if $o_1.f$ and $o_2.f$ are defined and not **null**, then they must reside in the same pool π'. Our objective is to consider how to achieve immediate homogeneity.

Remember that the static type of an object ($C\langle p \cdot ps\rangle$) determines the pool that object resides in, as well as the pools the object's fields reside in. In particular, if $\Gamma \vdash x : C\langle p \cdot ps\rangle$, then $\Gamma \vdash x : C'\langle p' \cdot \_\rangle$, where C' is some class, and $p' = p$, $p' =$ **none** or $\exists i.\ p' = ps[i]$, and the object pointed at by $x.f$ resides in p'. Uniformity ensures that all objects residing in the same pool p will have the type $C\langle p \cdot \_\rangle$. If, on top of uniformity, we can enforce *pool consistency*, *i.e.,* all objects in the same pool will have *identical* types (same class and same pool arguments), then we will have achieved immediate homogeneity.

To enforce pool consistency, we require that any types that coincide in the *first* pool argument will coincide in *all* pool arguments and be of the same class. That is, for types $C\langle p \cdot ps\rangle$ and $C'\langle p \cdot ps'\rangle$ in the same scope, $C = C'$ and $ps = ps'$. This is guaranteed by well-formedness of environments, which is defined below:

Definition 4.2 (Well-formed environments)**.**

$$\vdash \Gamma \quad \textit{iff} \quad \forall (\_ : T) \in \Gamma.\ \Gamma \vdash T \ \wedge\ \forall p.\ [\ \Gamma \vdash p :: [C\langle ps\rangle] \ \longrightarrow\ ps[0] = p\ ]$$

In the definition above, the requirement from the first conjunct (well-formedness of types) is standard, but the requirement from the second conjunct (*i.e.,* the type of a pool variable must have the variable itself as the first pool parameter) is novel. Such well-formed environments ensure pool consistency, *i.e.,* any types which have the same first argument are identical in the remainder.

Lemma 4.3 (Derivation of well-formed types from other well-formed types)**.**
If $\vdash \Gamma$, $\Gamma \vdash C\langle ps\rangle$ *and* $\Gamma_C \vdash C'\langle ps'\rangle$ *(where* Γ_C *is the environment used to typecheck the definition of class* C*, see Definition 4.6), then* $\Gamma \vdash C'\langle ps'[\mathcal{P}s(C)/ps]\rangle$.

```
1 class C<                                5 {
2     p1: [C<p1, p2, p3>],                6     f1: C<p2, p3, p1>;
3     p2: [C<p2, p3, p1>],                7     f2: C<p3, p2, p1>;
4     p3: [C<p3, p2, p1>]>                8 }
```

Listing 4.2: Example of the necessity of pool bounds

Proof. See § C.3. □

Lemma 4.4 (Well-formed expressions have well-formed types)**.**

If $\vdash \Gamma$ *and* $\Gamma \vdash e : T$*, then* $\Gamma \vdash T$*.*

Proof. By structural induction over the derivation of e and by using Lemma 4.3. See § C.3. □

Lemma 4.5 (Well-formed environments ensure pool consistency)**.**

$\vdash \Gamma \ \wedge \ \Gamma \vdash C\langle p \cdot ps\rangle \ \wedge \ \Gamma \vdash C'\langle p \cdot qs\rangle \ \longrightarrow \ ps{=}qs \ \wedge C{=}C'.$

Proof. If $\Gamma \vdash C\langle p \cdot ps\rangle$ and $\Gamma \vdash C'\langle p \cdot qs\rangle$, then $\Gamma \vdash p :: C\langle p \cdot ps\rangle$ and $\Gamma \vdash p :: C'\langle p \cdot ps'\rangle$ from Rule ObjTypeWF and the fact that $\mathcal{B}(C, \mathcal{P}s(C)[0]) = \mathcal{P}s(C)$ (Definition 4.7). However, because Γ is constructed so that p can only adhere to one pool bound (Definition 4.7), then it must hold that $C = C'$ and $ps = ps'$. □

We now see that our system enforces homogeneity. Namely, given two objects o_1 and o_2 in the same pool, and a field f, we will show that $o_1.f$ and $o_2.f$ reside in the same pool. Since o_1 and o_2 are in the same pool, they have type $\Gamma \vdash o_1 : C\langle p \cdot ps\rangle$ and $\Gamma \vdash o_2 : C\langle p \cdot ps'\rangle$, respectively. From the type system, and ignoring the cases for p and **none**, we obtain there exists some class C' and some i such that $\Gamma \vdash o_1.f : C'\langle p_1 \cdot \_\rangle$ and $\Gamma \vdash o_2.f : C'\langle p_2 \cdot \_\rangle$ where $p_1 = ps[i]$ and $p_2 = ps[i]$. All well-formed expressions have well-formed types, therefore we can apply Lemma 4.4, and obtain that $p_i = p'_i$, hence $o_1.f$ and $o_2.f$ will reside in the same pool.

Necessity of well-formed environments for homogeneity To see why well-formed environments are necessary to ensure homogeneity, consider Listing 4.2. If we apply Rule ObjTypeWF and substitute the pool parameters p_1, p_2, p_3 of bound $[C\langle p_2 \cdot p_3 \cdot p_1\rangle]$ (Line 3) with the pool parameters p_3, p_2, p_1 from Line 4 in that exact order, we obtain the bound $[C\langle p_2 \cdot p_1 \cdot p_3\rangle]$;

this bound is different from that of Line 3, hence Listing 4.2 would result in an ill-formed enviromnent.

Now, consider an environment Γ such that $\Gamma \vdash x : C\langle p_1 \cdot p_2 \cdot p_3 \rangle$. By applying Rule FieldRead for the expression $x.f_2.f_1.f_1$ multiple times, we derive that $\Gamma \vdash x.f_2.f_1.f_1 : C\langle p_1 \cdot p_3 \cdot p_2 \rangle$.

The first pool parameter of both x and $x.f_2.f_1.f_1$ is p_1, therefore they must belong to the same pool. However, by applying Rule FieldRead for field f_1 for both of these expressions, we conclude that $\Gamma \vdash x.f_1 : C\langle p_2 \cdot p_3 \cdot p_1 \rangle$ and $\Gamma \vdash (x.f_2.f_1.f_1).f_1 : C\langle p_3 \cdot p_2 \cdot p_1 \rangle$. We have therefore encountered a case of two objects that belong to pool p_1, but the objects they reference through field f_1 belong to different pools, which violates homogeneity.

4.4 Well-formedness

To prove soundness of the type system, we need the concepts of a well-formed program and a well-formed configuration.

4.4.1 Well-formed Programs

A program *prog* is *well-formed* if all of its class definitions and layout declarations are well-formed. For a class definition to be well-formed, all class parameters must have bounds whose first argument is that parameter, and a similar requirement must be made for all local pools. For example, (expanding on the code of Listing 2.2) the statement `pools pProf2: ProfL<pStu1>` would be illegal. The rest of the definitions are less surprising.

We define well-formed (formal) SHAPES^h programs as follows:

Definition 4.6 (Well-formed program)**.** *A SHAPES^h program is well-formed if all its layout and all its class declarations are well-formed.*

$$\vdash prog \ \textit{iff} \ (\ \forall cd \in prog[0].\ prog \vdash cd\ )\ \wedge\ (\ \forall ld \in prog[1].\ prog \vdash ld\ )$$

Definition 4.7 (Well-formed class declaration)**.** *A class C is well-formed if:*

- *Their first pool parameter has to be annotated with a bound that is of the same class and its parameters are the same as in the class declaration (and in the same order). That is, if the class pool parameters of the class C are $\mathcal{P}s(C) = p_1 \,..\, p_n$, then $\mathcal{B}(C, p_1) = [C\langle p_1, \ldots, p_n\rangle]$.*

- *The parameter list of all pool types must only contain parameters from the class' pool parameter list (i.e. $\mathcal{P}s(C)$). This means that the* **none** *keyword is disallowed as a pool parameter name.*

- *The fields must have class types that are well-formed against the typing context Γ where the class' formal pool parameters have their corresponding bounds as types. Moreover, Γ is well-formed.*

- *All the methods have a parameter and return type that is well-formed against the context Γ. Moreover, for each method, the corresponding method body is typeable against a context Γ' which is an augmentation of Γ and contains the types of* **this** *variable, local pool, and object variables of the method. Moreover Γ' is well-formed. Finally, each method must use a variable for its return method. This is necessary so as to ensure that the return value is not considered eligible for garbage collection.*

$$
\begin{aligned}
prog \vdash \textbf{class}\ C\langle p_1\colon [C_1\langle ps_1\rangle]\ ..\ p_n\colon [C_n\langle ps_n\rangle]\rangle\ \{\ fds\ mds\ \}\ \textit{iff}\\
\vdash \Gamma\ \wedge\ C_1 = C\ \wedge\ ps_1 = p_1\ ..\ p_n\\
\wedge\ \forall i.\ ps_i[0] = p_i\\
\wedge\ \forall i, j.\ ps_i[j] \neq \textbf{none}\\
\wedge\ \forall f : T \in fds.\ \Gamma \vdash T\\
\wedge\ \forall \textbf{def}\ m(x\colon t)\colon t'\ \{localPools; localVars\ ;\ stmts\} \in mds.[\\
\Gamma \vdash t\ \wedge\ \Gamma \vdash t'\\
\wedge\ \vdash \Gamma'\ \wedge\ \Gamma' \vdash stmts :\ t'\ ]\ \wedge\ \Gamma \vdash stmts\\
where\ \Gamma' = \Gamma, \textbf{this} : C\langle p_1\ ..\ p_n\rangle, x : t,\\
p'_1\colon L_1\langle ps'_1\rangle,\ ..\ , p'_k\colon L_k\langle ps'_k\rangle,\\
x_1\colon C'_1\langle ps''_1\rangle,\ ..\ , x_m\colon C'_m\langle ps''_m\rangle\\
localPools = \textbf{pools}\ p'_1\colon L_1\langle ps'_1\rangle\ ..\ p'_k\colon L_k\langle ps'_k\rangle\\
localVars = \textbf{locals}\ x_1\colon C'_1\langle ps''_1\rangle\ ..\ x_m\colon C'_m\langle ps''_m\rangle\\
where\ \Gamma = p_1\colon [C_1\langle ps_1\rangle]\ ..\ p_n\colon [C_n\langle ps_n\rangle]
\end{aligned}
$$

We define $\Gamma \vdash stmts$ *as follows:*

- $\Gamma \vdash e;\ stmts$ *iff* $\Gamma \vdash e\ \wedge\ \Gamma \vdash stmts$
- $\Gamma \vdash e$ *iff* $(e = \textbf{new}\ t) \rightarrow \Gamma \vdash t$

We now define well-formedness of layout declarations:

Definition 4.8 (Well-formed layout declaration)**.** *A layout declaration for instances of a class* C *is well-formed iff the disjoint union of its clusters' fields is the set of the fields declared in* C*.*

$$
\begin{aligned}
prog \vdash \textbf{layout}\ L\colon [C] = \textbf{rec}\ \{fs_1\}\ ..\ \textbf{rec}\ \{fs_n\}\ \textit{iff}\\
\{\mathcal{F}s(C)\} = \biguplus_{i \in 1\ ..\ n} \{fs_i\}
\end{aligned}
$$

This definition excludes repeated or missing fields. For example, given the class `Student` from

Listing 2.2, the following two layout declarations are ill-formed:

```
// repeated field
layout BadStudentL1: Student = rec{name, age} + rec{age, supervisor};
// missing field
layout BadStudentL2: Student = rec{name} + rec{age};
```

4.4.2 Well-formed Configurations

Defining well-formed configurations for SHAPES^h must take into account the fact that the pool parameters of the type of the same object may be different in different environments: An object o passed through a function call may have the type $C\langle p_1 \cdot p_2 \rangle$ in the caller's environment and the type $C\langle p_3 \cdot p_4 \rangle$ in the callee's environment.

To overcome this limitation of pool parameters when defining well-formed configurations, we use runtime types, wherein we replace each pool parameter with a pool address π or **none**. That is, the pool parameters of static types are variables (*e.g.*, $C\langle p_1 \cdot p_2 \rangle$), whereas the pool parameters of runtime types are pool addresses (*e.g.*, $C\langle \pi_1 \cdot \pi_2 \rangle$). The benefit of runtime types is that object and pool addresses do not change (barring isomorphism); the above object o will have a fixed runtime type (*e.g.*, $C\langle \pi_1 \cdot \pi_2 \rangle$) throughout execution.

Definition 4.9 (Runtime types)**.** *A runtime type τ is defined as follows:*

$$
\begin{aligned}
\tau \in \mathit{RunType} &::= \mathit{RunClassType} \cup \mathit{RunPoolType} \cup \mathit{RunBound} \\
\mathit{RunClassType} &::= C\langle \pi_1 \,..\, \pi_n \rangle \\
\mathit{RunPoolType} &::= L\langle \pi_1 \,..\, \pi_n \rangle \\
\mathit{RunBound} &::= [C\langle \pi_1 \,..\, \pi_n \rangle]
\end{aligned}
$$

In the context of a well-formed configuration, we can expect that an object with runtime type $C\langle \pi \cdot \pi s \rangle$ belongs to the pool with address π and that the pool at address π has a runtime type

$L\langle\pi \cdot \pi s\rangle$ such that $\mathcal{Cl}(L) = C$. *This implies uniformity.* Moreover, for two objects o_1, o_2 with runtime type $C\langle\pi \cdot \pi s\rangle$, we require that $o_1.f, o_2.f$ point to objects that belong to the same pool π'. π' is derived purely from the pool addresses $\pi \cdot \pi s$ and the type of field f in class C. *This implies homogeneity.*

Additionally, if, in the environment of a stack frame Φ, an object, pool, or class parameter adheres to the static type $C\langle ps\rangle$, $L\langle ps\rangle$, or, $[C\langle ps\rangle]$ respectively, then we can expect the pool parameters to be $\Phi(ps)$ object, pool, or bound to be C or L, respectively.

Given the above expectations, we now define the well-formedness of a runtime configuration:

Definition 4.10 (Well-formed high-level configurations)**.** *Well-formedness is defined as follows:*

- *Strong agreement for objects and pools:*
  - $\mathcal{X} \vDash \omega \triangleleft C\langle\pi s\rangle$ *iff* $\quad \mathcal{X}(\omega) = (C, \pi s, \rho) \ \wedge \ \mathcal{X} \vDash \pi s[0] \colon [C\langle\pi s\rangle] \ \wedge$ $\forall f.\ \mathcal{X} \vDash \rho(f) \colon \mathcal{F}(C, f)[\mathcal{P}s(C)/\pi s]$
  - $\mathcal{X} \vDash \pi \triangleleft L\langle\pi s\rangle$ *iff* $\quad \mathcal{X} \vDash \pi s[0] \colon L\langle\pi s\rangle \ \wedge \ \mathcal{Cl}(L) = C \ \wedge$ $\forall i.\ \mathcal{X} \vDash \pi s[i] \colon \mathcal{B}(C, \mathcal{P}s(C)[i])[\mathcal{P}s(C)/\pi s]$
- *Weak agreement for objects, pools, and bounds:*
  - $\mathcal{X} \vDash \omega \colon C\langle\pi s\rangle$ *iff* $\quad \mathcal{X}(\omega) = (C, \pi s, \_)$
  - $\mathcal{X} \vDash \mathbf{null} \colon C\langle\_\rangle$
  - $\mathcal{X} \vDash \pi \colon L\langle\pi s\rangle$ *iff* $\quad \mathcal{X}(\pi) = (L, \pi s) \ \wedge \ \pi = \pi s[0]$
  - $\mathcal{X} \vDash \pi \colon [C\langle\pi s\rangle]$ *iff* $\quad \mathcal{X}(\pi) = (L, \pi s) \ \wedge \ \pi = \pi s[0] \ \wedge \ \mathcal{Cl}(L) = C$
  - $\mathcal{X} \vDash \mathbf{none} \colon [C\langle\_\rangle]$
  - $\mathcal{X} \vDash \mathbf{none} \colon \mathbf{None}$
- *Well-formed heap:*

  $\vDash \mathcal{X}$ *iff* $\quad [\forall \omega \in \mathrm{dom}(\mathcal{X}).\ \exists \tau.\ \mathcal{X} \vDash \omega \triangleleft \tau] \ \wedge \ [\forall \pi \in \mathrm{dom}(\mathcal{X}).\ \exists \tau.\ \mathcal{X} \vDash \pi \triangleleft \tau]$

- *Well-formed stack frame and heap against an environment:*

$$\begin{aligned}\Gamma \vDash \mathcal{X}, \Phi \quad \textit{iff} \quad & \vDash \mathcal{X} \wedge \\ & \forall x \in \mathrm{dom}(\Phi).\ \exists C, ps.\ [\Gamma(x) = C\langle ps\rangle \ \wedge\ \mathcal{X} \vDash \Phi(x) \colon C\langle \Phi(ps)\rangle] \wedge \\ & \forall p \in \mathrm{dom}(\Phi).\ \exists L, C, ps.\ [\\ & \quad [\Gamma(p) = L\langle ps\rangle \ \wedge\ \mathcal{X} \vDash \Phi(x) \colon L\langle \Phi(ps)\rangle] \vee \\ & \quad [\Gamma(p) = [C\langle ps\rangle] \ \wedge\ \mathcal{X} \vDash \Phi(x) \colon [C\langle \Phi(ps)\rangle]] \\ &]\end{aligned}$$

- *Well-formed sequence of stack frames and heap against a sequence of environments:*
  - $\epsilon \vDash \mathcal{X}, \epsilon$
  - $\Gamma \cdot \Gamma s \vDash \mathcal{X}, \Phi \cdot \Sigma \quad \textit{iff} \quad \Gamma \vDash \mathcal{X}, \Phi \ \wedge\ \Gamma s \vDash \mathcal{X}, \Sigma$

□

Theorem 4.11 guarantees that if a well-formed configuration takes a reduction step, then the resulting configuration is well-formed too, and the resulting value agrees with the type of the statements.

Theorem 4.11 (Type Safety)**.** *For a well-formed program prog, given a heap $\mathcal{X}$, stack frame sequence Σ, corresponding typing environment sequence Γs, and sequence of statements stmts:*

If $\Gamma s \vDash \mathcal{X}, \Sigma \ \wedge\ \Gamma s[0] \vdash \mathit{stmts} : C\langle ps\rangle \ \wedge\ \mathcal{X}, \Sigma, \mathit{stmts} \rightsquigarrow \mathcal{X}', \Sigma', \beta$

then $\Gamma s \vDash \mathcal{X}', \Sigma' \ \wedge\ \mathcal{X}' \vDash \beta \colon C\langle \Sigma'(ps)\rangle$

Proof. By structural induction over the derivation $\mathcal{X}, \Sigma, \mathit{stmts} \rightsquigarrow \mathcal{X}', \Sigma', \beta$. More in § C.3. □

4.5 SHAPES in the large

We have presented the design of SHAPES through SHAPES^h, a minimal OO calculus. Our design allows fast type checking, as it only requires a simple substitution. It is also "backwards compatible" with existing OO languages, because **none** can always be used as a pool parameter.

SHAPES has been conceived as a language extension and should be, ideally, orthogonal to other features of OO languages. In particular, our implementation of SHAPES called SHAPES-z (§ 6)

can support the usual control flow structures (*i.e.,* conditionals, loops, return statements), quality-of-life features present in other languages such as scoping and mixed declarations & code.

With respect to other features present in OO languages, we expect SHAPES would be able to ideally support:

- Global pools, which can be added in an easy manner.
- Access modifiers for fields (*i.e.,* `private` and `public`) to enforce encapsulation. Because it is possible for layout declarations to expose the inner details of a class and hence violate encapsulation, it would also be beneficial to introduce a module system and require that all layout types for a class must be defined within the module the class is defined as well. This is similar to Rust traits [KN18], where a trait implementation for a `struct` must be defined in either the `struct`'s module or the trait's module.
- Types present in other languages (*e.g.,* array types). Regarding array types, inline arrays can be currently emulated with multiple fields and getters/setters that receive an index and branch on it. Additionally, our work on [TFW+18] presents how pool-backed dynamic arrays can be accommodated in SHAPES.
- Inheritance/polymorphism for standalone objects (in line with other OO languages), but not for pools. The rationale is that if class `Circle` inherits from class `Shape`, then being able to store an instance of `Circle` into a pool of `Shape` objects would require us to consider schemes for storing the values of the subclasses' additional fields into a pool; this would complicate the design of pools and possibly hinder performance. Additionally, we would not be able to store an instance of `Shape` into a pool of type `Circle` (in a manner similar to how we cannot store an instance of `Shape` into an array of type `Circle[]`).
- Static trait dispatch (à la Rust [KN18]) with possibly minimal work; this is thanks to the fact that we make use of method specialisation (§ 5.5) when we will translate SHAPES^h into SHAPES$_\ell$, a low-level language (§ 5). Dynamic trait dispatch, on the other hand,

could be supported, but at the expense of storing additional runtime information (address of pool, if any, and its layout).

- Java-style generics: A significant deviation from Java generics, however, would be the fact that the upper bounds on the type parameters would need to express the pools as well. We envisage that this can be achieved in a manner similar to that of [PNCB04].

Type system extensions can be also added to our design. Structural equality could be added with minimal hassle: Two types would be structurally equal if their classes were structurally equal, and their pool arguments were nominally equal. Then pools could hold objects of structurally equal types. Existential types could be added, but at the expense of homogeneity and a runtime lookup of pools' layouts.

```
1  class Professor {
2      name: String;
3      ssn: String;
4  }
5  class Student<pProf: [Professor<pProf>]> {
6      name: String;
7      age: int;
8      supervisor: Professor<pProf>;
9  }
10 layout ProfL: Professor = ...;
11 layout StudentL: Student = ...;
12 ...
13 pools pStu1: StudentL<pStu1, pProf1>,
14       pProf1: ProfL<pProf1>;
15       pProf2: ProfL<pProf2>;
16 s1 = new Student<pStu1>;
17 s2 = new Student<pStu1>;
18 p1 = new Professor<pProf1>;
19 p2 = new Professor<pProf2>;
20 s1.supervisor = p1; // OK!
21 s2.supervisor = p2; // ERR
22 ...
```

Figure 4.9: Listing 2.2 with the suggested syntax simplifications applied

Another point of extension for SHAPES would be the possibility of syntax succinctness thanks to the guarantees provided by homogeneity: If the first pool parameter of a type is not `none`, we can omit the remaining pool parameters. For example, in Line 18 of Listing 2.2, one need only write `Student<pStu>` instead of `Student<pStu, pProf>`. Additionally, the first pool parameter of a class declaration can be replaced with a keyword (*e.g.,* `mine`). For example, the definition of pool `pStu` in Line 6 of Listing 2.2 can be replaced with the `mine` keyword (with `mine` having the bound `Student<mine, pProf>`. Figure 4.9 depicts how the example of Listing 2.2 would look like with the above syntax simplification proposals.

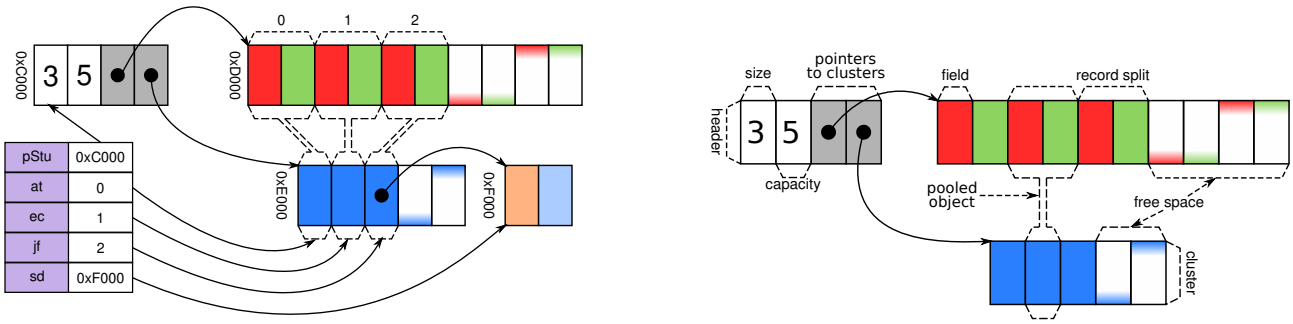
Chapter 5

SHAPES$_\ell$: Low-level SHAPES

Our aim for designing SHAPES was to allow developers to improve the cache usage of their applications via a high-level and type-safe language. This *does not imply* that we are content with providing a language where performance is an afterthought. After all, our end objective is to provide a tool that allows developers to improve the performance of their software; this makes performance considerations tantamount to the design of SHAPES.

Indeed, in § 2.7, we introduced the concept of *pool homogeneity* and argued in its favour on the expectation of additional performance guarantees, despite the fact that it would restrict the set of valid SHAPES programs. As such, it is reasonable to expect that we provide our justifications for why we expect SHAPES to be performant.

To that extent, we present SHAPES$_\ell$, an untyped intermediate representation with pool-aware instructions that operate on a flat memory model and which offers no explicit support for either objects or pools. That is, despite not being standalone entities, pools in SHAPES$_\ell$ are *implicitly* represented on the heap (similar to how objects in most language runtimes are implicitly represented as contiguous chunks of contiguous memory) and objects allocated into SHAPES$_\ell$ pools are allocated consecutively. Additionally, SHAPES$_\ell$ provides no support for the lookup of layout information at runtime; the code generated must statically know the layout a pool uses. Moreover, SHAPES$_\ell$ instructions can be easily translated to existing low-level intermediate representations such as LLVM [LA04] and/or invocations to a standard memory



(a) Stack and heap representation for Listing 4.1

(b) SHAPES$_\ell$ pool components and nomenclature

Figure 5.1: Representation of objects and pools in SHAPES$_\ell$

allocation library (*e.g.*, `malloc()`).

§ 5.1 and § 5.2 present the SHAPES$_\ell$ runtime configuration and syntax, respectively. § 5.3 presents the operational semantics of SHAPES$_\ell$. In § 5.5, we present how we translate SHAPES^h into SHAPES$_\ell$ by performing *specialisation*; more specifically, similar to template instantiation in C++ [ISO12] and monomorphisation with Rust traits [KN18], multiple specialisations of SHAPES^h functions are generated, each corresponding to a different combination of pool layouts.

5.1 Runtime Configuration

Like SHAPES^h, the SHAPES$_\ell$ runtime configuration consists of a heap (χ) and a stack (σ) of frames (ϕ). These runtime entities are presented in Figure 5.2. The heap is modelled as a sequence of memory cells that can grow infinitely; each cell contains a value (γ). Values can be addresses (α), natural numbers, or **null**. That is, we assume for the sake of convenience that both numbers and references have the same size. Frames map variable names to values.

Figure 5.1a shows a SHAPES$_\ell$ configuration that could occur after executing the example of Listing 4.1. This configuration shows the stack values corresponding to the pool `pStu` (Line 4) and the objects `at`, `ec`, `jf`, and `sd` (Lines 4–7), as well as how these entities are represented on the SHAPES$_\ell$ heap. Figure 5.1b lists the components of a SHAPES$_\ell$ pool and presents the nomenclature used for SHAPES$_\ell$ pools throughout this section.

$$\chi \in \mathit{Heap}^\ell = \mathit{Address}^\ell \to (\mathit{Size}^\ell \cup \mathit{Capacity}^\ell \cup \mathit{Value}^\ell)$$
$$\phi \in \mathit{SFrame}^\ell = \mathit{Variable}^\ell \to \mathit{Value}^\ell \qquad \alpha \in \mathit{Address}^\ell = \mathbb{N}$$
$$\sigma \in \mathit{Stack}^\ell = (\mathit{SFrame}^\ell)^* \qquad j \in \mathit{Size}^\ell = \mathbb{N}$$
$$\gamma \in \mathit{Value}^\ell = \mathit{Address}^\ell \cup \mathit{Index}^\ell \cup \{\textbf{null}\}$$
$$M \in \mathit{Capacity}^\ell = \mathbb{N}$$
$$k \in \mathit{Index}^\ell = \mathbb{N}$$

Figure 5.2: Low level runtime entities.

A standalone object (*e.g.,* the `Professor` corresponding to `sd`, Line 9) is modelled as a contiguous chunk of allocated memory. A reference to a standalone object consists of the address α to this chunk (*e.g.,* variable `sd` on the stack of Figure 5.1a).

A pool is modelled as several such chunks, with one being the *header* and the rest being the *clusters*. The *header* consists of the *size* and *capacity* of the pool, and the *pointers to the pool's clusters*. A reference to a pool points to the address of the pool's header. At any given time, a pool can only contain up to a finite number of objects; this number of objects is reflected in the pool's *capacity*. The pool's *size* indicates the number of objects the pool currently contains. As an example, the size and capacity of pool `pStu` (Line 4) is 3 and 5, respectively. Note that the header contains *no information* regarding the pool's layout.

All pooled objects that belong to the same pool have the values of their fields placed in chunks of contiguous memory that correspond to the pool's *clusters*; the pool header keeps a pointer to each of these clusters. A pool has the same number of clusters as the layout it adheres to. In our example, pool `pStu` adheres to layout `StudentL` (Line 1), hence it consists of two clusters: One corresponding to fields `name` and `age` and another corresponding to field `supervisor`.

Different clusters store different fields of a pooled object, hence pooled objects are effectively subdivided into *record splits*, with each record split being located on a specific cluster. For example, in Figure 5.1a, the objects `at`, `ec`, `jf` (Lines 5–7) are each subdivided into one record split consisting of fields `name` and `age` and another consisting of field `supervisor`. These record splits are each placed in the first and second cluster, respectively, of pool `pStu`.

In a SHAPES$_\ell$ pool, the k-th record split from each cluster will store the values for the respective fields of the k-th (zero-indexed) object in a pool. The pool's layout determines which record split contains which field of an object and how the fields are ordered in a record split. For example, in Figure 5.1a, the pooled object `jf` (Line 7) is the 2nd (zero-indexed) object in pool `pStu`, hence the 2nd record split from each cluster will contain the values of `jf` for fields `name` (for the first cluster), and `age`, `supervisor` (for the second cluster). The same applies to objects `at` and `ec` (Lines 5–6), which are the 0th and 1st objects in `pStu`, respectively.

A pooled object is uniquely identified by the address of the pool it belongs to and the index k indicating its position inside the pool. As an example, object `jf` (Line 7) in Figure 5.1a, is uniquely identified by the address of pool `pStu` (*i.e.,* `0xC000`) and its index inside `pStu` (*i.e.,* `2`). Despite that, references to pooled objects in SHAPES$_\ell$ *do not have to store the pool address.* This is because we rely on the pool storing that object to *always be in scope.* This is indeed the case with SHAPES^h: If $\Gamma \vdash o : C\langle p \cdot ps\rangle$, then o resides in pool p. As an example, the references to objects `at`, `ec`, `jf` (Lines 5–7) need only store the index of them inside `pStu` (*i.e.,* `0`, `1`, and `2`, respectively).

Since reference density can sometimes be non-trivial (*e.g.,* two applications on the SPECjvm98 benchmark used at least 40% of their allocated memory to store references to objects [DH99]), storing only the index can be a noteworthy improvement in terms of cache utilisation and memory usage. It is worth pointing out that some of the currently existing libraries for pooling and clustering (*e.g.,* DynaSOAr [SM19]) also attempt to compress the reference to the pool and the index in one machine word. However, this imposes an inherent limit on the number of objects in a pool on these implementations; SHAPES$_\ell$ suffers from no such constraints. In § 8.2, we discuss how SHAPES can be extended so that developers can reduce the footprint of references even further.

$$
\begin{array}{rcll}
prog^\ell & ::= & (fun^\ell)^+ & Program \\
fun^\ell & ::= & \textbf{fun}\ fn(\textbf{this}, p^+, x)\ \{vars^\ell; stmts^\ell\} & Function \\
stmts^\ell & ::= & rhs^\ell \mid rhs^\ell\ ;\ stmts^\ell & Statements \\
rhs^\ell & ::= & \textbf{null} \mid x = rhs^\ell \mid x \mid fn(x^+) \mid \textbf{alloc}(N) \mid \textbf{read}(x, i) \mid \textbf{write}(x, x', i) & Instruction \\
 & & \mid \textbf{palloc}(p, N^*) \mid \textbf{plread}(p, x, i, N, j) \mid \textbf{plwrite}(p, x, x', i, N, j) & \\
vars^\ell & ::= & \textbf{locals}\ (p = \textbf{plcreate}(N^*)^\ell)^*\ x^* & LocalsDecl
\end{array}
$$

Figure 5.3: SHAPES$_\ell$ syntax where $x, p \in Variable^\ell$, $fn \in FunctionId^\ell$, $N, i, j \in \mathbb{N}$.

5.2 Syntax of SHAPES$_\ell$

Figure 5.3 presents the syntax of SHAPES$_\ell$. A program consists of functions (fun^ℓ), each with parameters, local variables, and a body. Unlike SHAPES^h, SHAPES$_\ell$ makes no distinction between object and pool variables.

SHAPES$_\ell$ provides instructions that construct new objects or pools and access their fields. These come in pool-unaware (**alloc**, **read**, **write**) and pool-aware variants (**palloc**, **plread**, **plwrite**, **plcreate**).

5.3 Operational Semantics

SHAPES$_\ell$ execution has the format $\chi, \sigma, stmts^\ell \rightsquigarrow \chi', \sigma', \gamma$. Thus, a SHAPES$_\ell$ configuration (heap χ and stack σ) is reduced to a new configuration and return value γ.

Pool-agnostic operations Pool-agnostic operations in SHAPES$_\ell$ are similar to what we would expect from a typical intermediate representation:

- **alloc** constructs a new standalone object in memory (of size N). Construction of a new standalone `Professor` object is performed with the instruction `alloc(2)` (since `Professors` have 2 fields).
- **read** and **write** access an object's field f given its address and the offset i of f inside the object. Instruction `read(sd, 1)`, for example, fetches the value of field `ssn` (declared in `Professor`, Listing 2.2) from object `sd`.

$$\text{[Assignment]}\ \frac{\chi, \sigma, rhs^{\ell} \rightsquigarrow \chi', \sigma', \gamma}{\chi, \sigma, x = rhs^{\ell} \rightsquigarrow \chi', \sigma'[x \mapsto \gamma], \gamma} \qquad \text{[Var]}\ \frac{}{\chi, \sigma, x \rightsquigarrow \chi, \sigma, \sigma(x)}$$

$$\text{[Val]}\ \frac{}{\chi, \sigma, \mathbf{null} \rightsquigarrow \chi, \sigma, \mathbf{null}}$$

$$\text{[Sequence]}\ \frac{\chi, \sigma, rhs^{\ell} \rightsquigarrow \chi'', \sigma'', \_ \quad \chi'', \sigma'', stmts^{\ell} \rightsquigarrow \chi', \sigma', \gamma}{\chi, \sigma, rhs^{\ell};\ stmts^{\ell} \rightsquigarrow \chi', \sigma', \gamma} \qquad \text{[Garbage Collection]}\ \frac{\chi, \sigma \simeq_{\sigma} \chi'', \sigma'' \quad \chi'', \sigma'', stmts^{\ell} \rightsquigarrow \chi', \sigma', \gamma}{\chi, \sigma, stmts^{\ell} \rightsquigarrow \chi', \sigma', \gamma}$$

$$\text{[Alloc]}\ \frac{\alpha = \max\{\mathrm{dom}(\chi)\} + 1}{\chi, \sigma, \mathbf{alloc}(N) \rightsquigarrow \chi[\alpha\ ..\ \alpha + (N-1) \mapsto \mathbf{null}], \sigma, \alpha}$$

$$\text{[Object Read]}\ \frac{\alpha = \sigma(x) + i}{\chi, \sigma, \mathbf{read}(x, i) \rightsquigarrow \chi, \sigma, \chi(\alpha)} \qquad \text{[Object Write]}\ \frac{\alpha = \sigma(x) + i \quad \gamma = \sigma(x')}{\chi, \sigma, \mathbf{write}(x, x', i) \rightsquigarrow \chi[\alpha \mapsto \gamma], \sigma, \gamma}$$

Figure 5.4: Operational semantics of SHAPES_{ℓ} of pool-agnostic operations.

Pool-aware operations The SHAPES_{ℓ} pool-aware operations are as follows:

- **plread** and **plwrite**: Suppose that the pooled object o belongs to pool p with layout L and at α and that it has an index k inside p. Then, to access field f of o:
  - We first determine the index i of the cluster field f belongs to in L, thus obtaining address $\alpha' = \chi(\alpha + i + 2)$.
  - We then determine the address of the record split corresponding to o. The size of each record split is N, with N being the number of fields in the cluster f belongs to in L. Thus, our record split is located at address $\alpha'' = \alpha' + k * N$.
  - The address of the cell corresponding to field f of o is $\alpha'' + j$, where j is the offset j of the field f inside the record split in question.

  For example, to read field `age` of object `jf` in in pool `pStu`, we execute the instruction `plreadc(pStu, jf, 0, 2, 1)`. This is because for layout `StudentL`, field `age` is placed in the 0-th cluster ($i = 0$), the size of a record split in that cluster is $N = 2$ and `age` is the 1-st (zero-based) field in such a record split ($j = 1$).

[Fun]

$$\frac{\begin{array}{c}\mathcal{F}\mathbf{un}(fn) = (\mathbf{this} \cdot ps \cdot x, vars^\ell, stmts^\ell) \\ vars^\ell = \mathbf{locals}\ p_1 = \mathbf{plcreate}(Ns_1)\ ..\ p_n = \mathbf{plcreate}(Ns_n)\ x_1\ ..\ x_m \\ \chi_{i-1}, \epsilon, \mathbf{plcreate}(Ns_i) \rightsquigarrow \chi_i, \epsilon, \alpha_i \text{ for } i = 1\ ..\ n \\ \sigma' = [\mathbf{this} \mapsto \sigma(x), x \mapsto \sigma(x), ps \mapsto \sigma(ps)][p_1\ ..\ p_n \mapsto \alpha_1\ ..\ \alpha_n, x_1\ ..\ x_m \mapsto \mathbf{null}] \cdot \sigma \\ \chi_n, \sigma', stmts^\ell \rightsquigarrow \chi', \_, \gamma\end{array}}{\chi_0, \sigma, fn(x \cdot ps \cdot x) \rightsquigarrow \chi', \sigma, \gamma}$$

Figure 5.5: Operational semantics of SHAPES$_\ell$ functions.

- **plalloc** constructs a new pooled object in pool p. Alongside p, it takes a sequence $N_0\ ..\ N_{m-1}$ of parameters that specify the size of a record split in each of the m clusters. These parameters will allow **plalloc** to initialise the fields of the newly created object to **null**. As an example, `plalloc(pStu, 2, 1)` will construct a new `Student` inside pool `pStu`; the constants 2 and 1 correspond to the number of fields each cluster of `StudentL` contains, given that `pStu` is of layout `StudentL`.

  Allocation of pooled objects is trivial when the underlying pool can still accommodate objects (*i.e.,* size less than capacity): Similar to a bump allocator, we need to only increment the pool's size. For example, allocating a pool in `pStu` (Figure 5.1a) would increase the size of `pStu` to 4 and yield 3 as the new object's index. If the pool cannot accommodate any more objects, then the garbage collector (§ 5.3.1) will grow the pool in question beforehand.

- **plcreate** creates a new pool and returns its address. It takes the sizes of record splits in each cluster. The runtime picks an initial capacity for the pool, allocates the header and clusters and marks the pool as initially empty (*i.e.,* size of 0). For example, we use instruction `plcreate(2, 1)` to create a pool that adheres to layout `StudentL`. We discuss possible strategies for picking an initial pool capacity in § 8.2.

Method call It behaves similar to method calls in imperative languages, with the exception that pools are passed as arguments and pools are constructed explicitly (by using **plcreate**) at the beginning of the method's body.

The rules for pool-agnostic, pool-aware operations, and Rule Fun are given in Figure 5.4,

[Pool Read]
$$\frac{\sigma(p) = \alpha \quad \sigma(x) = k \quad \alpha' = \chi(\alpha + i + 2) + N * k + j}{\chi, \sigma, \mathbf{plread}(p, x, i, N, j) \rightsquigarrow \chi, \sigma, \chi(\alpha')}$$

[Pool Write]
$$\frac{\sigma(p) = \alpha \quad \sigma(x) = k \quad \sigma(x') = \gamma \quad \alpha' = \chi(\alpha + i + 2) + N * k + j}{\chi, \sigma, \mathbf{plwrite}(p, x, x', i, N, j) \rightsquigarrow \chi[\alpha' \mapsto \gamma], \sigma, \gamma}$$

[Pool Alloc]
$$\frac{\sigma(p) = \alpha \quad \chi(\alpha) = j \quad j < \chi(\alpha + 1) \quad n = |Ns| \quad \alpha_i = \chi(\alpha + i + 2) + Ns[i] * j \text{ for } i = 0 \,..\, n{-}1 \quad \alpha'_i = \alpha_i + Ns[i] - 1 \text{ for } i = 0 \,..\, n{-}1}{\chi, \sigma, \mathbf{plalloc}(p, Ns) \rightsquigarrow \chi[\alpha{+}1 \mapsto j{+}1][\alpha_0 \,..\, \alpha'_0 \mapsto \mathbf{null}, \,..\, , \alpha_{n-1} \,..\, \alpha'_{n-1} \mapsto \mathbf{null}], \sigma, j{+}1}$$

[Pool Create]
$$\frac{\alpha = \max\{\mathrm{dom}(\chi)\} + 1 \quad n = |Ns| \quad M \geq 0 \quad \alpha_i = (\alpha{+}2){+}i \text{ for } i = 0 \,..\, n{-}1 \quad \alpha'_0 = \alpha_{n-1} + 1 \quad \alpha'_i = \alpha'_{i-1} + Ns[i] * M \text{ for } i = 1 \,..\, n{-}1 \quad \alpha'_e = \alpha'_{n-1} + Ns[n-1] * M - 1 \quad \chi' = \chi[\alpha \mapsto 0, \alpha + 1 \mapsto M][\alpha_0 \,..\, \alpha_{n-1} \mapsto \alpha'_0 \,..\, \alpha'_{n-1}][\alpha'_0, \,..\, , \alpha'_e \mapsto \mathbf{null}]}{\chi, \sigma, \mathbf{plcreate}(Ns) \rightsquigarrow \chi', \sigma, \alpha}$$

Figure 5.6: Pool-oriented operational semantics of SHAPES_ℓ.

Figure 5.6, and Figure 5.5, respectively.

For simplicity and similar to § 4.2, we also use the convention that accessing and modifying a variable through a stack of frames σ addresses only the variable on the top-most stack frame ϕ. That is, if $\sigma = \phi \cdot \sigma'$, then $\sigma(x)$ and $\sigma[x \mapsto \gamma]$ are a shorthand for $\phi(x)$ and $\phi[x \mapsto \gamma] \cdot \sigma'$, respectively.

SHAPES_ℓ method lookup is defined as follows:

$$\mathcal{F}\mathbf{un}(fn) \equiv ((\mathbf{this}, ps, x), vars^\ell, stmts^\ell) \textit{ iff } \exists fun^\ell \in prog^\ell . fun^\ell = \mathbf{fun} fn(\mathbf{this}, ps, x)\{vars^\ell; stmts^\ell\}$$

5.3.1 The Garbage Collection rule

As we mentioned in § 5.3, if a pool has exhausted its capacity before an execution of a **plalloc** statement, the garbage collector is run so as to grow the pool in question and/or free up space within the pool by removing unreachable objects from the pool. Rule Garbage Collection (Figure 5.5) dictates how a garbage collector designed or retrofitted to accommodate SHAPES must operate.

Rule Garbage Collection states that the GC can only run in-between SHAPES$_\ell$ statements. During a GC cycle, both standalone and pool-allocated objects can be garbage collected. Moreover, the GC can not only collect and reorganise standalone objects in memory (as usual), but it can also relocate, grow, and shrink pools as well as collect and reorder the objects belonging to a pool to achieve compaction. Along with our pool representation, it is this compaction of pooled objects that allows us to achieve spatial locality within pools.

The GC reorganises the current runtime configuration χ, σ into a new configuration χ', σ' such that the two are equivalent ($\chi, \sigma \simeq_\sigma \chi', \sigma'$). That is, all objects and pools reachable in χ, σ through must have an isomorphic counterpart in χ', σ' and, additionally, pooled objects cannot be moved into another pool. We present the definition of $\simeq_\sigma$ in § C.2.

Note that pool growth does not necessary imply a partial or full GC invocation: The semantics of Rule Garbage Collection and the use of indices for references to pooled objects do permit a pool to be grown by merely having each of its clusters grown (*e.g.,* à la `realloc()` in C); in fact, we expect this to be the behaviour observed in the vast majority of cases in a future implementation.

Additionally, note that the pools themselves (as opposed to the objects in them) need not be garbage collected — their lifetime is bound to the frame where they are defined, allowing an entire pool to be released from memory in a single hit at the return from such a frame. This will not cause dangling pointers as the types necessary to point to objects in the pool are no longer nameable in the system.

We do not expect the design of SHAPES$_\ell$ to rely on any particular garbage collection technique

or algorithm, hence we do not expect writing a pool-aware garbage collector to be even more daunting and excruciating compared to writing a "typical" garbage collector. For a discussion on movement and compaction in pools, see [FHW+17]. Additionally, in § 8.2, we discuss possible future directions with respect to SHAPES and garbage collection.

5.4 Lack of concurrency support

An observation that needs to be pointed out is that SHAPES_ℓ does not lend itself to an easier support for concurrency with respect to pools. As an example, consider an object o belonging to pool p and two threads T_1, T_2. Additionally, consider that pool p is currently at full capacity and construction of a new object within p would require the clusters of p to be resized.

Now, suppose thread T_1 constructs a new object within pool p and thread T_2 accesses field f of object o. If we do not require any mutual exclusion, one possible execution scenario would be the following:

1. T_2 fetches the address of the cluster that values of f belong to.
2. T_1 resizes and relocates the clusters of p, then constructs a new object within p.
3. T_2 uses the now stale cluster address to fetch the value $o.f$, thus resulting in a use-after-free scenario.

As such, a solution supporting concurrency within SHAPES_ℓ pools would need to ensure that pool resizing is synchronised with respect to other operations on the pool, such as field access and object construction. We discuss a few such possible solutions in § 8.2.

5.5 Translation

We now describe the process of translating SHAPES^h into SHAPES_ℓ. The most significant difference between SHAPES^h and SHAPES_ℓ that we need to take into account during translation

is that SHAPES^h classes are polymorphic with respect to the layouts of pools (hence their member methods are also polymorphic), whereas SHAPES$_\ell$ does not provide any functionality to implement any such polymorphism. This implies that when translating SHAPES^h to SHAPES$_\ell$:

- We need to know whether an object is standalone or pool allocated so as to emit the appropriate variant of an instruction (*e.g.,* **read** vs **plread**).
- When dealing with a pooled object, we need to know the layout of the pool it belongs to, so that we can specify the appropriate constant values for parameters such as cluster index, record split size, etc.
- When translating a method invocation, we need to propagate any layout information we already know about the callee's pool parameters (so that when translating the called function, we will know the appropriate instructions to emit) or the method itself should be able to obtain the layout information regarding its pool parameters from scratch.

As an example, consider the translation of method `clone()` for class `Student` (Listing 2.2):

```
1 def clone(): Student<pStu, pProf> {
2   var obj = new Student<pStu, pProf>;
3   obj.name = this.name;
4   obj.age = this.age;
5   obj.supervisor = this.supervisor;
6   obj
7 }
```

The SHAPES$_\ell$ code emitted for `clone()` needs to behave differently when called on an object of type `Student<none, none>`, compared to when called on an object of type `Student<p, none>`, where p is a pool of `Students`.

To tackle this, we can modify SHAPES$_\ell$ to provide features for polymorphism and rely on the runtime to resolve layout information (*e.g.,* to perform field access) or we can assume that all layout information inside a method body is already known at compile time (hence all layout information for *e.g.,* field access is known at compile time) and require method specialisation in the case of method call. We decided to make use of specialisation in SHAPES$_\ell$. We use

specialised environments (Δ), where pool bounds are eliminated and pool variables have layout types or **None**:

Definition 5.1. $\Delta ::= x\colon t, \Delta \mid p\colon L\langle np^{+}\rangle, \Delta \mid p\colon \mathbf{None}, \Delta \mid \mathbf{none} : \mathbf{None}$

Definition 5.2 (Environment specialisation)**.** *We define that* Δ *specialises* Γ *(*$\Gamma \vdash \Delta$*) as follows:*

$$
\begin{aligned}
\Gamma \vdash \Delta \ \textit{iff} \quad & \mathrm{dom}(\Gamma) = \mathrm{dom}(\Delta) \wedge \\
& \forall x.[\ \Gamma(x) = C\langle\_\rangle \ \rightarrow \ \Delta(x) = \Gamma(x)\ ] \wedge \\
& \forall p.[\ \Gamma(p) = L\langle\_\rangle \ \rightarrow \ \Delta(p) = \Gamma(p)\ ] \wedge \\
& \forall p.[\ \Gamma(p) = [C\langle ps\rangle] \ \rightarrow \ \Delta(p) = \mathbf{None} \ \vee \ \exists L.[\ \mathcal{C}l(L) = C \ \wedge \ \Delta(p) = L\langle ps\rangle\ ]\ ]
\end{aligned}
$$

For two sequences of environments Γs*,* Δs*, we state that* Δs *specialises* Γs *(*$\Gamma s \vdash \Delta s$*) as follows:*

$\Gamma s \vdash \Delta s$ *iff* $\forall i.\ \Gamma s[i] \vdash \Delta s[i]$

In our examples, we shall be using two specialized environments, Δ_1 and Δ_2, such that:

Δ_1(s)	= `Student<pStu1, pProf1>`	Δ_2(s)	= `Student<pStu1, pProf1>`
Δ_1(pStu1)	= `StudentL<pStu1, pProf1>`	Δ_2(pStu1)	= `None`
Δ_1(pProf1)	= `ProfL<pProf1>`	Δ_2(pProf1)	= `ProfL<pProf1>`

We will be using the following definition of `StudentL`:

```
layout StudentL: Student = rec{name, age} + rec{supervisor};
```

Translation also makes use of lookup functions that provide information about the various layouts – full definitions are in Figure 4.6.

Translating Expressions and statements Figure 5.8 defines the translation of SHAPES^h expressions and statements in terms of rules of the form $[\![e]\!]_\Delta$ and $[\![stmts]\!]_\Delta$, where e and *stmts* are SHAPES^h expressions or statement sequences, and Δ is a specialised typing environment.

Translating Expressions and statements The first five rules are not that surprising: Variables and values are mapped to themselves; an assignment leaves the left hand side unmodifed

Expression	Δ_1	Δ_2
`new Student<pStu1, pProf1>`	`plalloc(p, 2, 1)`	`alloc(3)`
`s.age`	`plread(pStu1, s, 0, 1, 1)`	`read(s, 1)`
`s.getAge()`	`#Student_getAge_StudentL_ProfL( s, pStu1, pProf1)`	`#Student_getAge_None_ProfL( s, null, pProf1)`

Figure 5.7: Example translations

and translates the right hand side; a sequence of expressions is translated into a sequence of their translations.

Translating Object Creation and Field Access The next rule describes object creation. For a non-pooled object, *i.e.,* for an object of type $C\langle np \cdot \_ \rangle$ where $\Delta(np) = \textbf{None}$, we emit the instruction $\textbf{alloc}(N)$ where N is the number of fields in class C. For a pooled object, *i.e.,* an object of type $C\langle p \cdot \_ \rangle$ where $\Delta(p) = L\langle \_ \rangle$ we emit the instruction $\textbf{plalloc}(p, N_1 \; .. \; N_m)$ where m is the number of clusters in L, and N_i is the number of fields in the i-th cluster of layout L.

Similarly, for field access $x.f$, we distinguish between standalone and pooled objects. In the fist case, we emit $\textbf{read}(x, k)$ where k is the offset of f in the class of x. In the second case, we emit $\textbf{plread}(p, x, i, N, j)$, where p is the pool that contains x, and i is the cluster that contains f in the layout of p, and j is the offset of f within that cluster's corresponding record split, and N is the number of cells in that record split. Similar ideas apply to field write.

Figure 5.7 shows how the SHAPES^h expressions `new Student<pStu1, pProf1>` and `s.age` are translated into SHAPES$_\ell$ under environments Δ_1 and Δ_2, respectively.

Translating Method Call For method call, we make use of name mangling to determine the correct method to invoke, in a similar manner to what languages such as C++ do [ISO12]. The name of the method to be called is generated from $\mathcal{N}ame$. $\mathcal{N}ame$ generates a mangled method name by combining the member method's name and the specialised typing environment Δ being used.

$$[\![x]\!]_\Delta \triangleq x \qquad [\![\textbf{this}]\!]_\Delta \triangleq this \qquad [\![\textbf{null}]\!]_\Delta \triangleq \textbf{null}$$

$$[\![x = rhs^\ell]\!]_\Delta \triangleq x = [\![rhs^\ell]\!]_\Delta$$

$$[\![e; stmts]\!]_\Delta \triangleq [\![e]\!]_\Delta; [\![stmts]\!]_\Delta$$

$$[\![\textbf{new } C\langle np \cdot \_\rangle]\!]_\Delta \triangleq \begin{cases} \textbf{alloc}(|\mathcal{F}s(C)|) & \text{if } \Delta(np) = \textbf{None} \\ \textbf{palloc}(p, |fs_0| \,..\, |fs_n|) & \text{if } np = p \wedge \Delta(p) = L\langle \_ \rangle \\ & \wedge\ fs_0 \,..\, fs_n = \mathcal{R}s(L) \end{cases}$$

$$[\![x.f]\!]_\Delta \triangleq \begin{cases} \textbf{read}(x, \mathcal{O}(C, f)) & \text{if } \Delta(x) = C\langle np, \_ \rangle \ \wedge\ \Delta(np) = \textbf{None} \\ \textbf{plread}(p, x, i, N, j) & \text{if } \Delta(x) = C\langle p, \_ \rangle \ \wedge\ \Delta(p) = L\langle \_ \rangle \\ & \wedge\ \mathcal{O}(L, f) = (i, j) \ \wedge\ N = |\mathcal{R}s(L)[i]| \end{cases}$$

$$[\![x.f = x']\!]_\Delta \triangleq \begin{cases} \textbf{write}(x, x', \mathcal{O}(C, f)) & \text{if } \Delta(x) = C\langle np, \_ \rangle \ \wedge\ \Delta(np) = \textbf{None} \\ \textbf{plwrite}(p, x, x', i, N, j) & \text{if } \Delta(x) = C\langle p, \_ \rangle \ \wedge\ \Delta(p) = L\langle \_ \rangle \\ & \wedge\ \mathcal{O}(L, f) = (i, j) \ \wedge\ N = |\mathcal{R}s(L)[i]| \end{cases}$$

$$[\![x.m(x')]\!]_\Delta \triangleq \mathcal{N}ame_{\Delta'}(m)(x, np'_1 \,..\, np'_k, x') \quad \text{if } \Delta(x) = C\langle np_1 \,..\, np_k \rangle \wedge \Delta' = \textbf{this}\colon C\langle np_1 \,..\, np_k \rangle, p_1\colon \Delta(np_1), \,..\, , p_k\colon \Delta(np_k) \wedge \forall i \in 1..k.\ np'_i = \begin{cases} \textbf{null} & \textit{if } \Delta(np_i) = \textbf{None} \\ np_i & \textit{otherwise} \end{cases}$$

$$[\![\textbf{pools } p_1\colon \_ \,..\, p_n\colon \_; \textbf{locals } x_1\colon \_ \,..\, x_m\colon \_; stmts]\!]_\Delta \triangleq \textbf{locals } p_1 = \textbf{plcreate}(Ns_1);\ ..\ p_n = \textbf{plcreate}(Ns_n)\ x_1 \,..\, x_m; [\![stmts]\!]_\Delta \quad \textit{where} \quad \forall i.[\ \mathcal{R}s(\Delta(p_i)) = fs_0 \,..\, fs_n \ \rightarrow\ Ns_i = |fs_0| \,..\, |fs_n|\ ]$$

Figure 5.8: Translation of Expressions

$$\mathcal{N}ame_\Delta(m) \equiv \#C\_m\_G_1\_ \,..\, \_G_n$$

$$\text{where } \Delta(\textbf{this}) = C\langle p_1, ..., p_n \rangle \ \wedge\ G_i = \begin{cases} \textbf{None} & \text{if } \Delta(p_i) = \textbf{None} \\ L & \text{if } \Delta(p_i) = L\langle \_ \rangle \end{cases}$$

Figure 5.7 shows the translation of the method call `s.getAge()` under Δ_1 and Δ_2. Notice that in the case of Δ_2, as the pool parameter `pStu1` is of type **None**, hence it will never be used

inside `getAge()`, we set the argument corresponding to `pStu1` in `getAge()` to **null**.

Translating Methods and Classes Specialisation of SHAPES^h functions is performed by enumerating all possible specialised environments. We obtain all such environments through the $\mathcal{S}pecialiseClass$ function, which substitutes the types of formal pool parameters with layout types or **None**.

$$\mathcal{S}pecialiseClass(C) \equiv$$
$$\{\Delta \mid \mathrm{dom}(\Delta) = \mathcal{P}s(C) \wedge$$
$$\forall p \in \mathcal{P}s(C).\ [\mathcal{B}(C,p) = [C'\langle ps\rangle] \rightarrow \Delta(p) = \mathbf{None} \ \vee\ \exists L.\mathcal{C}l(L) = C' \ \wedge\ \Delta(p) = L\langle ps\rangle]\}$$

Thus, we define translation of a method as:

$$\mathcal{S}pecialiseMethod(C,m) \ \equiv$$
$$\{\ \mathcal{N}ame_{\Delta'}(m)(\mathbf{this}, p_1, \ ..\ , p_n, x')\{[\![localPools; localVars; stmts]\!]_{\Delta'}\} \mid$$
$$\Delta \in SpecialiseClass(C) \ \wedge$$
$$\Delta' = \Delta, \mathbf{this} : C\langle p_1\ ..\ p_n\rangle, x' : t', p_1' \colon L_1\langle ps_1'\rangle, \ ..\ , p_k' \colon L_k\langle ps_k'\rangle,$$
$$x_1 \colon C_1'\langle ps_1''\rangle, \ ..\ , x_m \colon C_m'\langle ps_m''\rangle\}$$
$$where\ \mathcal{M}(C,m) = (\_, x' : t', localPools; localVars, stmts),$$
$$and\ localPools = \mathbf{pools}\ p_1' \colon L_1\langle ps_1'\rangle\ ..\ , p_k' \colon L_k\langle ps_k'\rangle;$$
$$and\ localVars = \mathbf{vars}\ x_1 \colon C_1'\langle ps_1''\rangle\ ..\ x_m \colon C_m'\langle ps_m''\rangle$$

Specialisation *will always terminate*. This is because a specialisation replaces the pool bound $[C\langle ps\rangle]$ of each formal pool parameter with a layout type $L\langle ps\rangle$ (such that $\mathcal{C}l(L) = C$) or **None** and each class C has a finite number of layouts and formal pool parameters.

5.5.1 Correctness of Translation

We now show that translation is correct; that is, executing well-typed high-level SHAPES^h code in a high level configuration gives equivalent results as executing the translation of that SHAPES^h code in an equivalent specialised low-level configuration and vice versa. We state

soundness and completeness of translation in Theorems 5.4 and 5.5. In both theorems, we use a utility predicate $(\mathcal{X}, \Sigma \simeq_{\Gamma s, \Delta s, \mathcal{I}, stmts}\ \chi, \sigma)$ to ensure that:

- We have an initial high-level configuration $\mathcal{X}, \Sigma$ that is well-formed against a specialised environment Δs (otherwise we are dealing with the wrong method specialisation).
- The high-level $(\mathcal{X}, \Sigma)$ and low-level configurations (χ, σ) are equivalent.
- The SHAPES^h statements *stmts* we are translating into SHAPES_ℓ are well-typed under the typing environment Γs used for compilation.

We define $\mathcal{X}, \Sigma \simeq_{\Gamma s, \Delta s, \mathcal{I}, stmts}\ \chi, \sigma$ as follows:

$$\begin{array}{lllll} \mathcal{X}, \Sigma \simeq_{\Gamma s, \Delta s, \mathcal{I}, stmts}\ \chi, \sigma & \textit{iff} & \Gamma s \vdash \Delta s & \wedge & \Delta s \vDash \mathcal{X}, \Sigma \\ & \wedge & [\![stmts]\!]_{\Delta s[0]} = stmts^\ell & \wedge & \mathcal{X}, \Sigma \simeq_{\Delta s, \mathcal{I}}\ \chi, \sigma \end{array}$$

The relation $\mathcal{X}, \Sigma \simeq_{\Delta s, \mathcal{I}}\ \chi, \sigma$ (defined in § C.2) asserts that the high-level and low-level configurations $\mathcal{X}, \Sigma$ and χ, σ are equivalent under the typing environment Δs modulo renaming; the renaming is defined by injection $\mathcal{I}$.

The theorems also use the relation $\beta \simeq_{\mathcal{I}', ps, \sigma}\ \gamma$ to express object equivalence between the high and low-level configurations. That is, the object with address β in the high level corresponds to the standalone object with address γ or the pooled object with index γ in the low level. The pool the object belongs to in the low-level (if any) is derived from the stack σ and the pool parameters *ps*.

Definition 5.3 (Equivalence between low-level configurations)**.** *We define* $\chi, \sigma \simeq_{\Delta s, \mathcal{J}}\ \chi', \sigma'$ *under an injection* $\mathcal{J} : Address^\ell \cup (Address^\ell \times Index^\ell)$ *as follows:*

$\chi, \sigma \simeq_{\Delta s, \mathcal{J}} \chi', \sigma'$ iff

$[\forall i.\ \forall p \in \mathrm{dom}(\Delta s[i]).[\sigma[i](p) = \sigma'[i](p) = \mathbf{null}\ \vee\ \mathcal{J}(\sigma[i](p)) = \sigma'[i](p)]\ \wedge$

$[\forall i, path.[\Delta s[i] \vdash\ path :\ C\langle p \cdot \_\rangle\ \wedge$

$\Delta s[i], \chi, \sigma[i], path \rightsquigarrow \gamma\ \wedge$

$\Delta s[i], \chi', \sigma'[i], path \rightsquigarrow \gamma' \rightarrow \sigma[0](p), \gamma \simeq_{\mathcal{J}} \sigma'[0](p), \gamma']$

$]$

We define $\alpha, \gamma \simeq_{\mathcal{J}} \alpha', \gamma'$ as follows:

$\alpha, \gamma \simeq_{\mathcal{J}} \alpha', \gamma'$ *iff*

$[\gamma = \gamma' = \mathbf{null}]\ \vee\ [\alpha = \alpha' = \mathbf{null}\ \wedge\ \mathcal{J}(\gamma) = \gamma']\ \vee$

$[\alpha \neq \mathbf{null}\ \wedge\ \alpha' \neq \mathbf{null}\ \wedge\ \mathcal{I}(\alpha) = \alpha'\ \wedge\ \mathcal{I}(\alpha, \gamma) = (\alpha', \gamma')]$

Theorem 5.4 (Sound Translation)**.** *For two SHAPES^h and SHAPES$_\ell$ configurations that are well-formed and equivalent, a sequence of well-typed SHAPES^h statements will yield SHAPES^h configurations and return values equivalent to the SHAPES$_\ell$ configurations and return values (respectively) yielded by the execution of a specialisation of the SHAPES^h statements into SHAPES$_\ell$.*

That is:

$\forall \mathcal{X}, \Sigma, \chi, \sigma, \Gamma s, \Delta s, \mathcal{I}, stmts, stmts^\ell, C, ps, \chi', \sigma'.$

If $\mathcal{X}, \Sigma \simeq_{\Gamma s, \Delta s, \mathcal{I}, stmts} \chi, \sigma\ \wedge\ \Gamma s[0] \vdash\ stmts :\ C\langle ps\rangle\ \wedge\ \chi, \sigma, stmts^\ell \rightsquigarrow \chi', \sigma', \gamma$

Then $\exists \mathcal{I}', \mathcal{X}', \Sigma', \beta.$

$\mathcal{X}, \Sigma, stmts \rightsquigarrow \mathcal{X}', \Sigma', \beta\ \wedge\ \mathcal{X}', \Sigma' \simeq_{\Delta s, \mathcal{I}'} \chi', \sigma'\ \wedge\ \beta \simeq_{\mathcal{I}', ps, \sigma} \gamma$

Proof. By structural induction over the derivation $\chi, \sigma, stmts^\ell \rightsquigarrow \chi', \sigma', \gamma$. See § C.3. □

Theorem 5.5 (Translation is complete)**.** *For two well-formed and equivalent SHAPES^h and SHAPES$_\ell$ configurations, the specialised translation of a sequence of well-typed SHAPES^h statements will yield SHAPES$_\ell$ configurations and return values equivalent to the SHAPES^h configurations and return values (respectively) yielded by the execution of the SHAPES^h statements.*

That is:

$$\forall \mathcal{X}, \Sigma, \chi, \sigma, \Gamma s, \Delta s, \mathcal{I}, \mathit{stmts}, \mathit{stmts}^{\ell}, C, \mathit{ps}, \mathcal{X}', \Sigma'.$$
$$\mathit{If}\ \mathcal{X}, \Sigma \simeq_{\Gamma s, \Delta s, \mathcal{I}, \mathit{stmts}} \chi, \sigma\ \wedge\ \Gamma s[0] \vdash \mathit{stmts} : C\langle \mathit{ps} \rangle\ \wedge\ \mathcal{X}, \Sigma, \mathit{stmts} \rightsquigarrow \mathcal{X}', \Sigma', \beta$$
$$\mathit{Then}\ \ \exists \mathcal{I}', \chi', \sigma', \gamma.$$
$$\chi, \sigma, \mathit{stmts}^{\ell} \rightsquigarrow \chi', \sigma', \gamma\ \wedge\ \mathcal{X}', \Sigma' \simeq_{\Delta s, \mathcal{I}'} \chi', \sigma'\ \wedge\ \beta \simeq_{\mathcal{I}', \mathit{ps}, \sigma} \gamma$$

Proof. By structural induction over the derivation $\mathcal{X}, \Sigma, \mathit{stmts} \rightsquigarrow \mathcal{X}', \Sigma', \beta$. See § C.3. □

5.6 Conclusion

We have presented SHAPES$_\ell$, a low-level language that we expect can be easily utilised to provide an efficient runtime for a SHAPES implementation. We claim that the design of SHAPES$_\ell$ results in a pool representation we expect to allow the emission of efficient code. In particular, we do not need to retain pool parameters or any other additional runtime type information and calculating addresses of fields only requires multiplications and additions by a constant (which can sometimes be reduced to more efficient computations, *e.g.,* shift-and-add). This would also apply even if the primitive values of SHAPES$_\ell$ (integers and pointers) were of different sizes.

We also expect that, thanks to our runtime design decisions with respect to pools, SHAPES can be also implemented in unmanaged languages: Even if such memory optimisations can be performed manually in unmanaged languages, we argue that being able to implement them in an easy-to-use manner is beneficial.

Similar to SHAPES^h, extensions could also be added to SHAPES$_\ell$. One such extension to SHAPES$_\ell$ would be to include representations for more sophisticated layouts (*e.g.,* AoSoA, *cf.,* § 3).

Chapter 6

SHAPES-z: A DSL implementation of SHAPES

In § 4, we introduced SHAPES^h, a high-level formalism of SHAPES and provided its memory safety guarantees. in § 5, we introduced SHAPES$_\ell$, a low-level representation of SHAPES and argued that SHAPES$_\ell$ can be can be translated to existing low-level intermediate representations in an easy manner.

While formal models are undoubtedly necessary in terms of demonstrating that the desired properties of a language do hold, it is also vital for us to demonstrate the viability of an implementation and measure its performance. To that extent, we developed SHAPES-z, an unmanaged implementation of the SHAPES idea and an accompanying compiler; Listing 6.1 presents a SHAPES-z example.

SHAPES-z implements the idea of SHAPES as an embedded DSL; the developer writes their business logic in SHAPES-z and then invokes it from C/C++ code. SHAPES-z is also an unmanaged language; SHAPES-z code is compiled into assembly code; it does not run within the confines of a runtime environment, such as the JVM.

Our rationale for designing SHAPES-z as an embeddable language is due to practicality reasons; if we were to develop SHAPES-z as a standalone language, we would need to introduce *e.g.,*

rudimentary file I/O for unit tests or a custom benchmarking framework, which is bound to lack in features and possibly introduce unexpected biases. These side objectives would distract us from our original objective of a SHAPES implementation and could even result in unintentional feature creep seeping in.

Furthermore, we ruled out the possibility of extending an existing *host language* with SHAPES constructs; SHAPES is bound to present incompatibilities with some features of the host language. For instance, C unions would be incompatible with SHAPES; C `structs` can be passed by value; SHAPES objects can be only passed by reference. This implies that a subset of the host language would need to be carefully carved out to ensure smooth interoperability with SHAPES. Additionally, modifying an existing compiler so as to embed SHAPES might end up requiring a significant amount of tinkering and fighting with the source code of the host language's compiler and/or runtime (especially when having to take garbage collection into account). Besides, the amount of effort to experiment and bend (for practical reasons) the grammar and semantics of SHAPES-z within an existing language would certainly make performing iterations to the design of SHAPES-z more challenging.

Additionally, our rationale for making SHAPES-z unmanaged is to demonstrate our claim that SHAPES can achieve its aim of making it easy to experiment with and modify object layouts, even when interacting with other unmanaged languages, such as C & C++.

SHAPES-z is embeddable in a manner similar to that of Lua and Tcl; Lua [IdFC07] and Tcl [O+89] are designed to be embedded into and invoked from C & C++ code within existing applications. However, the main difference between SHAPES-z and, *e.g.,* Lua, is that Lua is a general purpose language; SHAPES-z does not aim to be one. Additionally, SHAPES-z is statically typed, whereas Lua is dynamically typed.

6.1 The SHAPES-z language

SHAPES-z expands on SHAPES by introducing additional features and constructs; Listing 6.1 presents some of them. These features include but are not limited to primitive types (Lines 4–

```
1  class Vec3<p>
2      where p: [Vec3<p>]
3  {
4      x: f32;
5      y: f32;
6      z: f32;
7
8      fn mag_sq(): f32 {
9          return x*x + y*y + z*z;
10     }
11
12     fn mixed_decls_code(n: u64): u64 {
13         n += 1;
14         pool q: Vec3Soa<q>;
15         return n;
16     }
17
18     fn count_equal(): u64 {
19         let n: u64 = 0;
20         foreach(v: p) {
21             if x != v.x { continue; }
22             if y != v.y { continue; }
23             if z != v.z { continue; }
24
25             n += 1;
26         }
27
28         return n;
29     }
30 }
31
32 layout Vec3Soa: Vec3 =
33     rec{x} + rec{y} + rec{z};
34
```

Listing 6.1: SHAPES-z feature showcase

6), expressions (Line 9), control flow statements (Lines 21–23), and compound assignment (Line 25), with semantics similar to those in other conventional languages.

Moreover, SHAPES-z also introduces more convenient syntax, such as the ability to mix pool declarations and code (thus allowing pools to be instantiated in the middle of a method, *cf.,* Line 14). Adding support for mixed pool declarations and code does not require any significant amendments; the SHAPES type system (§ 4.3) need not be modified to accommodate this feature. Additionally, SHAPES-z allows iteration over all objects contained inside a pool with the `foreach` statement (Line 20). Support for iteration over pools is required so that we can implement our case studies (*cf.,* § 7) in SHAPES-z.

SHAPES-z compilation generates object code (similar to C & C++), as well as a C/C++ header that allows external code to invoke SHAPES-z code directly. Given that we designed SHAPES-z to be embeddable, SHAPES-z has no notion of a `main()` function, hence it is up to external code to invoke SHAPES-z functions. Unlike Lua, SHAPES-z does not support the invocation of arbitrary C functions.

§ B presents the SHAPES-z grammar.

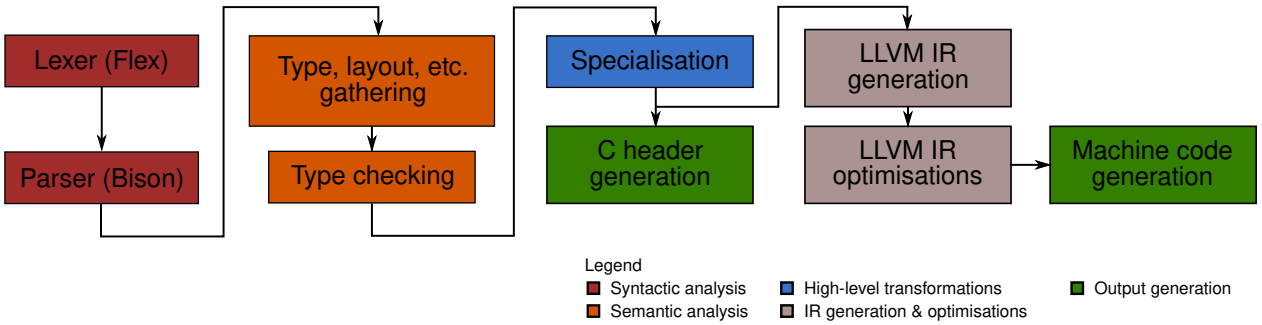


Figure 6.1: shapeszc compiler stages

6.2 shapeszc: The SHAPES-z compiler

We implemented shapeszc, a compiler for SHAPES-z written in C++; its repository is located at https://github.com/octurion/shapes-compiler. shapeszc amounts to approximately 8.6K source lines of code (SLoC) in total[1]. Figure 6.1 presents an overview of the compiler's architecture.

We will now present an overview of the shapeszc frontend and backend:

6.2.1 The shapeszc frontend

Lexing and parsing shapeszc uses GNU Flex [PEM16] and GNU Bison [DS21] for lexing and parsing. Given our choice of C++ as the implementation language for shapeszc, these seemed to be the most mature utilities with respect to writing C and C++ parsers.

We aimed to make use of their features to the largest possible extent; during development, we were pleasantly surprised to discover that Flex and Bison can be configured to support the generation of line and column number spans when parsing, as well as the ability of Bison to support error recovery, as well as operator precedence and associativity (or its lack thereof for comparison operators).

sloccount reports that the lexer and parser consist of approximately 1.8K SLoC.

After parsing, a syntax tree is generated and shapeszc proceeds to perform semantic analysis.

[1] Calculated by using sloccount [Whe04].

Semantic analysis Semantic analysis within SHAPES-z consists of collecting all types, layouts, pool names, etc, and then performing typechecking as described in § 4.3. This multi-pass approach is necessary because SHAPES (and, consequently, SHAPES-z) permits cyclic references between classes (*e.g.,* classes `A` and `B` can have fields of type `B` and `A`, respectively). This approach is similar to compilers such as `javac`[2].

`shapeszc` performs multi-pass semantic analysis as follows:

Pass 1 Collection of classes', fields', pool parameters', and methods' names.

Pass 2 Collection of class layouts.

Pass 3 Collection of fields', pool parameters', and methods' types.

Pass 4 Typechecking of all types collected from Phases 1, 2, and 3.

Pass 5 Collection and typechecking of the methods' bodies; annotation of all expressions in the AST with type information.

The end result of semantic analysis is the generation of an Abstract Syntax Tree (AST). At each pass, the AST is augmented with the additional information collected from it. The exception to this is *Pass 4*, which performs type checking, as described in § 4.3.

With respect to error reporting and multiple passes, we only proceed with the next pass if the current pass reported no errors. This is to prevent unnecessary "cascading" errors from being reported (*e.g.,* no typechecking in method bodies will be performed if a field of a class contains an undefined pool parameter). Moreover, we report all semantic errors on a specific pass, since we expect them to occur independently of all other reported errors.

`sloccount` reports that semantic analysis consists of approximately 4.0K SLoC.

Error reporting The features provided by Flex and Bison regarding line and column number spans, as well as the multiple pass design of `shapeszc` made it easier for us to output compiler

[2] https://github.com/openjdk/jdk/blob/739769c8fc4b496f08a92225a12d07414537b6c0/src/jdk.compiler/share/classes/com/sun/tools/javac/comp/TypeEnter.java

```
class Main<p1> where
    p1: [Main<p1>]
{
    f1: i32;
}

layout Foo: Main = rec{f1, f2};
layout Foo: Main = rec{f1};


```

Listing 6.2: Malformed SHAPES-z code

```
|Line 7| Field 'f2' has not been defined.
layout Foo: Main = rec{f1, f2};
                           ^^

|Line 8| Layout 'Foo' is already defined.
layout Foo: Main = rec{f1};
       ^^^

|Line 7| Existing definition is here:
layout Foo: Main = rec{f1, f2};
       ^^^
```

Listing 6.3: Compiler errors of Listing 6.2

errors in a manner that we claim that is not only user friendly, but also aesthetically pleasing. Listing 6.2 presents an example of malformed SHAPES-z code; Listing 6.3 presents the error messages generated by shapeszc for Listing 6.2.

Our implementation of compiler error reporting in shapeszc is heavily inspired by similar work performed in rustc, the Rust compiler [KN18]. Although the focus on outputting user friendly error messages might seem unnecessary at first, especially for a research project, it did pay dividends and saved us from lots of annoyances (*e.g.,* dealing with misleading and/or cryptic error messages) when coding our case studies (*cf.,* § 7) and shapeszc unit tests.

6.2.2 The shapeszc backend

Since we aim SHAPES-z to be an unmanaged language, we decided on using LLVM [LA04] within shapeszc for optimisation and code generation. LLVM is written in C++; while unofficial bindings for other languages do exist, we were not confident with regards to the maturity and stability of these bindings. This is the main reason why we decided to pick C++ as our implementation language (which also affected our choice of Flex and Bison in part).

The design of SHAPES, and, consequently, SHAPES-z, provides the potential to exploit a great variety of possible optimisations. For instance, when performing address calculation to access the fields of a pooled object, we can deduce that this address calculation can never overflow, hence by providing such a hint to the LLVM optimiser, we allow it to emit potentially better

optimised code. This is, effectively, our primary reason why we ruled out the possibility of compiling SHAPES-z code into C or C++ code: We expect to be able to provide more information to the optimiser compared to a conventional C/C++ compiler.

Because SHAPES-z is an unmanaged embeddable language, a scheme for invoking SHAPES-z code from the host language is necessary. To that extent, `shapeszc` generates a C/C++ header, so that SHAPES-z pools and objects can be instantiated from C/C++ code.

We now present how `shapeszc` performs code generation and optimisation; we will initially present a short introduction to the LLVM intermediate representation (IR) and then explain in detail how `shapeszc` makes use of LLVM to generate and optimise code. With respect to code generation, we follow § 5.5, with the exception that we are generating LLVM IR rather than SHAPES$_\ell$ code.

`sloccount` reports that LLVM IR generation and optimisation consists of approximately 2.4K SLoC.

An LLVM IR primer The LLVM IR [Pro21] is a statically typed, machine-independent intermediate representation (IR) used by LLVM. An LLVM IR source file corresponds to a *module*; a module defines and declares functions and global variables[3].

Similar to conventional intermediate representations, LLVM functions consist of *basic blocks*; each *basic block* consists of sequential instructions and is terminated by a branch or return instruction. Moreover, LLVM functions must adhere to what is known as Static Single Assignment (SSA) form [RWZ88]: Every local variable of an LLVM function must be assigned exactly once.

LLVM variables and expressions are typed; the LLVM type system bears quite a lot of resemblance to that of C and provides, among others, integer and floating-point types, structures (identical to C `structs`), and pointer types.

Listing 6.4 presents a C++ code example; Listing 6.5 presents its equivalent LLVM IR. Lines 1–

[3] `shapeszc` does not make use of global variables.

```
1 struct A {
2   int x;
3 };
4 struct B {
5   int y;
6 };
7 void copy(A* a, B* b) {
8   a->x = b->y;
9 }
```

Listing 6.4: C++ sample code

```
1 %st.A = type { i32 }
2 %st.B = type { i32 }
3 define void @copy(%st.A* %0, %st.B* %1) {
4   %3 = getelementptr %st.B, %st.B* %1, i64 0, i32 0
5   %4 = load i32, i32* %3, align 4
6   %5 = getelementptr %st.A, %st.A* %0, i64 0, i32 0
7   store i32 %4, i32* %5, align 4
8   ret void
9 }
```

Listing 6.5: Equivalent LLVM IR of Listing 6.4

2 declare two struct types corresponding to the struct definitions of Lines 1 and 4 in Listing 6.4. Line 3 declares function `foo`, along with its parameters and (lack of a) return type. Lines 4 and 6 are `getelementptr` expressions (also known as GEP expressions). A GEP expression takes one or more expressions as indices and performs address calculation; it can be used to calculate the address of the i-th element in an array and/or the k-th field of a struct. Line 4 calculates the address of field `y` within struct `%0`. Similarly, Line 6 calculates the address of field `x` within struct `%1`. Line 5 represents a memory load and corresponds the right-hand side of Line 8 within Listing 6.4; Line 7 represents a memory store and corresponds to the left-hand side of Line 8 within Listing 6.4.

Specialisation Before code generation takes place, `shapeszc` performs specialisation. As we described in § 5.5, during specialisation, each formal pool parameter is specialised to correspond to either a pool of a specific layout or the heap (**none**). This is done in a combinatorial manner, and all possible specialisations for each class is generated. Afterwards, for each specialisation, `shapeszc` generates all class methods, object/pool constructors and destructors.

C/C++ header generation We will now present an example of the C/C++ header generated by `shapeszc`. Listing 6.6 presents a two-dimensional vector with floating-point coordinates in SHAPES-z, along with a Struct-of-Arrays layout; Listing 6.7 presents the part of the header generated by `shapeszc` that corresponds to layout `Soa`. Line 5 is the definition of `Soa` pools; Lines 3–4 define the clusters' layouts. The representation of pools is identical to that of § 5 (with the exception of types).

```
1 class Vec2<p>
2   where p: [Vec2<p>]
3 {
4   x: f32;
5   y: f32;
6
7   fn mag_sq(): f32 { return x*x + y*y; }
8 }
9 layout Soa: Vec2 = rec{x} + rec{y};
```

Listing 6.6: 2D vector in SHAPES-z

Lines 16–19 are the declarations of the object and pool methods. Line 16 corresponds to the pool constructor. Line 17 corresponds to the pool destructor, which deallocates the pool, as well as all the objects contained within the pool. Line 18 corresponds to the object constructor, which allocates memory, constructs, and initialises an object inside the pool.

The rationale for exposing object and pool constructors and destructors to the host environment is practicality: Because SHAPES-z does not currently have expose a `main()` method or an entry point, invoking a SHAPES-z method requires pools and objects to have been constructed initially; after SHAPES-z code is executed, the host will most likely want to deallocate SHAPES-z pools and objects that have been constructed so as to be a "good citizen" and not, *e.g.,* leak memory.

Line 19 corresponds to method `mag_sq`; in the `Soa` specialisation of this method, two arguments need to be passed: The reference to the pool (which has layout `Soa`) and the reference to the **`this`** parameter. The reference to **`this`** is represented as a pool index of type **`uintptr_t`**, which is an integer type capable of holding any pointer [ISO12].

The type and method names in the header are mangled to prevent accidental name clashing. Since no C/C++ application binary interface (ABI) seems to permit hashes (#) in method names, we use a name mangling strategy that borrows heavily from ideas used for name mangling in the C++ Itanium ABI [ita]. For instance, class `Vec2`, specialised under the `Soa` layout, has its class name mangled as follows:

- The number of pool parameters in the class (one for `Vec2`), along with the character `C`.

```c
#include <stdint.h>

struct _shapes_cluster_0_1C4Vec23Soa { float x; };
struct _shapes_cluster_1_1C4Vec23Soa { float y; };
struct shapes_pool_1C4Vec23Soa {
  uintptr_t size;
  uintptr_t capacity;
  struct _shapes_cluster_0_1C4Vec23Soa* cluster0;
  struct _shapes_cluster_1_1C4Vec23Soa* cluster1;
};

#ifdef __cplusplus
extern "C" {
#endif

void _shapes1C4Vec23Soa_P(struct shapes_pool_1C4Vec23Soa*);
void _shapes1C4Vec23Soa_D(struct shapes_pool_1C4Vec23Soa*);
uintptr_t _shapes1C4Vec23Soa_C(struct shapes_pool_1C4Vec23Soa*);
float _shapes1C4Vec23Soa_M6mag_sq(uintptr_t, struct shapes_pool_1C4Vec23Soa*);

#ifdef __cplusplus
}
#endif
```

Listing 6.7: C++ header generated for Listing 6.6 (`Soa` layout only)

- The length of the SHAPES-z class name (`4`), followed by the class name `Vec2`.
- For each of the pool parameters, `N` if the specialisation corresponds to the heap, otherwise the layout name length (`3`), followed the layout name (`Soa`).

Hence, the mangled class name generated is `1C4Vec23Soa`.

Method names consist of the mangled class name and one of the following suffixes:

- `_C` for object constructors.
- `_P` for pool constructors.
- `_D` for pool destructors.
- `_M` for object methods, followed by the SHAPES-z name length, and the name itself (*e.g.*, `_M6mag_sq` for `mag_sq`).

```
%struct.pool.1C4Vec23Soa = type { i64, i64, %struct.cluster.0.1C4Vec23Soa*, %struct.cluster.1.1C4Vec23Soa* }
%struct.cluster.0.1C4Vec23Soa = type { float }
%struct.cluster.1.1C4Vec23Soa = type { float }

define float @_shapes1C4Vec23Soa_M6mag_sq(i64, %struct.pool.1C4Vec23Soa* nocapture readonly)
    local_unnamed_addr norecurse nounwind readonly {
entry:
  %2 = getelementptr inbounds %struct.pool.1C4Vec23Soa, %struct.pool.1C4Vec23Soa* %1, i64 0, i32 2
  %3 = load %struct.cluster.0.1C4Vec23Soa*, %struct.cluster.0.1C4Vec23Soa** %2, align 8
  %4 = getelementptr inbounds %struct.cluster.0.1C4Vec23Soa, %struct.cluster.0.1C4Vec23Soa* %3, i64 %0, i32 0
  %5 = load float, float* %4, align 4
  %6 = fmul float %5, %5
  %7 = getelementptr inbounds %struct.pool.1C4Vec23Soa, %struct.pool.1C4Vec23Soa* %1, i64 0, i32 3
  %8 = load %struct.cluster.1.1C4Vec23Soa*, %struct.cluster.1.1C4Vec23Soa** %7, align 8
  %9 = getelementptr inbounds %struct.cluster.1.1C4Vec23Soa, %struct.cluster.1.1C4Vec23Soa* %8, i64 %0, i32 0
  %10 = load float, float* %9, align 4
  %11 = fmul float %10, %10
  %12 = fadd float %6, %11
  ret float %12
}
```

Listing 6.8: LLVM IR for Soa layout of Listing 6.6

shapeszc uses the LLVM API to build the IR representation; Listing 6.8 presents a simplified version (for the sake of brevity) of the LLVM IR code generated for method mag_sq of Listing 6.6, when specialised under the Soa layout.

The LLVM IR generation aspect of shapeszc is, for the most part, conventional and unsurprising. The only noteworthy aspect is the fact that the object/pool construction and allocation logic is also implemented within LLVM IR, but performs calls to malloc(), realloc(), and free() (or semantically equivalent functions corresponding to a custom memory allocator). This is the only case where we depend on external functions in SHAPES-z.

This minimal dependency only on a memory allocator is made possible thanks to specialisation. A welcome side effect of specialisation is the massive simplification with respect to code generation: Thanks to specialisation, the memory representation of objects and pools is always known at compile time; runtime introspection to determine the location of an object's field (via, *e.g.,* "field virtual tables") is therefore unnecessary. An elaborate scheme to represent a reference to a pooled object is also superfluous: Pools are represented in such a way that only an index into a pool is necessary to reference an object in it; we can always determine at compile time the pool an object belongs to, as well as the layout of that specific pool.

Due to historical reasons, LLVM implements two pass managers: A *new pass manager* (which

makes use of `llvm::PassManagerBuilder`[4]) and a *legacy pass manager* (which makes use of `llvm::legacy::PassManager`[5]).

SHAPES-z performs optimisation on the LLVM IR with the *new pass manager*, which "set[s] up a standard optimization sequence for languages like C and C+". Our setup of the optimisation pipeline uses LLVM's TBAA pass § 6.2.3 explicitly and is set to an optimisation level of `OptimizationLevel::O3`[6]. Generation of machine code is performed using LLVM's *legacy pass manager* (due to technical reasons, we could not use the new pass manager for machine code generation); while setting an optimisation level for machine code emission seemed impossible, we observed that the machine code generated seemed optimised for trivial examples (*e.g.,* no redundant instructions, register allocation seemed to have been performed).

6.2.3 Alias analysis in SHAPES-z

In typed programming languages, in many cases, references to objects of distinct types are guaranteed to never alias. For instance, two references to a Java `ArrayList` and `HashMap`, respectively, are guaranteed to never alias. According to the C standard, if `T` and `U` are distinct types, two pointers of type `T*` and `U*`, respectively, can be assumed in most cases[7] to never alias. This is known as the *strict aliasing rule* [fS18].

This observation allows us to perform what is known as *Type-based Alias Analysis* [DMM98], also known as TBAA.

A benefit regarding the design of SHAPES, and, therefore, SHAPES-z is that we can expand on the idea of TBAA beyond the scope of merely standalone objects: References to pools of different layouts are guaranteed to never alias, therefore references to objects that are known

[4] https://llvm.org/doxygen/classllvm_1_1PassManagerBuilder.html
[5] https://llvm.org/doxygen/classllvm_1_1legacy_1_1PassManager.html
[6] https://llvm.org/doxygen/classllvm_1_1OptimizationLevel.html#a097296a5feaefc188dafa71b19204714
[7] Some exceptions apply. For instance, `char*` pointers are permitted to alias to any other pointer. This is because, similar to `void*` pointers, a pointer of an arbitrary type can be converted to `char*` and back and yield the original pointer [fS18].

```
1  %struct.A = type { i32 }
2  %struct.B = type { i32 }
3
4  define void @copy(%struct.A* %0, %struct.B* %1) {
5    %3 = getelementptr %struct.B, %struct.B* %1, i64 0, i32 0
6    %4 = load i32, i32* %3, align 4, !tbaa !4 ; TBAA access tag
7    %5 = getelementptr %struct.A, %struct.A* %0, i64 0, i32 0
8    store i32 %4, i32* %5, align 4, !tbaa !5
9    ret void
10 }
11
12 !0 = !{!"TBAA_root"} ; Root node for all TBAA metadata
13 !1 = !{!"int", !0, i64 0} ; Scalar type `i32`
14 !2 = !{!"A", !1, i64 0, !1, i64 4} ; Struct type `A`
15 !3 = !{!"B", !2, i64 0, !1, i64 8} ; Struct type `B`
16 !4 = !{!3, !1, i64 0} ; Field access for field `A.x`
17 !5 = !{!4, !1, i64 8} ; Field access for field `B.y`
```

Listing 6.9: LLVM IR of Listing 6.4, augmented with TBAA metadata.

to belong to different pools will also never alias; a reference to a standalone object can never alias to a reference to a pooled object. It is thus natural to attempt to take advantage of these guarantees provided by SHAPES/SHAPES-z to produce more efficient code.

The LLVM IR type system, however, provides us with no implicit or explicit guarantees about type-based aliasing; LLVM mandates that it is well-defined for any pointer of type `T*` to be reinterpreted as a pointer of type `U*`[8]. Therefore, LLVM has to conservatively assume that pointers of different types may alias [Pro21]; any information regarding type-based aliasing will need to be provided in a different manner.

Indeed, within LLVM, taking advantage of TBAA (or any other alias analysis scheme for that matter) requires the IR to be annotated with additional metadata. The two kinds of TBAA metadata that can be supplied to LLVM are as follows:

- Type descriptors corresponding to the scalar types (Line 13) and struct types (Lines 14–15) being used. Struct type descriptors must also specify the fields' types and offsets of a struct type.

[8] In fact, this is how *e.g.,* unions and inheritance can be implemented without additional LLVM language constructs.

- Access tags annotations on load and store operations (Lines 6 and 8). Each access tag consists of the scalar type and offset of the (possibly nested) field being accessed.

As an example, consider Listing 6.4; due to the *strict aliasing* rule of C++ [ISO12], two respective pointers to structs `Foo` and `Bar` cannot alias. Listing 6.9 presents the equivalent LLVM IR representation for Listing 6.4, now enriched with TBAA metadata to explicitly express this aliasing constraint.

Supplying this metadata for standalone SHAPES-z objects is conceptually simple. The challenge lies on whether or not we can use TBAA metadata for pools and pooled objects. Given the fact that we make use of method specialization, we statically know the layout of each pool in a given method. As such, supplying TBAA metadata to handle pooled objects turns out to be quite easier than anticipated.

Consider Listing 6.6; in method `mag_sq`, under an `Soa` specialisation, we can guarantee that fields `x`, `y`, and `z` are each stored within separate clusters. As such, we ideally want to express this property with TBAA metadata, so that LLVM can exploit this, if possible. Listing 6.10 presents the LLVM IR of method `mag_sq`, now augmented with TBAA metadata.

Due to the presence of pools and layouts in SHAPES-z, we expand on the approach of generating TBAA metadata compared to a "conventional" procedural language as follows:

- Every pool header gets associated with a TBAA struct type of a specific class and layout (Line 29); this ensures that pointers to pools of different types and/or layouts will be considered unaliasable.
- The clusters of each pool are represented as an array of record splits; we associate each record split with a distinct TBAA struct type (Lines 32–33). This ensures that structurally equivalent but distinct record splits are also considered to never alias. This is what ensures two pooled objects that differ in their layout to be considered unaliasable: Indeed, given a field `f` in class `C`, and two layouts `L1` and `L2`, the value of `f` will be stored

```
%struct.pool.1C4Vec23Soa = type { i64, i64, %struct.cluster.0.1C4Vec23Soa*, %struct.cluster.1.1C4Vec23Soa* }
%struct.cluster.0.1C4Vec23Soa = type { float }
%struct.cluster.1.1C4Vec23Soa = type { float }

define float @_shapes1C4Vec23Soa_M6mag_sq(i64, %struct.pool.1C4Vec23Soa* nocapture readonly) {
entry:
  %2 = getelementptr inbounds %struct.pool.1C4Vec23Soa, %struct.pool.1C4Vec23Soa* %1, i64 0, i32 2
  %3 = load %struct.cluster.0.1C4Vec23Soa*, %struct.cluster.0.1C4Vec23Soa** %2, align 8, !tbaa !10
  %4 = getelementptr inbounds %struct.cluster.0.1C4Vec23Soa, %struct.cluster.0.1C4Vec23Soa* %3, i64 %0, i32 0
  %5 = load float, float* %4, align 4, !tbaa !12
  %6 = fmul float %5, %5
  %7 = getelementptr inbounds %struct.pool.1C4Vec23Soa, %struct.pool.1C4Vec23Soa* %1, i64 0, i32 3
  %8 = load %struct.cluster.1.1C4Vec23Soa*, %struct.cluster.1.1C4Vec23Soa** %7, align 8, !tbaa !11
  %9 = getelementptr inbounds %struct.cluster.1.1C4Vec23Soa, %struct.cluster.1.1C4Vec23Soa* %8, i64 %0, i32 0
  %10 = load float, float* %9, align 4, !tbaa !13
  %11 = fmul float %10, %10
  %12 = fadd float %6, %11
  ret float %12

!0 = !{!"shapes_tbaa_root"} ; TBAA root
!1 = !{!"intptr", !0, i64 0} ; Scalar type for pool index
!2 = !{!"f32", !0, i64 0} ; Scalar type `f32`

; Scalar type for pointer to clusters for layout Soa
!3 = !{!"_tbaa_cluster_ptr0_1C4Vec23Soa", !0, i64 0} ; Cluster 0
!4 = !{!"_tbaa_cluster_ptr1_1C4Vec23Soa", !0, i64 0} ; Cluster 1

; Struct type for pool header for Soa
!5 = !{!"_tbaa_pool1C4Vec23Soa", !1, i64 0, !1, i64 8, !3, i64 16, !4, i64 24}

; Struct type for the clusters of Soa
!6 = !{!"_tbaa_cluster0_1C4Vec23Soa", !2, i64 0} ; Cluster 0
!7 = !{!"_tbaa_cluster1_1C4Vec23Soa", !2, i64 0} ; Cluster 1

; Field accesses
!8 = !{!5, !1, i64 0} ; Pool size
!9 = !{!5, !1, i64 8} ; Pool capacity
!10 = !{!5, !3, i64 16} ; Pool cluster 0 ptr
!11 = !{!5, !4, i64 24} ; Pool cluster 1 ptr
!12 = !{!6, !2, i64 0} ; Field x
!13 = !{!7, !2, i64 0} ; Field y
```

Listing 6.10: LLVM IR of Listing 6.6, augmented with TBAA metadata.

in record splits of different TBAA struct types, accessing `f` in pooled objects of different layouts will be always considered to never alias.

As we can see, the design of SHAPES/SHAPES-z does allow alias analysis to be performed by leveraging already existing schemes (TBAA). What is more, we were initially expecting support for alias analysis to be a daunting and gruelling task; we were pleasantly surprised to find out the `llvm::MDBuilder` class[9] made this process much than anticipated.

One aspect that our current `shapeszc` implementation does not consider with respect to aliasing is that for two different objects o_1, o_2 belonging to the same pool and a field f, the memory locations where $o_1.f$ and $o_2.f$ are placed are guaranteed to never alias. This fact cannot be

[9] https://llvm.org/doxygen/classllvm_1_1MDBuilder.html

provided to LLVM through TBAA, but it can be provided through LLVM's `noalias` feature[10]; we leave this to a future implementation.

6.3 Conclusion

We have presented SHAPES-z, an implementation of SHAPES as an unmanaged embedded language, and its design considerations. We have also presented `shapeszc`, a compiler for SHAPES-z, as well as its implementation considerations, both with respect to the frontend (*i.e.,* lexing, parsing, semantic analysis) and with respect to the backend (*i.e.,* LLVM IR code generation and optimisation via LLVM). We also do not expect any of our design considerations with respect to SHAPES-z to hinder any possible further development or introduction of further extensions.

A noteworthy aspect of `shapeszc` is that, with respect to its design, it does not deviate in a significant manner from either SHAPES^h (§ 4) or SHAPES_ℓ (§ 5). That is, our implementation of SHAPES-z follows our formalism: We did not need to resort to additional decisions when implementing type checking; merely following the definition for a well-formed SHAPES program was sufficient.

Furthermore, we did not need to make significant changes or sacrifices to adapt SHAPES_ℓ into LLVM, with the exception of name mangling. This is to be expected, however, as we had to provide a name mangling scheme that allowed code to be invoked from C and C++, as well as not cause accidental collisions with C or C++ symbols.

[10] https://llvm.org/docs/LangRef.html#noalias-and-alias-scope-metadata

Chapter 7

Case studies

We will now investigate the usefulness of the concepts of SHAPES; we consider a sequence of examples and discuss whether the use of SHAPES turns out to be beneficial for readability, maintainability, and performance.

To that extent, we implemented *five case studies*. These case studies are:

- *OP2* (§ 7.2): Numerical computations on unstructured grids.
- *3D Skeletal animation* (§ 7.3): Animation of 3D models with skeletal animation and calculation of relevant attributes (*e.g.,* positions of vertices) on each frame.
- *Currency* (§ 7.4): Simple querying on a sorted flat database.
- *Traffic* (§ 7.5): Traffic simulation on a street network with optional speed limits and traffic lights.
- *Doors* (§ 7.6): Small case study with actors and doors, each of a specific allegiance; we must determine which doors should be opened with respect to proximity of characters of the same allegiance.

For our case studies, we selected examples that should ideally consist of a non-trivial SLoC count and/or correspond to real-world use cases. We group these case studies into 2 categories:

- *Different layouts*: Case studies *OP2* and *Skeletal Animation* are mainly concerned with switching between AoS, Mixed, and SoA layouts. In *OP2*, we compare against an existing open source library that provides a more limited form of pooling and clustering compared to what is achievable by SHAPES. *Skeletal Animation* explores the use of different layouts to determine the fastest layout for a specific algorithm, as does *Traffic*.

- *Multiple pools*: Case study *Currency* is mainly concerned with the usefulness of multiple pools. It reflects a query system with real-world data and made-up queries, with observable performance improvements occurring from using multiple pools of the same class and having each pool use a different layout. *Doors* also partitions objects into pools to improve performance.

Due to practical considerations (our implementation of shapeszc was still underway at the time), we initially hand-compiled them into equivalent C++ code. § 7.2–7.6 evaluate the performance of the handwritten C++ variants of these case studies.

After we completed our implementation of shapeszc, we wrote SHAPES-z versions of the *Doors* and *Currency* case studies; in § 7.7, we evaluate the performance of our SHAPES-z impemention against our original handwritten C++ code with respect to these two case studies.

We group our case studies into 2 categories:

– *Different layouts*: Case studies *OP2* (§ 7.2), *Skeletal Animation* (§ 7.3), and *Traffic* (§ 7.5) are mainly concerned with switching between AoS, Mixed, and SoA layouts. In *OP2*, we compare against an existing open source library that provides a more limited form of pooling and clustering compared to what is achievable by SHAPES. *Skeletal Animation* and *Traffic* explore the use of different layouts to determine the fastest layout for a specific algorithm.
– *Multiple pools*: Case study *Currency* (§ 7.4) is mainly concerned with the usefulness of multiple pools. It reflects a query system with real-world data and made-up queries, with observable performance improvements occurring from using multiple pools of the same class and having each pool use a different layout. *Doors* (§ 7.6) also partitions objects into pools to improve performance.

Additionally, we make the following claims regarding our case studies with respect to SHAPES:

C1 The use of SHAPES should make it easy for the developer to experiment with various layouts to determine the most optimal one for each domain, thus providing a potential improvement in readability and maintainability.

C2 This ease in development SHAPES provides means that a developer can expect performance that is on par or better to already existing solutions with respect to pooling and clustering.

C3 There is merit in making SHAPES more flexible and allowing the developer to use Mixed layouts; that is, there are cases where Mixed layouts outperform AoS and SoA.

C4 Placing objects of the same type into multiple pools (rather than one pool) can improve cache utilisation and/or allow further algorithmic improvements.

§ 7.8 presents our conclusions, as well as our judgement on whether or not Claims C1–C4 are, indeed, satisfied.

The C++ code for our our case studies is publicly available[1].

7.1 Implementation & evaluation considerations for our case studies

Given the absence of pools and layouts in C++, we made use of C++-specific features that are not present in SHAPES-z (*e.g.,* template metaprogramming) so as to make our code more succinct. Although such features in SHAPES-z would be certainly convenient to have, it is still possible to write them in SHAPES-z by making use of alternative constructs.

Three of our case studies (*Currency*, *Traffic*, *Doors*) generate their datasets randomly (fully or in part) by using the C++11 Mersenne Twister RNG [ISO12]. To ensure that we are not

[1] https://github.com/octurion/ecoop-artifact

Name	CPU (Intel)	L3 size	RAM size - type - MHz	OS	Compiler
desktop	i7-6700K	8 MB	16 GB - DDR3 - 2133	Ubuntu 16.04	gcc 5.4.0
graphic	Xeon E5-1620 v2	10 MB	16 GB - DDR3 - 1866	Ubuntu 18.04	gcc 7.4.0
voxel	i7-4790	8 MB	16 GB - DDR3 - 1600	Ubuntu 18.04	gcc 7.4.0
ray	i7-7700K	8 MB	16 GB - DDR4 - 2400	Ubuntu 18.04	gcc 7.4.0

Table 7.1: Machine specifications

```
1  class Point<pPt: [Point<pPt>]>
2    {x: double; y: double;}
3  class Segment<
4      pSeg: [Segment<pSeg, pPt>],
5      pPt: [Point<pPt>]> {
6    p1: Point<pPt>; p2: Point<pPt>;
7    def len(): double {
8      var dx = p2.x - p1.x, dy = p2.y - p1.y;
9      return sqrt(dx * dx + dy * dy);
10   }
11 }
12 layout PointL: Point = rec{x} + rec{y};
13 layout SegmentL: Segment = rec{p1} + rec{p2};
14
15 def main() {
16   pools pPt1: PointL<pPt1>,
17         pSeg1: SegmentL<pSeg1, pPt1>;
18   ... // Create objects in pPt1, pSeg1
19   print(sum_lens_shapespp<pPt1, pSeg1>());
20 }
21 <ps: [Segment<ps, pp>], pp: [Point<pp>]>
22 def sum_lens_shapespp(): double {
23   var sum = 0;
24   foreach (var e: ps)
25     sum += e.len();
26   return len_sum;
27 }
28 ...
```

Listing 7.1: Example SHAPES-z code

introducing any accidental randomness bias, we use 100 seeds derived from the first 500 decimal digits of π (first 5 decimal digits correspond to the first seed, next 5 correspond to the second seed, *etc.*).

We ran our case studies on four machines; Figure 7.1 lists their specifications. We used `CMake` as our build system; all case studies were compiled as a `Release` build (which implies the `-O3 -DNDEBUG` flags). We used the Google C++ Benchmark library for our measurements[2] except for *OP2*; its long running time renders this library useless in this case. Instead, we measure wall clock time (`CLOCK_REALTIME`), which is what both OP2 programs measure as well.

When we compare SHAPES-z against our handwritten C++ code, we do not make use of the `voxel` machine; this is due to the fact that this machine had been decommissioned by the time we had completed our implementation of `shapeszc`.

With respect to the presentation of our results, we mainly use *notched* box plots; a notched box plot introduces a narrowing around the box plot median which indicates the 95% confidence interval. We also use the geometric mean to derive summary data.

[2] https://github.com/google/benchmark

```
1  class Point { double x, y; };
2  class Segment { Point *p1, *p2; };
3
4  void sum_lens(double* acc,
5                double* x1, double* y1,
6                double* x2, double* y2) {
7    double dx = *x2 - *x1, dy = *y2 - *y1;
8    acc += sqrt(dx * dx + dy * dy);
9  }
10 ...
11 op_set segs = op_decl_set(NUM_SEGS, "segs");
12 op_set points = op_decl_set(NUM_POINTS, "points");
13
14 double* x_data = calloc(NUM_POINTS, sizeof(*x_data));
15 double* y_data = calloc(NUM_POINTS, sizeof(*y_data));
16 double* p1_data = calloc(NUM_SEGS, sizeof(*p1_data));
17 double* p2_data = calloc(NUM_SEGS, sizeof(*p2_data));
18
19 ... // Fill x_data, y_data, p1_data, p2_data
20 op_dat x = op_decl_dat(points, 1, "double", x_data, "x");
21 op_dat y = op_decl_dat(points, 1, "double", y_data, "y");
22 op_map seg_p1 =
23   op_decl_map(segs, points, 1, p1_data, "seg_p1");
24 op_map seg_p2 =
25   op_decl_map(segs,points, 1, p2_data, "seg_p2");
26 ...
27
28 ...
29 double sum = 0;
30 op_par_loop(sum_lens, segs,
31   op_arg_gbl(&sum, 1, "double", OP_INC),
32   op_arg_dat(x,1,seg_p1,1,"double",OP_READ),
33   op_arg_dat(y,1,seg_p1,1,"double",OP_READ),
34   op_arg_dat(x,1,seg_p2,1,"double",OP_READ),
35   op_arg_dat(y,1,seg_p2,1,"double",OP_READ));
36 ...
```

Listing 7.2: Equivalent OP2 code for Listing 7.1

We now present our case studies in detail.

7.2 OP2

As we mentioned in § 3.1.2, OP2 [GMS+11] is a C++ library intended for computations on unstructured grids and is mainly focused on easing parallelisation of such applications (via, *e.g.,* MPI [for15], OpenMP [Boa18], CUDA [NC22]). OP2 mainly attempts to tackle the issue of executing a kernel over a set of data in parallel in a declarative manner. Moreover, it also provides capabilities for pooling and clustering, albeit more limited compared to those of SHAPES.

We will first introduce OP2 through an artificial example that calculates the sum of the lengths of line `Segments`: Listing 7.1 presents the corresponding SHAPES-z code. Lines 1–11 present the `Point` and `Segment` types, respectively; we will be using an SoA layout for both (Lines 12–13). Method `sum_lens_shapespp` (Line 22) calculates the length sum by traversing all objects in `pSeg1` (Line 24).

We present the equivalent OP2 code in Listing 7.2; In OP2, objects of the same type can be grouped into *sets* (Lines 11–12). To perform clustering, the developer must *allocate and fill in* the data of the clusters *manually* (Lines 14–19); OP2 will then keep track of the clusters (Lines 20–25) so as to access the appropriate fields during kernel execution. *Maps* (Lines 22–

25) correspond to references to objects in other sets, but the developer has to *manually* use an *index* to create a reference to an object in a set. As observed, clustering and set creation have to be performed at runtime and in an ad-hoc and type-unsafe manner.

Execution of kernel `sum_lens` will run over the line `Segments` in `segs` (Line 30). OP2 will obtain pointers to the `x`, `y` components of `p1` and `p2` for each `Segment`; these correspond to the parameters of `sum_lens` (Lines 4–6). The process of obtaining pointers to the fields of `Points` is not automatic; the arguments of Lines 32–35 specify how OP2 will obtain these pointers (in this case, by dereferencing fields `p1` and `p2` of each point. It is easy to see that, compared to SHAPES-z, readability and type safety must be sacrificed in order to use OP2 and improve performance.

The OP2 project provides two example C++ case studies that make use of parallelism (through OpenMP). These are called `airfoil` and `aero`.

`aero` consists of 408 SLoC (in C++)[3], out of which 311 SLoC correspond to the actual computations. We implemented `aero` (including the exact original OpenMP directives being used) in C++ in a manner that is identical to the runtime pool representation used by SHAPES-z and compare a *Mixed* and *AoS* version of our implementation against the original.

`airfoil` consists of 321 SLoC (in C++), out of which 282 correspond to the actual calculations. We implemented `airfoil` in C++ in a manner that is identical to the runtime pool representation used by SHAPES-z and compare a *Mixed*, *AoS*, and *SoA* version of our implementation against the original (Figure 7.3).

The original implementations of both `aero` and `airfoil` use a *Mixed* layout. Results for `aero` and `airfoil` are presented in Figures 7.2 and 7.3, respectively. Our measurement methodology is to run each variant of Airfoil and Aero on each machine 20 times.

With respect to `aero`, we observe that our *Mixed* implementation (geometric mean for all executions of 10.799 s) is virtually identical to that of the original OP2 implementation (geometric mean for all executions of 10.837 s); we also argue that a rewrite in SHAPES-z would also

[3] Calculated using `sloccount` [Whe04].

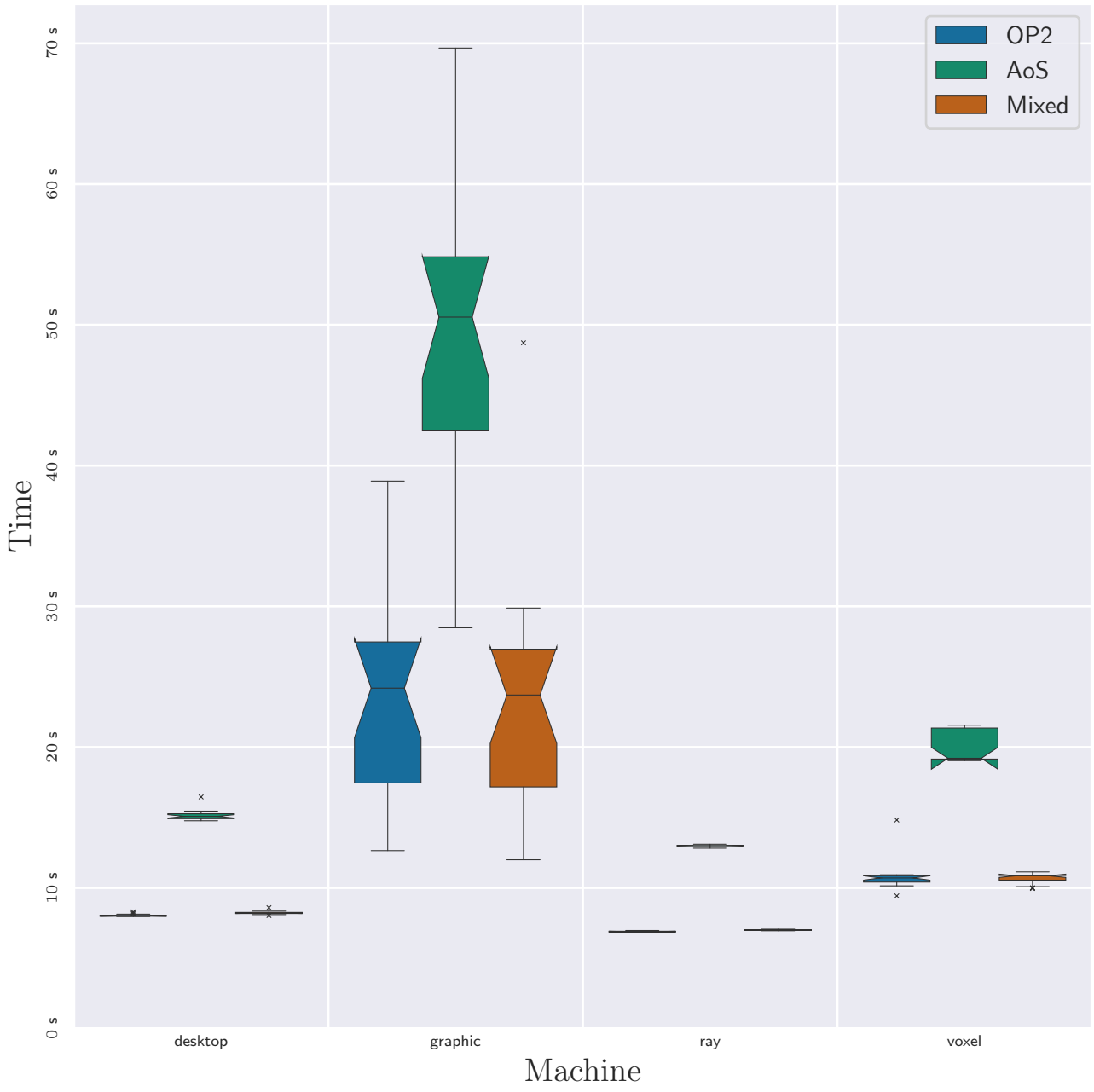


Figure 7.2: *OP2 Aero* results for the *original OP2* implementation, and the *AoS* and *Mixed* ports, respectively. (Lower times are better)

improve readability, usability, and type safety. We argue that this supports Claim C2.

Moreover, we observe that with respect to our implementation, *SoA* outperforms *AoS* (no loading of unrelated fields in memory, hence no cache pollution), yet *Mixed* outperforms *SoA*. This supports Claim C3. We speculate *Mixed* fares better for two reasons:

- When accessing a field `f` of an object in an *SoA* layout, it is possible that the values corresponding to field `f` of adjacent objects are also loaded into the cache due to spatial

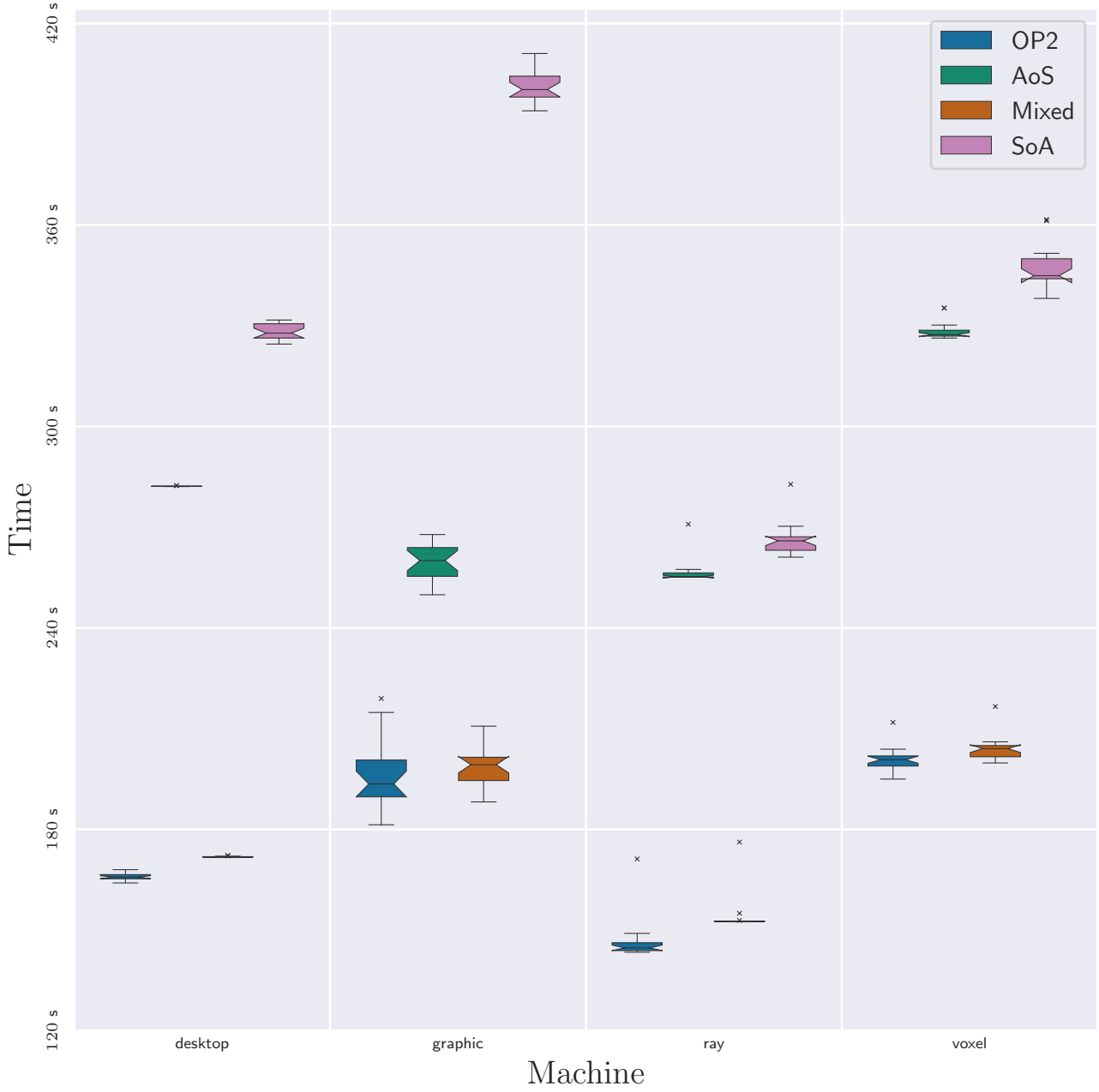


Figure 7.3: *OP2 Airfoil* results for the *original OP2* implementation, and the *AoS*, *Mixed*, and *SoA* ports, respectively. (Lower times are better)
NB: X axis has been trimmed at the left to save space.

locality. If we are accessing objects laid out in SoA in a random manner (in this case indirectly), this loading of adjacent values will amount to cache pollution.

- The hardware prefetcher can only keep track of up to a specific number of sequential access patterns [Fog12]; the prefetcher cannot keep up with the excessive clustering resulting from the *SoA* layout.

Results for `airfoil` are presented in Figure 7.3; we observe that these results follow nearly identical patterns to those of our `aero` comparison: Our *Mixed* implementation (geometric mean for all executions of 180.946 s) is almost identical (albeit with a small slowdown) to the original `airfoil` implementation (geometric mean for all executions of 175.628 s). Despite this slowdown, we argue it would improve readability, usability, and type safety if it were to be written in SHAPES-z, hence we argue that this is a worthwhile tradeoff and that it supports Claim C2.

7.3 3D skeletal animation

In the MD5Anim [Hen05] skeletal animation format, a 3D model ("stickman") consists of joints, weights, and vertices. Joints are organised in a tree; there is a 1-N relationship between joints and weights and a 1-N relationship between vertices and weights.

Animation of the model includes the following 2 phases:

Phase 1 Calculate the joints' new orientations in a top-down recursive manner.

Phase 2 Calculate the position of each weight from the weight's current position and the orientation of the joint it belongs to.

Our case study consists of creating instances of such stickmen from given data, then measuring the time taken to animate them.

It is not initially obvious how to layout our data in an optimal manner. We decided to focus on the following two "axes" of data layouts:

- Joints are either *Scattered* in memory (*i.e.,* **none** pool) or the joints for one instance of a "stickman" are placed close to each other in memory in an array, hence *Pooled*.
- Weights use an *AoS*, an *SoA*, or a *Mixed* layout.

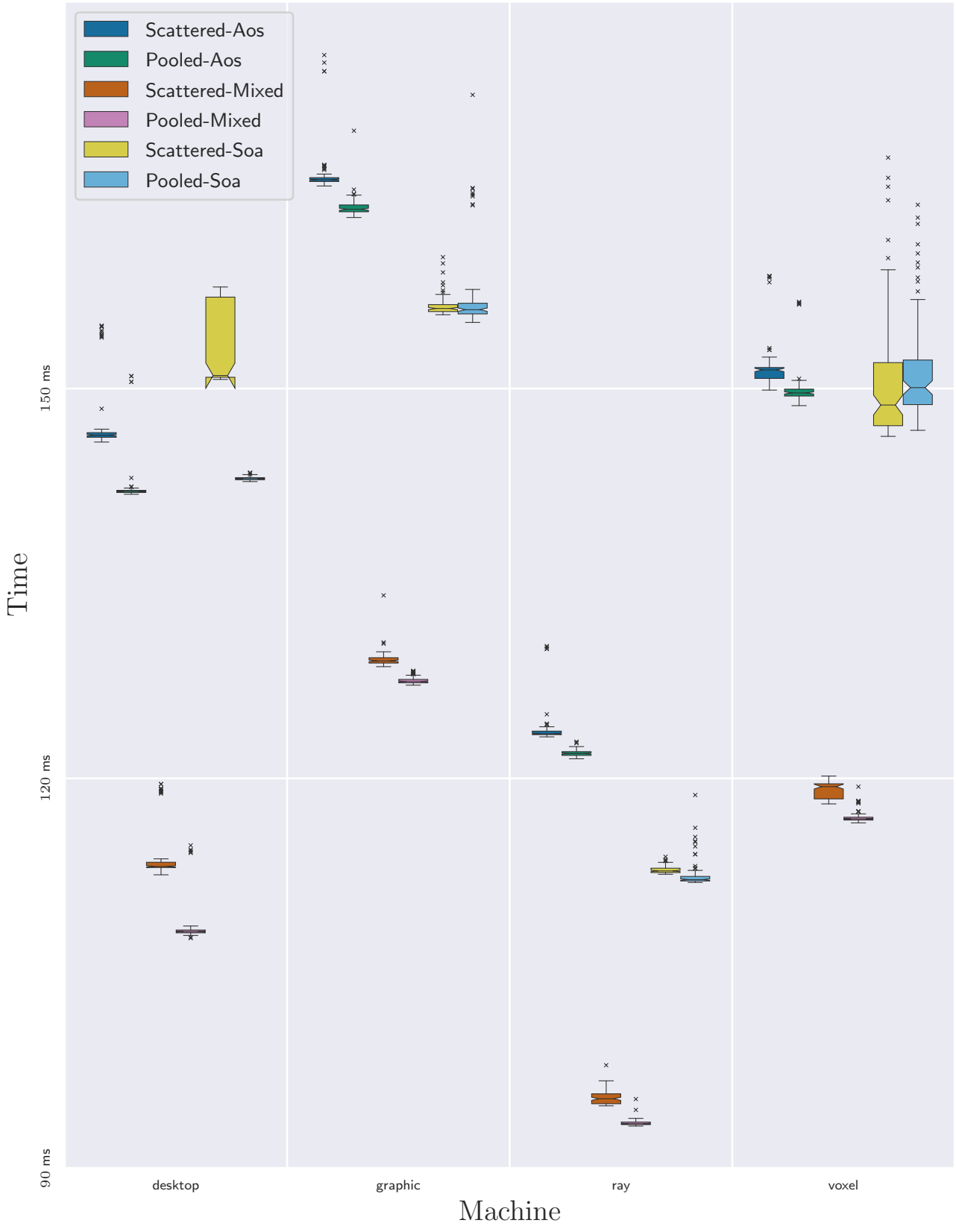


Figure 7.4: *Skeletal animation* results for *Scattered-AoS*, *Pooled-AoS*, *Scattered-Mixed*, *Pooled-Mixed*, *Scattered-SoA*, and *Pooled-SoA* layouts, respectively (where $N = 5000$). (Lower times are better)
NB: Y axis has been trimmed at the bottom to save space.

This results in 6 possible data layouts. It is not obvious at first which of these data layouts is optimal for the animation algorithm. Unfortunately, since we are handcoding our implementation in when writing it in C++, we have to manually write all 6 versions of it (once per layout). This is a monotonous, time-consuming, and error-prone task. Indeed, a striking observation regarding our C++ implementation is that we have duplicated the code that performs the necessary mathematical operations, in order to deal with the different layouts.

With SHAPES-z, however, we would only need to write the two and three possible layouts for the joints and weights, respectively and since model animation is oblivious to layouts of the pools we are using, we only need to modify their layouts at the site they are defined, then measure. We argue that this supports Claim C1.

Figure 7.5 presents our results. To make the differences in execution more visible, we ran our code 100 times for the case where $N = 5000$; we present the results of this specific execution in Figure 7.4. Notice that due to the way we were making use of Google Benchmark, the extraction of confidence intervals for each data point is unfortunately infeasible.

In Figure 7.4, we observe a form of "tiering": A couple of layouts have almost identical performance and have consistently the best times (the "fast tier"), whereas the remaining pools lag behind them, all close to each other (the "slow tier").

We observe that *Pooled* joints outperform *Scattered* ones, irrespective of the layout used for the weights. This is expected (due to better cache locality), but the speedup is not significant; this can be explained by the expectation that the number of joints in a 3D model will be much smaller compared to the number of weights.

Additionally, we observe that, similar to *OP2*, the *Mixed* layout outperforms both *AoS* and *SoA*. We speculate that this occurs for the exact same reasons as the *Mixed* layout used in OP2 (§ 7.2).

Therefore, the optimal layout for the animation algorithm is *PooledMixed* (and both Figure 7.4 and Figure 7.5 support this). To that extent, we argue that there is, indeed, merit in making use of mixed layouts (Claim C3).

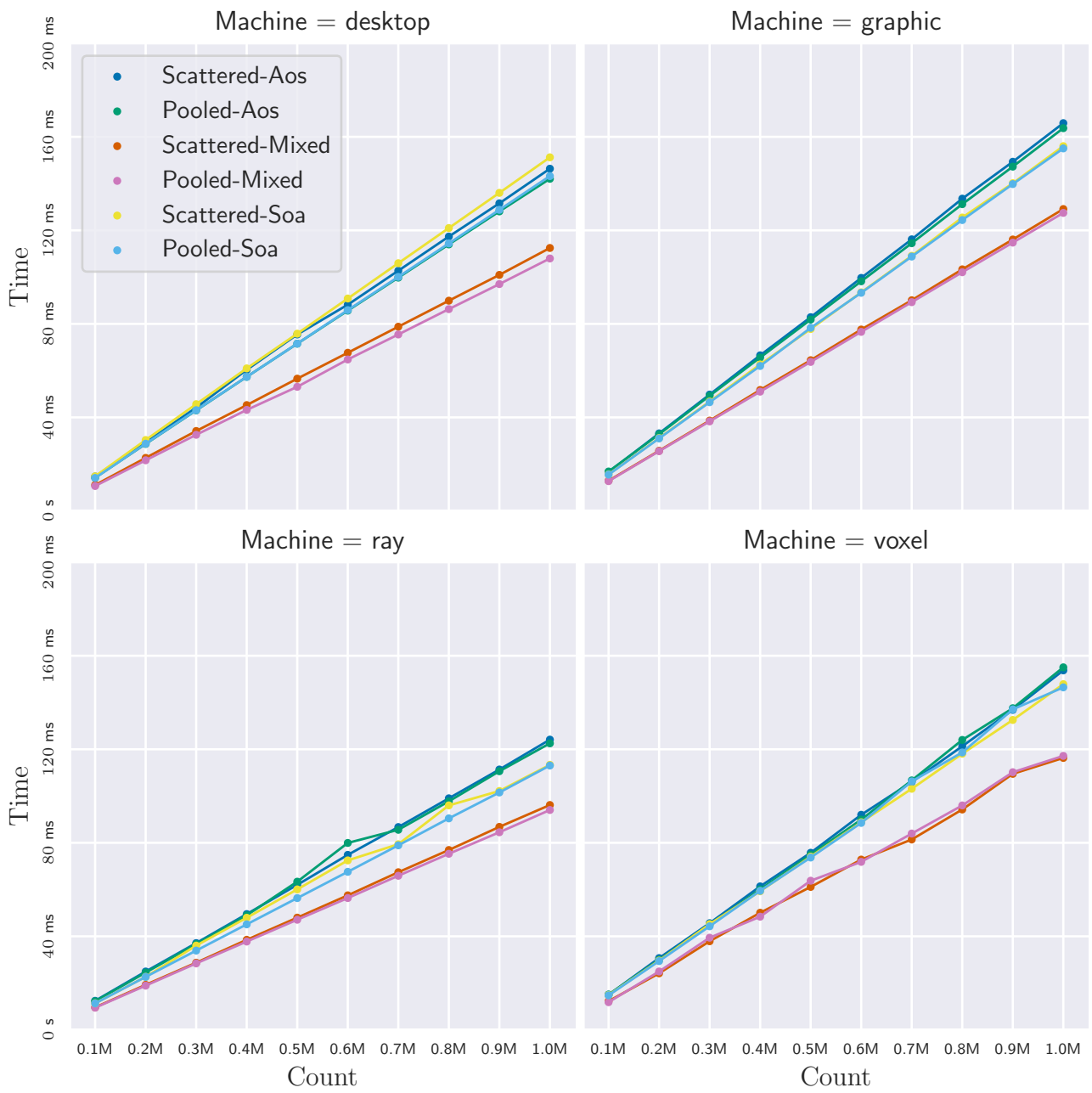


Figure 7.5: *Skeletal animation* results for *Scattered-AoS*, *Pooled-AoS*, *Scattered-Mixed*, *Pooled-Mixed*, *Scattered-SoA*, and *Pooled-SoA* layouts, respectively, with N being variable. (Lower times are better)

7.4 Currency

The European Central Bank keeps a record of all daily exchange rates of 41 currencies against the Euro since 1st January 1999[4]. As a case study, we implemented a query system that looks up the exchange rate of a specific currency against the Euro on a specific date. The range of

[4] https://www.ecb.europa.eu/stats/eurofxref/eurofxref-hist.zip, Wayback Machine URL: https://web.archive.org/www.ecb.europa.eu/stats/eurofxref/eurofxref-hist.zip

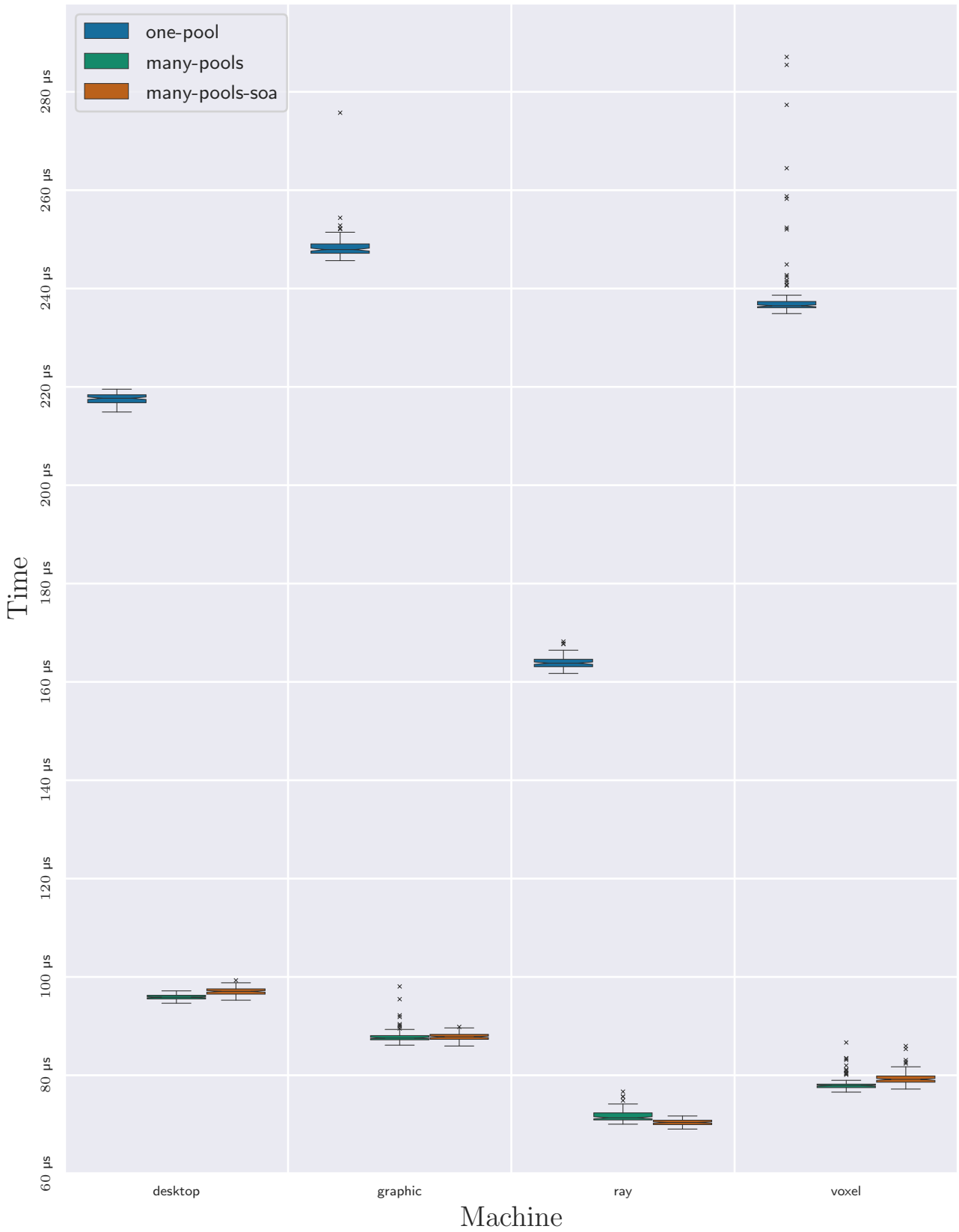


Figure 7.6: *Currency* results for *one AoS pool*, *one AoS and one Mixed pool*, and *one AoS and one SoA pool*, respectively. (Lower times are better)
NB: Y axis has been trimmed at the bottom to save space.

sates we are using spans from 1st April 1999 up to 8th May 2019. We assume the following:

- Most queries will refer to dates that are "recent"; we assume 80% of our queries will reference exchange rates since 2018-01-01 and the remaining 20% to reference older exchange rates.
- For the sake of simplicity, we only support the lookup of USD and GBP exchange rates, each at a 50% probability.

In our case study, we perform a number ($N = 5000$) of queries against three different implementations of this query system. The differences in these implementations are as follows:

- All exchange rates are placed in *One Pool* (in an AoS layout) or are partitioned into *Two Pools* ("recent" and the remaining dates).
- In the case of *Two Pools*, either an *SoA* layout is used for the "recent" dates or a *Mixed* layout. An AoS layout is used for the pool of not "recent" dates.

We present our results in Figure 7.6. We observe that, compared to the *Two pools, Mixed* approach (geometric mean for all executions of 82.929 µs), the *One Pool* approach (geometric mean for all executions of 214.553 µs) appears to be slower by a factor of 2.58x (with respect to the derived geometric means). The *Two pools, SoA* approach (geometric mean for all executions of 83.093 µs) is almost on par with the *Two pools, Mixed* approach with respect to their geometric means. This gives some credence (albeit weak) to Claim C4.

7.5 Traffic

We implement a case study [SSM18] that simulates road traffic according to the Nagel-Schreckenberg traffic model [NS92]. For our evaluation, we ported a version of that benchmark[5] into

[5] Hosted in https://github.com/prg-titech/dynasoar/blob/9dab3900c142aa8ee41966647bd97ef3d035768c/example/traffic/baseline_aos/traffic.cu

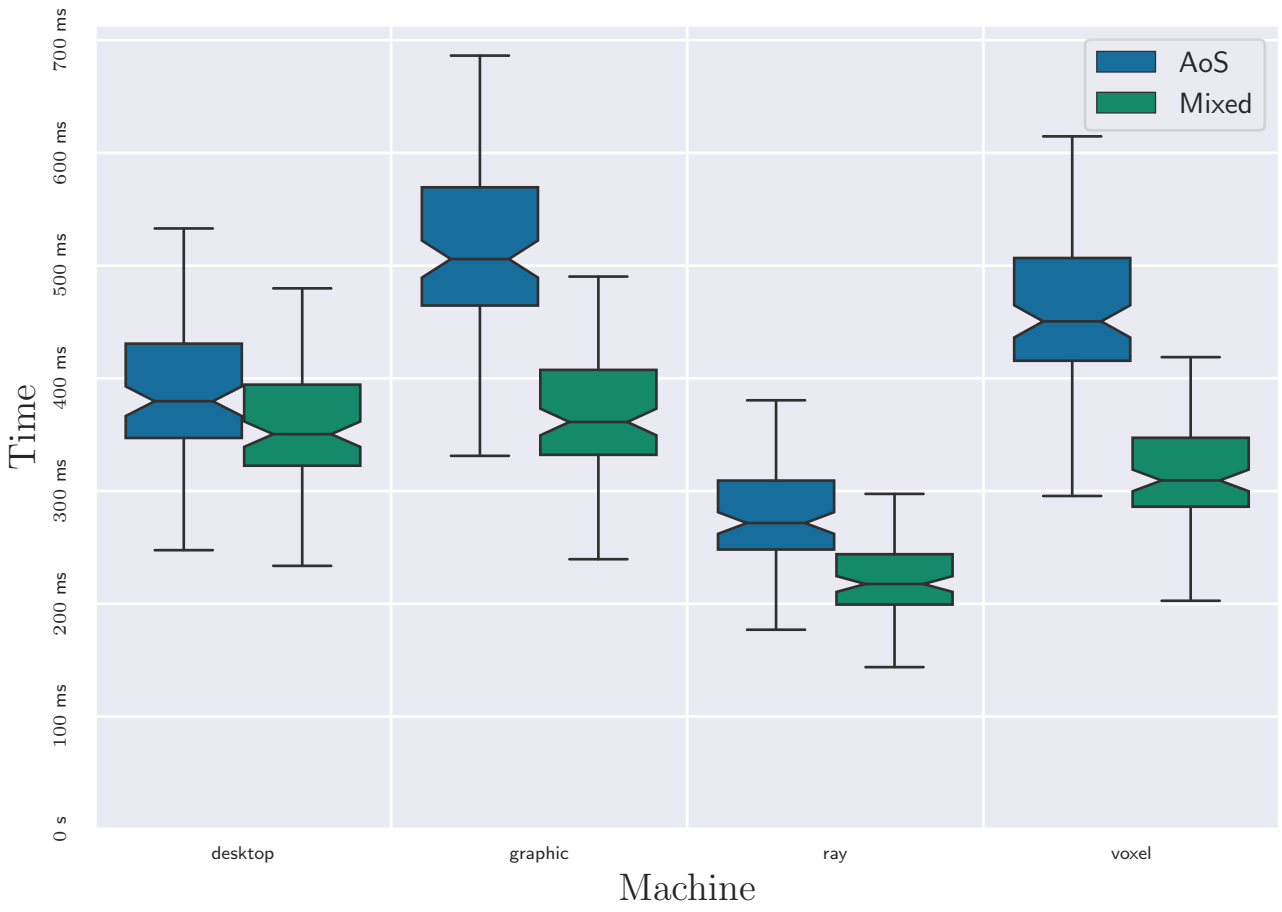


Figure 7.7: *Traffic* results for *AoS* and *Mixed* layouts, respectively. (Lower times are better)

vanilla C++. The simulation consists of iterating over a collection of *cells* and a collection of *traffic lights*.

Roads in this model are split into equally-sized *cells*; each cell contains up to one car. Unidirectional edges between cells represent traffic flow; because cells are equally-sized, edges are weightless. Cells also represent intersections; edges from and to an intersection cell represent how traffic from and to adjacent cells flows via this intersection. A street network is therefore represented as a graph of cells.

Cells have a *maximum velocity*; this is intended to represent speed limits. Moreover, cells adjacent to intersections are controlled by *traffic lights*; these dictate whose cell's traffic can pass through the intersection at any given time.

The original implementation constructs a random street network, places cars in random cells, then runs the simulation for 1000 iterations; we decided to follow this approach as well. Each

iteration consists of two steps:

- First, determine and store the path each car will take. The length of a path is capped by the car's *velocity*. A car's velocity is mutable and it only affects the path length for the current iteration. At an intersection, a random outgoing cell is chosen. If a currently being calculated path would pass through an occupied cell, the path will end on the current cell.
- Then, determine destination cells for all cars. The cars are then moved to their respective cells.

We implement an AoS and a Mixed variant. The Mixed variant is a best-effort SoA layout for two reasons:

- Each Cell contains a random number generator state (to *e.g.,* simulate random car speedups/slowdowns, *etc.*) which is effectively a black box, hence it is not transformed into SoA.
- Cells and traffic lights contain resizeable arrays of a maximum size (to *e.g.,* track incoming/outgoing cells, track the path the car (if any) on that cell is going to take). An implementation of an array with a size capped at compile time would consist of an array of a fixed size and a size field. Using such an implementation would mean that we would use an abstraction, hence we do not place the array size and contents in different clusters.

Results are presented in Figure 7.7. The geometric mean of all executions for the AoS and Mixed variants amounts to 392.741 ms and 304.332 ms, respectively. This corresponds to a ratio of 1.29x between these two geometric means. We attribute this difference in performance to less cache pollution (as in *OP2*).

Thus, being able to easily switch from an AoS to an Mixed layout if we were using SHAPES-z would easily result in an easy speedup. This supports Claims C1 and C3.

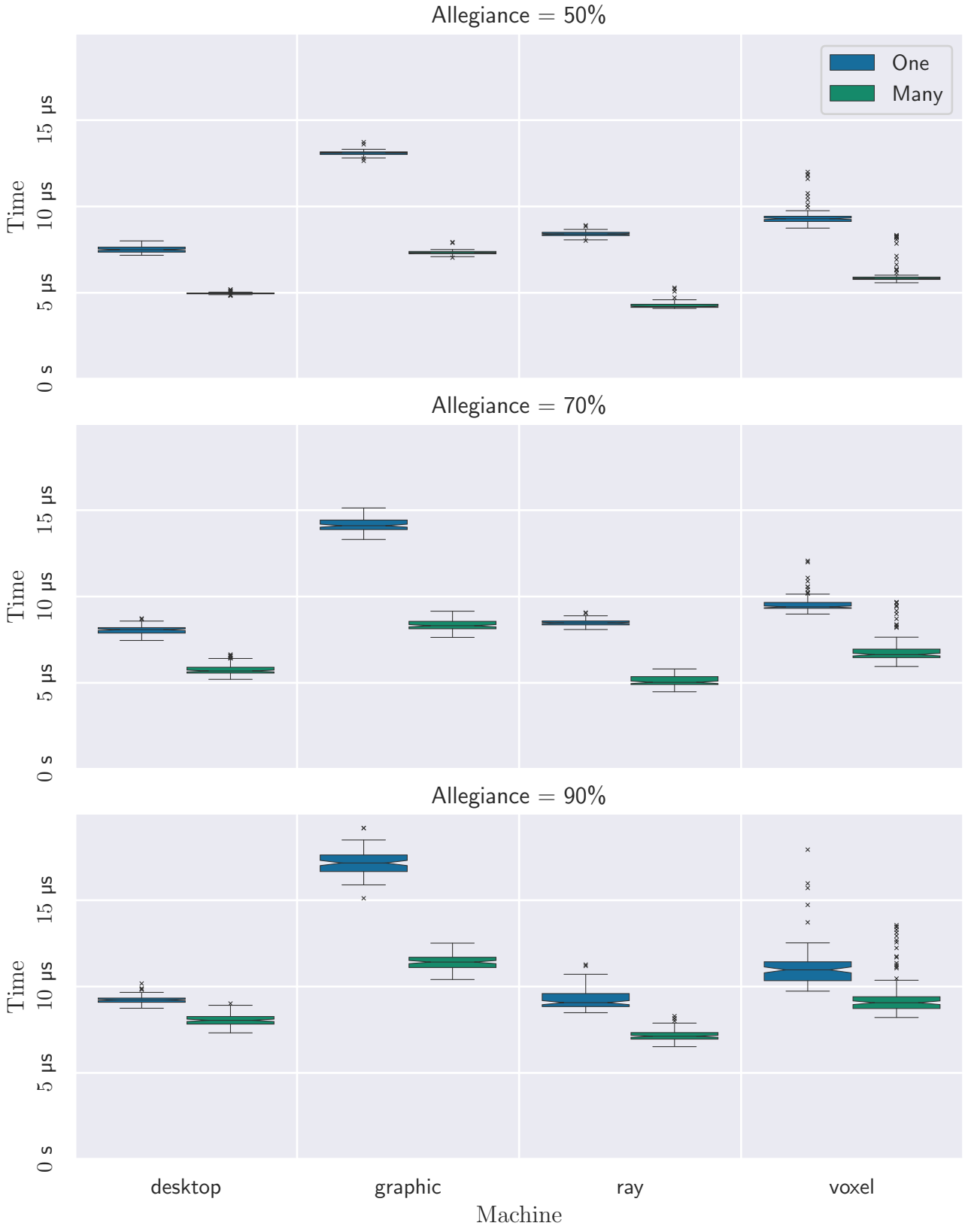


Figure 7.8: *Doors* results for *one* and *many* pools, respectively. (Lower times are better)

7.6 Doors

Consider a list of *characters* placed in a 2D space; each character belongs to one of two teams (Red team vs. Blue team). Now, consider a list of automatic doors in the same space; doors also have an allegiance and they can only open when a character of the same allegiance is in their proximity[6]. Given a list of characters and doors, our objective is to determine which doors have a character of the same allegiance within proximity of themselves, hence must be opened.

An obvious optimisation is to partition doors and characters into allegiance-specific pools (*i.e.,* one red, one blue character pool, and one red, one blue door pool), therefore allegiance checks can be eliminated (which is what we have done in our code).

To that extent, we compare the performance of checking 100 randomly generated doors and characters. We run our case study with 50%, 70%, and 90% of characters and doors belonging to the Red team. We assume an AoS layout in both cases. Figure 7.8 presents the relevant results. We observe the following geometric means for each category:

- 50%: 9.386 µs for *One pool*, 5.546 µs for *Many pools*
- 70%: 9.799 µs for *One pool*, 6.411 µs for *Many pools*
- 90%: 11.303 µs for *One pool*, 8.868 µs for *Many pools*

In aggregate, the geometric mean for the *One pool* approach is 10.131 µs, whereas for the *Many pools* approach, the geometric mean is 6.807 µs.

As such, we observe that the use of multiple pools achieves non-trivial speedups, even with 90% of characters and doors being red (which results in fewer eliminated checks). This supports Claim C4.

[6] Inspired from material from the following URL: https://web.archive.org/web/20190517194356/https://deplinenoise.files.wordpress.com/2015/03/gdc2015_afredriksson_simd.pdf

7.7 Evaluation of shapeszc

We will now evaluate shapeszc by comparing the compiled SHAPES-z code against its equivalent handwritten C++ counterpart. As stated in § 7.1, we have performed this comparison only for the *Doors* and *Currency* case studies; this is due to practical considerations (*i.e.,* time constraints) and does not imply that the remaining case studies cannot be implemented in SHAPES-z.

Our comparison will be twofold: Firstly, we will compare how SHAPES-z fares against handwritten C++ in terms of execution speed (*direct comparison*). Secondly, we will compare what sort of performance improvement/degradation we observe between different layouts in SHAPES-z and compare them to the respective improvement/degradation we observe between different layouts in handwritten C++ (*indirect comparison*).

Notice that with respect to our comparisons, there are three noteworthy caveats that need to be taken into consideration:

V1 shapeszc was developed by a single person within a timespan of approximately 7 months; mature compilers such as GCC and Clang have been in development for years and the amount of engineering time put into either of them is orders of magnitude bigger than that of shapeszc.

V2 We are making use of the public LLVM API to generate a "standard" `-O3` optimisation pipeline (§ 6.2.2); it is possible that Clang is making use of a different optimisation pipeline.

V3 Even though both shapeszc and Clang generate LLVM IR (which is then optimised by LLVM), considering that the LLVM and Clang projects are under the LLVM umbrella, it may be possible for LLVM to have been better tailored to better optimise LLVM IR generated by Clang or for Clang to emit LLVM IR that happens to be more amenable to optimisation by LLVM. While such information is most likely publicly available (*e.g.,* within posts in the LLVM mailing list), considering that LLVM is open source, performing

the appropriate research is bound to require additional time to be spent, especially to a person with little to no internal familiarity with projects under the LLVM umbrella.

We now present our measurements for the *Doors* and *Currency* case studies. Note that, as we stated in § 7.1, we are not running results on `voxel` machines.

Doors Figure 7.9 presents the C++ measurements against our shapeszc implementation for the *Doors* case study. Figure 7.10 aggregates the medians of all of these executions and compares the median C++ and SHAPES-z execution times for each layout, machine, and allegiance probability: A median that is located "above" the diagonal line indicates a faster SHAPES-z median execution time; a median located "below" the diagonal line indicates a faster C++ median execution time.

With respect to a *direct comparison*, when comparing the execution time of C++ and SHAPES-z, we observe that the geometric mean of all C++ executions is 8.220 µs, whereas the geometric mean of all SHAPES-z executions is 8.247 µs. While this may suggest an almost identical execution speed, Figure 7.10 indicates that there is, indeed, some variance in the results. For instance, SHAPES-z tends to perform better on the `graphic` machine, whereas C++ tends to perform better on the `desktop` machine.

With respect to an *indirect comparison*, when comparing the *One pool* and *Many pools* approaches in SHAPES-z, we obtain a geometric mean of 8.969 µs and 7.582 µs, respectively. While this is certainly a result that is consistent with our C++ results in § 7.6, one point of concern is that the geometric mean of the C++ variant of the *Many pools* approach is 6.807 µs.

We argue that our SHAPES-z results for *Doors* appear quite promising, and that with respect to *Doors* SHAPES-z appears to be on more-or-less equal footing with C++, especially when taking Caveats V1–V3 into account. We argue that this gives credence to Claim C2.

Currency Figure 7.11 presents the measurements of C++ against our shapeszc implementation for the *Currency* case study. Figure 7.12 aggregates the medians of all of these executions

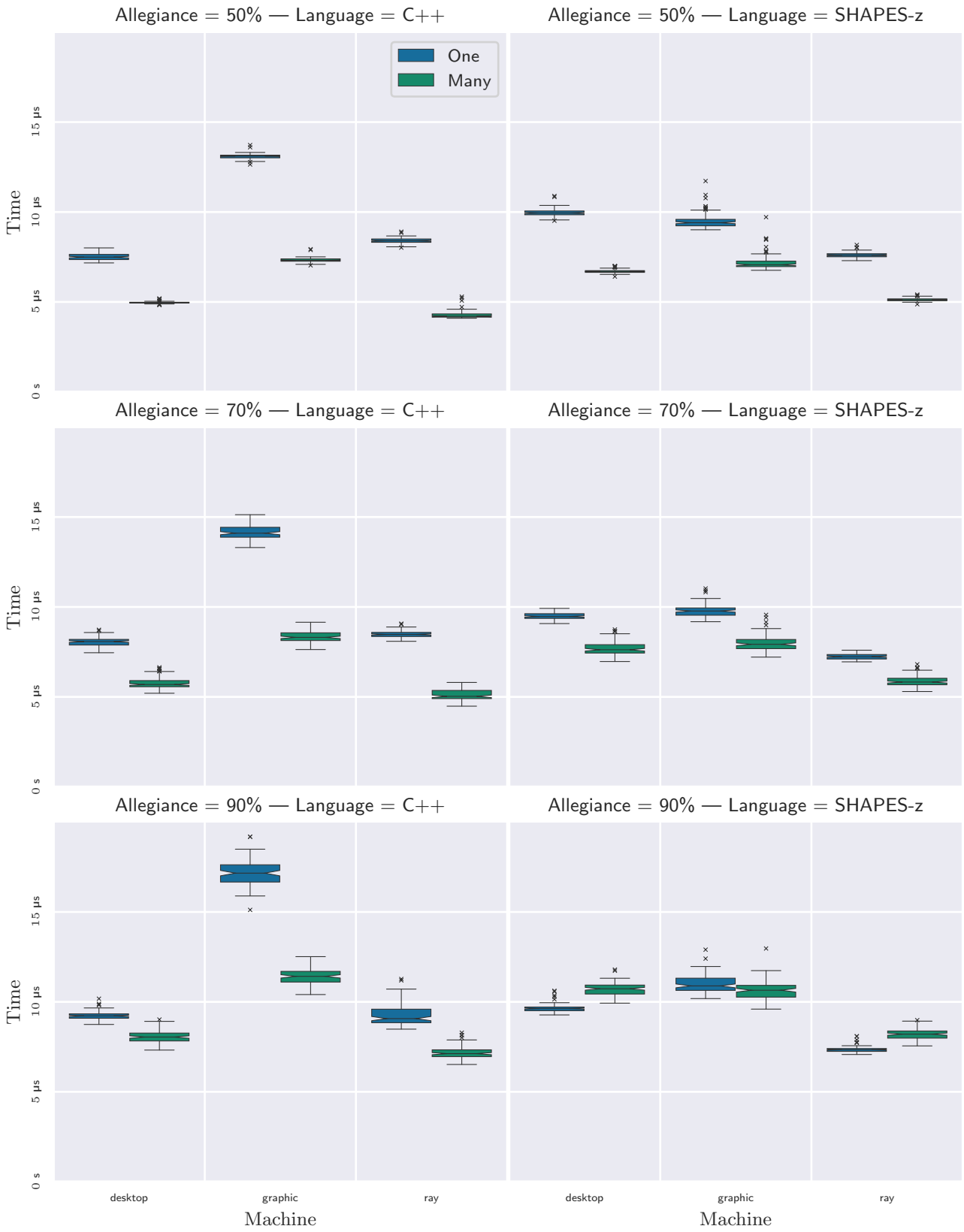


Figure 7.9: *Doors* results: C++ *vs.* SHAPES-z variant for *one* and *many* pools, respectively. (Lower times are better)

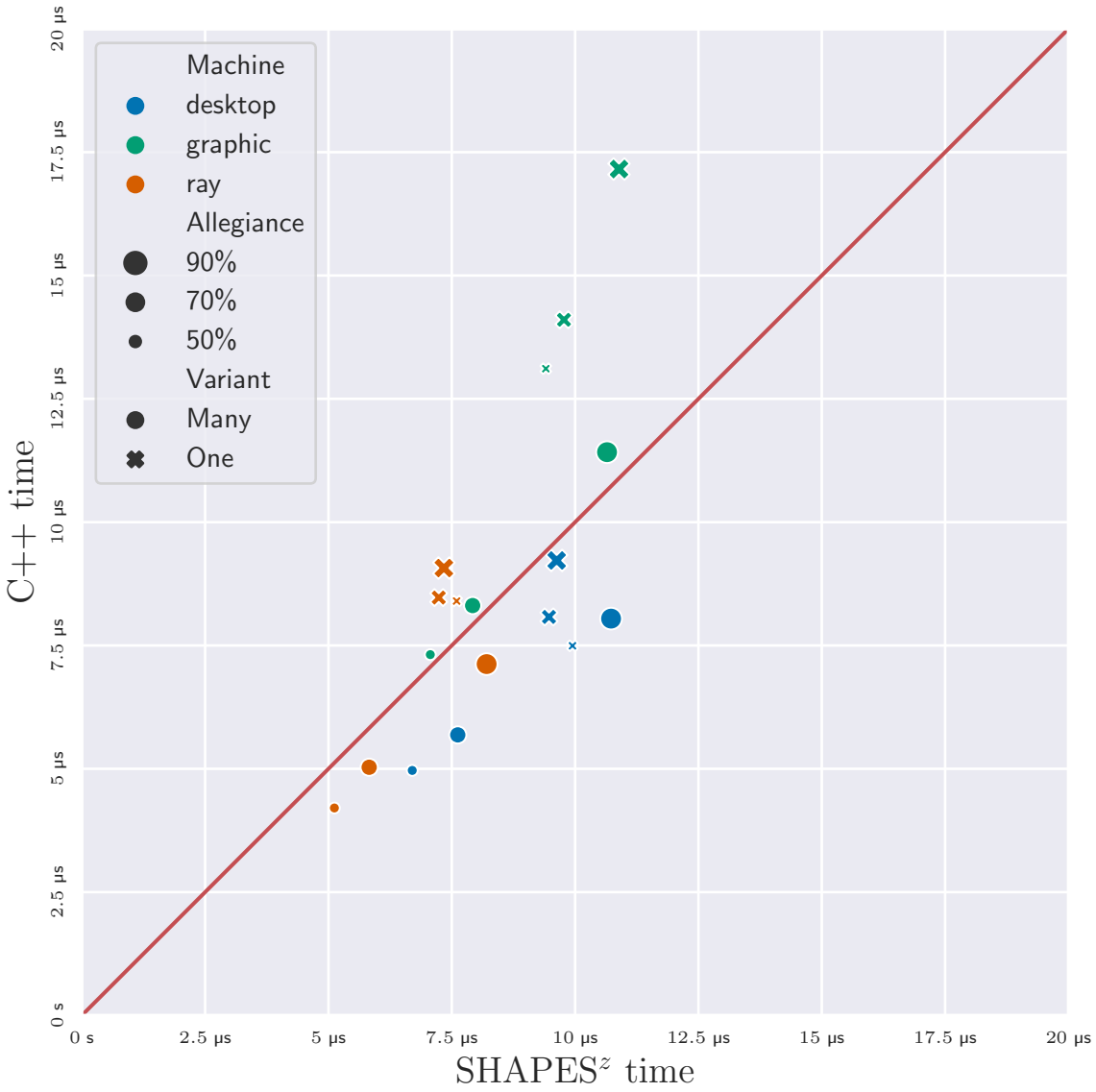


Figure 7.10: Comparison of medians of *Doors* results: C++ *vs.* SHAPES-z variant.

and compares the median C++ and SHAPES-z execution times for each layout and machine: similar to Figure 7.10, a median located "above" the diagonal line indicates a faster SHAPES-z median execution time; a median located "below" the diagonal line indicates a faster C++ median execution time.

With respect to a *direct comparison*, when comparing the execution time of C++ and SHAPES-z, we observe that the geometric mean of all C++ executions is 113.920 µs, whereas the geometric mean of all SHAPES-z executions is 278.204 µs. Indeed, Figure 7.12 indicates that the median execution of C++ is much faster than its median SHAPES-z counterpart. This indicates a possible optimisation issue in our compiler, which is certainly alarming.

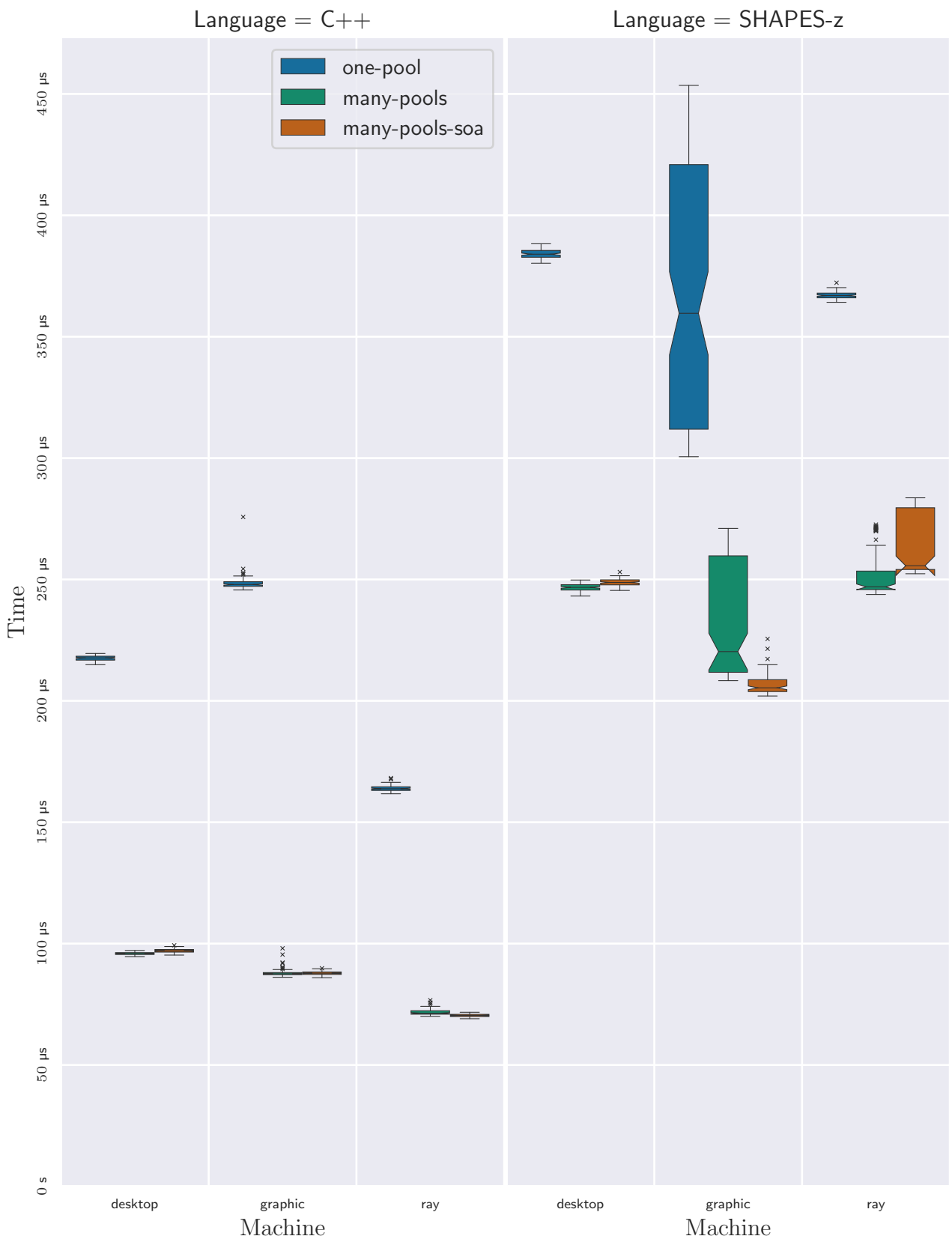


Figure 7.11: *Currency* results: C++ *vs.* SHAPES-z variant for *one AoS pool*, *one AoS and one Mixed pool*, and *one AoS and one SoA pool*, respectively. (Lower times are better)

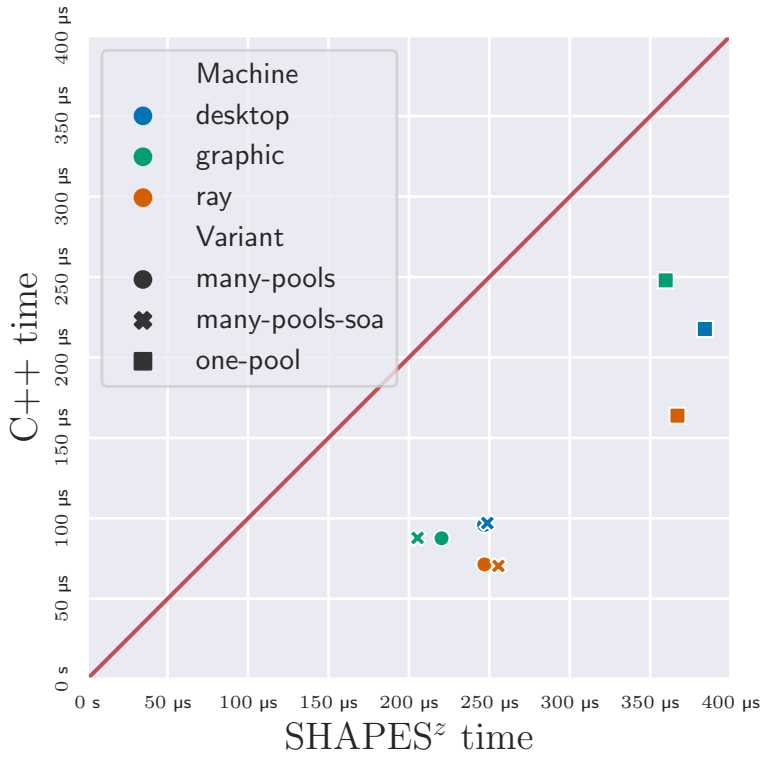


Figure 7.12: Comparison of medians of *Currency* results: C++ *vs.* SHAPES-z variant.

With respect to an *indirect comparison*, when comparing the *One pool*, *Many pools, Mixed* and *Many pools, SoA* approaches in SHAPES-z, we obtain a geometric mean of 371.426 µs, 243.095 µs and 238.474 µs, respectively. While this is certainly a result that is consistent with our C++ results in § 7.6, one point of concern is that, with respect to the geometric mean, the ratio between the *Many pools, SoA* and *One pool* approaches is approximately 1.56x, whereas in § 7.6, we observed this ratio to be approximately 2.58x.

We argue that our SHAPES-z results for *Doors* appear quite promising, and that with respect to *Doors* SHAPES-z appears to be on more-or-less equal footing with C++, especially when taking Caveats V1–V3 into account. We argue that this gives credence to Claim C2.

Attempting to close the direct comparison gap With respect to the direct comparison in the *Currency* case study, we tried a variety of approaches so as to close the gap in execution time. More specifically:

- We compiled our C++ code with Clang and compared our SHAPES-z code against it (since we were using GCC). With respect to *Doors*, the SHAPES-z version was virtually

on an equal footing compared to the C++ version compiled with Clang; with respect to *Currency*, the slowdown of the SHAPES-z version was more-or-less similar regardless of the compiler choice for the C++ version.

- We tried using different LLVM versions compared to LLVM 8, which was the version we initially used (specifically, LLVM versions 10–13). This did not seem to be of great significance.

- We peeked at the machine code generated for our SHAPES-z and compared it against the one generated for our C++ code in a purely empirical manner. The most striking difference we observed was that, in the case of `shapeszc`, we observed *loop alignment* [FHM99] performed by LLVM. However, we could not determine how to disable this with respect to code generation.

- We tried using `llvm-opt` for optimisations; `llvm-opt` is a standalone LLVM IR optimiser provided by LLVM. While we did observe cases where some optimisation opportunities were picked by `llvm-opt`, compared to LLVM's `PassManagerBuilder` pipeline, this improvement did not turn out to be significant. Moreover, `llvm-opt` did not seem to perform loop alignment during code generation; given that we did not observe any significant difference, it seems unlikely that loop alignment would have been a culprit.

- We tinkered with LLVM's `PassManagerBuilder` API in the hopes of finding an additional switch that would allow us to eke out some additional optimisability by LLVM, to no avail. We considered explicitly fiddling with what optimisation passes were being run and in what order, but due to our lack of expertise with optimisation pass order and lack of knowledge with the LLVM internals, we abandoned this idea.

- The *Currency* case study uses binary search, by invoking `std::lower_bound()` in the C++ version; we tried modifying our SHAPES-z version to behave exactly like the GNU C++ STL version of `std::lower_bound()`, as well as replace `std::lower_bound()` with an equivalent C++ version, but to no avail.

- Since our C++ code makes heavy use of `std::vector`, we peeked into the internals of the

GNU C++ STL to determine if there was anything noteworthy. We did observe that, unlike our pools (which store the pool size and pointers to clusters), that particular version of `std::vector` used pointers to indicate the beginning and end of elements (`_M_start` and `_M_finish`, respectively). While certainly a possible approach for micro-optimisations, this difference is definitely not enough to close the gap in the direct comparison.

As such, we are suspecting that Caveats V1–V3 seem to be the most significant culprits.

7.8 Conclusions

We have taken examples from a range of applications; we claim that SHAPES makes our code more readable compared to the version where we perform these optimisations by hand (Claim C2). When the developer is uncertain of what data structure to use, SHAPES makes it easier to experiment with several and pick the most performant one for the use case at hand (Claim C1). Finally, we have shown that layout/pool consideration, *i.e.,* use of Mixed layouts (Claim C3) and of multiple pools (Claim C4) can affect performance significantly.

Additionally, when convenience and ease of use is the primary concern, `shapeszc` does show that it has potential to be on par with C++ in terms of performance. We anticipate that a future `shapeszc` version has the potential to be within a close range in terms of performance against handwritten C++ code (*i.e.,* C++).

As such, we believe that incorporating the concepts of pooling and clustering as proposed by SHAPES into existing and/or new languages is worth considering. Moreover, we hope that a future version of `shapeszc` will be able to perform optimisations in a manner that is close to the optimisation capabilities of current compilers.

Chapter 8

Conclusions

We have presented SHAPES, a language extension that uses a type-based approach to integrate memory optimisation in managed languages to enable greater control of the memory layout (hence potentially improving cache utilisation), whilst keeping the business logic layout-oblivious. It relies on types both to document and enforce aspects of data locality and to protect object abstraction and combat high-level memory safety bugs which may arise when manually deconstructing objects in structure-of-arrays transformations.

8.1 Achievements

We presented SHAPES as a means of allowing developers to utilise CPU caches in a high-level manner, while also staying close to the spirit of object-oriented programming. We initially dove into the design and architecture of CPU cache hierarchies (§ 1), so as to make better informed decisions with respect to how to best design SHAPES. We then presented the core ideas of SHAPES (*pooling* and *clustering*) and introduced its design through gradually more extended and refined stages, whilst also presenting our rationale for our design decisions (§ 2), which also demonstrates that our design was not done in an ad hoc manner. We also presented our aims with respect to ensuring memory safety in SHAPES.

In § 3, we presented existing work with respect to pooling and clustering and compared it with

the design of SHAPES; we concluded that SHAPES does, indeed, bring additional benefits into the table, hence it is worth pursuing. We also took note of possible future extensions that SHAPES could benefit from.

In § 4, we formalised SHAPES via SHAPES^h, a formal language; we presented the SHAPES^h operational semantics and type system, as well as formulated the concept of SHAPES^h well-formedness, which we then proved. In § 5, we presented SHAPES$_\ell$, a low-level language that can be easily implemented as a runtime, presented how SHAPES^h can be compiled into SHAPES$_\ell$ and proved that this compilation is correct and complete. We also argued why we expect the translation from SHAPES^h to SHAPES$_\ell$ to be performant during program execution. (*e.g.,* our use of specialisation means that an ahead-of-time approach to compilation can be used).

In § 6, we introduced SHAPES-z, our implementation of SHAPES as an embedded domain-specific language, as well as the design of `shapeszc`, our SHAPES-z compiler. We also presented the design of the `shapeszc` frontend (syntactic and semantic analysis), as well as the backend (LLVM code generation). In § 7, we introduced our case studies through which we evaluated the claims we made by SHAPES: We concluded that the benefits of SHAPES we claim do indeed have merit.

8.2 Future work

In § 3, when comparing SHAPES with existing work, we indicated some points for possible future extensions, such as additional layouts. We now reflect on how SHAPES (and SHAPES-z) can be extended to implement these extensions and also present a few potential extensions:

Support for SIMD In § A, we introduce SHAPES$^{\text{SIMD}}$, our future extension of SHAPES, which we expect to allow developers to also reap the benefits of SIMD in a high-level and easy-to-use manner.

Additional layouts A common pattern among the work we examined in § 3 was the support for a hybrid layout, called *Arrays-of-Structs-of-Arrays* (AoSoA). Support for AoSoA is certainly worth considering, since only one pool cluster needs to be allocated; furthermore, our observation that *mixed* layouts are worth considering (§ 7) and our discussion of SIMD instruction sets (§ A) strengthens our argument.

We expect that AoSoA layouts can be added to SHAPES in a straightforward manner, by mainly extending the layout declaration syntax. One consideration that should also be taken into account is whether or not we should allow the number N of elements per chunk to also be variable to an extent, so as to *e.g.,* simultaneously accommodate machines with differing SIMD register width (*e.g.,* 128-bit SSE and 256-bit AVX [Int11]).

Automatic layout selection With respect to automatic layout selections, we consider the addition of an optional `advisory` keyword in pool declarations, which would indicate that the compiler is free to select a different layout; we envision that automatic layout selection can be determined via *e.g.,* machine learning or profile-guided optimisation (*e.g.,* data collected regarding cache misses via the `perf` tool in Linux [Gre20]).

Parallelism and concurrency DynaSOAr [SM19] and OP2 [GMS+11] each provide features that aid the developer with respect to concurrency: DynaSOAr allows concurrent object allocation within pools; OP2 generates execution plans wherein objects are logically partitioned in a manner that prevents an object's field from being potentially modified by two concurrent threads.

As discussed in § 5.4, in order to support concurrency in SHAPES, we would need to ensure that pool resizing is synchronised with respect to other operations on the pool, such as field access and object construction. One possible approach would be to introduce a per-pool lock that synchronises all operations within that specific pool. This approach, while certainly easy to implement, is suboptimal for applications that primarily access object fields.

We can eliminate the necessity for locking by introducing *pool borrowing*, akin to the concept

of borrowing present in Rust [KN18]. More specifically, a pool is owned by exactly one thread, but borrows of this specific pool can be created and passed to other threads. While the pool is bing borrowed, object allocation within the pool is disallowed; only when all borrows of a pool cease can object construction within the pool be performed.

An alternative approach would be to change the in-memory representation of pools, so as to make concurrent allocation and pool resizing easier to implement; we leave such a pool representation for a future SHAPES or SHAPES-z implementation.

Additional performance considerations SHAPES is also capable of accommodating additional features concerning performance: A layout can be extended to accommodate additional features, such as padding (to address false sharing [TLH94]), alignment, and placement of auxiliary fields (*e.g., mark word* and *klass pointer* on the HotSpot VM [hot06]).

The layout syntax can be also extended to allow the developer to constrain the index of an object in a specific pool to a range smaller than the machine word size (*e.g.,* 16-bit or 32-bit indices on 64-bit machines) so as to further improve on memory usage and cache utilisation. This can be achieved thanks to the use of specialisation and it can be syntactically implemented by *e.g.,* adding annotations to the respective layout.

Another possible consideration with respect to performance would be to provide a means of limiting the number of specialisations generated, so as to allow the developer to mitigate the justified concern of possible code bloat. One approach to tackling this would be to explicitly indicate which specialisations are allowed to be generated. This is similar to the concept of `extern template` introduced in C++11 [ISO12].

Additional garbage collection considerations In § 5.3.1, we stated our cautious optimism about the possibility of creating a pool-aware garbage collector. A further logical step would be to make a preliminary judgement on how easy would the addition of support for SHAPES_ℓ to existing GCs be. This is certainly a very ambitious goal, especially when taking into account the potential amount of engineering effort required even conceptually. However, we argue that

considering our objective to have SHAPES support managed languages, it is natural to demand interoperability with existing garbage collectors and consider whether or not this is worth pursuing.

Our design does not specify how the initial capacity of a pool will be picked. As possible options, we are currently envisaging either an implementation-defined default, a user-specified initial capacity (*e.g.,* via annotations) or a capacity derived from profile-guided optimisation.

In § 5.3.1, we presented how SHAPES can be integrated into a garbage collector. A possible extension on SHAPES GC would be to provide a custom API for reordering objects in a pool. This would, for instance, allow the nodes of a tree to be reordered sequentially in memory in the order that an algorithm traverses the tree.

8.3 Reflections

We started developing SHAPES with the intent to allow developers to better utilise the cache. However, our approach was certainly not the most conventional: Rather that starting implementing various constructs as prototypes and then build a type system on top of them, we initially began developing a formal model, which we then revised frequently. Such an approach is certainly not risk-free: If we failed to develop a correct and complete model, we would have to make use of models with glaring issues and/or weaknesses. The upside of this approach, however, is that once we have a correct and complete formal model, we do not need to resort to solutions that "work for 90% of the cases".

Moreover, we also developed a fully-fledged SHAPES compiler; this decision was also a risky one: Developing a compiler is certainly a non-trivial and time-consuming task, especially as a single-developer project. This means that should the compiler approach fail or pan out for any reason, the time spent on it will have been completely wasted. Thankfully, this turned out to not be the case and we are quite satisfied with our end results.

Bibliography

[BAD+09] 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. A type and effect system for deterministic parallel Java. In *OOPSLA*, pages 97–116, 2009.

[Boa18] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[Boy01] John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[CDBM15] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12. ACM, 2015.

[CDD+10] Pete Cooper, Uwe Dolinsky, Alastair F Donaldson, Andrew Richards, Colin Riley, and George Russell. Offload–automating code migration to heterogeneous multicore systems. In *High Performance Embedded Architectures and Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings 5*, pages 337–352. Springer, 2010.

[CKJA98] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *ASPLOS VIII*, pages 139–149. ACM, 1998.

[Cor15] Microsoft Corporation. *Direct3D 11.3 Functional Specification*. 2015. URL: https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm.

[Cor22] Microsoft Corporation. *The Typescript Handbook*. September 2022. Commit hash: c3f0d8. URL: https://www.typescriptlang.org/assets/typescript-handbook.pdf.

[CÖSW13] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. *Ownership Types: A Survey*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_3.

[DH99] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In Rachid Guerraoui, editor, *ECOOP' 99 — Object-Oriented Programming*, pages 92–115, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 106–117, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/277650.277670.

[DS21] Charles Donnely and Richard Stallman. Bison manual. https://www.gnu.org/software/bison/manual/bison.pdf, 2021. Wayback machine URL: https://web.archive.org/web/20210506070747/http://www.gnu.org/software/bison/manual/bison.pdf.

[FD15] Juliana Franco and Sophia Drossopoulou. Behavioural types for non-uniform memory accesses. *PLACES 2015*, page 39, 2015.

[FHM99] A. Fraboulet, G. Huard, and A. Mignotte. Loop alignment for memory accesses optimization. In *Proceedings 12th International Symposium on System Synthesis*, pages 71–77, 1999. doi:10.1109/ISSS.1999.814263.

[FHW+17] Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. You can have it all: Abstraction and good cache performance. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 148–167, New York, NY, USA, 2017. ACM. doi:10.1145/3133850.3133861.

[Fog12] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, pages 02–29, 2012.

[for15] Message Passing Interface forum. *MPI: A Message-Passing Interface Standard–Version 3.1*. 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[fS18] International Organization for Standardization. ISO/IEC 9899:2018: Information technology—Programming Languages—C, 2018.

[FTD+18] Juliana Franco, Alexandros Tasos, Sophia Drossopoulou, Tobias Wrigstad, and Susan Eisenbach. Safely abstracting memory layouts. FTfJP, Formal Techniques for Java-like Programs, 2018.

[FWD16] Juliana Franco, Tobias Wrigstad, and Sophia Drossopoulou. Towards enabling low-level memory optimisations at the high-level with ownership-like annotations. IWACO, International Workshop on Aliasing, Capabilities and Ownership, 2016.

[FWF+90] David W Forslund, Charles Wingate, Peter Ford, J Stephen Junkins, Jeffrey Jackson, and Stephen C Pope. Experiences in writing a distributed particle simulation code in C++. In *C++ Conference*, pages 177–190, 1990.

[GJS+14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 8 edition (Java series), 2014.

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM Sigplan Notices*, volume 37, pages 282–293. ACM, 2002.

[GMS+11] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H.J. Kelly. Performance analysis of the OP2 framework on many-core architectures. *SIGMETRICS Perform. Eval. Rev.*, 38(4):9–15, March 2011. doi:10.1145/1964218.1964221.

[Gre20] Brendan Gregg. perf. http://www.brendangregg.com/perf.html, Wayback Machine URL: https://web.archive.org/web/20200207065027/http://www.brendangregg.com/perf.html, 2009–2020.

[HBM+04] Xianglong Huang, Stephen M Blackburn, Kathryn S Mckinley, J Eliot, B Moss, Zhenlin Wang, and Perry Cheng. The Garbage Collection Advantage: Improving Program Locality. In *OOPSLA*, 2004.

[Hen05] David Henry. MD5Mesh and MD5Anim files formats. http://tfc.duke.free.fr/coding/md5-specs-en.html, Wayback Machine: https://web.archive.org/web/20180816101227/http://tfc.duke.free.fr/coding/md5-specs-en.html, 2005.

[HL90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298, 1990. doi:10.1145/97879.97911.

[HL18] Holger Homann and Francois Laenen. SoAx: A generic C++ structure of arrays for handling particles in hpc codes. *Computer Physics Communications*, 224:325–332, 2018.

[Hof05] H.P. Hofstee. Power efficient processor architecture and the cell processor. In *11th International Symposium on High-Performance Computer Architecture*, pages 258–262, 2005. doi:10.1109/HPCA.2005.26.

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285, 1991.

[hot06] HotSpot glossary of terms. https://web.archive.org/web/20190927112007/http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html, 2006.

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[IdFC07] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1, 2007.

[Int11] Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual. 2011.

[IPW01] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

[ISO12] ISO ISO. IEC 14882: 2011 Information technology—Programming languages—C++. *International Organization for Standardization, Geneva, Switzerland*, 27:59, 2012.

[ita] *Itanium C++ ABI*, 1999–. URL: https://itanium-cxx-abi.github.io/cxx-abi/abi.html.

[JK17] Nouraldin Jaber and Milind Kulkarni. Data structure-aware heap partitioning. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 109–119, New York, NY, USA, 2017. ACM. doi:10.1145/3033019.3033030.

[KBR14] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL Shading Language–Language Version: 4.40*. 2014. URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.40.pdf.

[KJQ08] Jeff Keasler, Terry Jones, and Dan Quinlan. TALC: A simple C language extension for improved performance and code maintainability. 05 2008.

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[Kor15] Simon Kornblith. Julia structs of arrays. https://github.com/simonster/structsofarrays.jl/blob/v0.0.3/src/StructsOfArrays.jl, 2015.

[LA00] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *Acm Sigplan Notices*, 35(5):145–156, 2000.

[LA03] Chris Lattner and Vikram Adve. Data structure analysis: A fast and scalable context-sensitive heap analysis. Technical report, U. of Illinois, 2003.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[LA05] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *PLDI '05*, pages 129–142. ACM, 2005.

[LHH14] Roland Leißa, Immanuel Haffner, and Sebastian Hack. Sierra: a SIMD extension for C++. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pages 17–24. ACM, 2014.

[Lim] ARM Holdings Limited. *ARM®Architecture Reference Manual*, g.a edition. URL: https://documentation-service.arm.com/static/60119835773bb020e3de6fee.

[MHR+15] Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. Columnar objects: Improving the performance of analytical applications. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 197–210. ACM, 2015.

[NC22] NVIDIA®Corporation. *CUDA C++ Programming Guide*. 2022. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[NS92] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *Journal de physique I*, 2(12):2221–2229, 1992.

[NSL+11] Chris J Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, et al. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 224–235. IEEE, 2011. doi:10.1109/CGO.2011.5764690.

[O+89] John K Ousterhout et al. *Tcl: An embeddable command language*. Citeseer, 1989.

[Pav14] Igor Pavlov. 7-Zip LZMA benchmark (Intel Haswell), 2014. Wayback machine URL: http://web.archive.org/web/20220901062607/https://www.7-cpu.com/cpu/Haswell.html. URL: https://www.7-cpu.com/cpu/Haswell.html.

[PEM16] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with Flex. https://westes.github.io/flex/manual/, 2016. Wayback Machine URL: https://web.archive.org/web/20220807072603/https://westes.github.io/flex/manual/.

[PHCP03] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 172–184, New York, NY, USA, 2003. ACM. doi:10.1145/604131.604147.

[PM12] Matt Pharr and William R Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

[PNCB04] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)*, 2004.

[PPSdM01] Preeti Ranjan Panda, Preeti Ranjan Panda, Luc Semeria, and Giovanni de Micheli. Cache-efficient memory layout of aggregate data structures. In *Proceedings of the*

14th International Symposium on Systems Synthesis, ISSS '01, pages 101–106, New York, NY, USA, 2001. ACM. URL: http://doi.acm.org/10.1145/500001.500026, doi:10.1145/500001.500026.

[Pro21] LLVM Project. LLVM language reference manual. https://llvm.org/docs/LangRef.html, 2003–2021. Wayback Machine URL: https://web.archive.org/web/20210420030314/https://llvm.org/docs/LangRef.html.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/73560.73562.

[SBP+12] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. *Specification, IBM, janvier*, 2012.

[SKK+13] Kamal Sharma, Ian Karlin, Jeff Keasler, J McGraw, and Vivek Sarkar. User-specified and automatic data layout selection for portable performance. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[SKK+15] Kamal Sharma, Ian Karlin, Jeff Keasler, James McGraw, and Vivek Sarkar. Data layout optimization for portable performance. volume 9233, pages 250–262, 08 2015. doi:10.1007/978-3-662-48096-0_20.

[SM18] Matthias Springer and Hidehiko Masuhara. Ikra-Cpp: A C++/CUDA dsl for object-oriented programming with structure-of-arrays layout. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, page 6. ACM, 2018.

[SM19] Matthias Springer and Hidehiko Masuhara. DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in In-*

formatics (LIPIcs), pages 17:1–17:37, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10809, doi:10.4230/LIPIcs.ECOOP.2019.17.

[SSM18] Matthias Springer, Yaozhu Sun, and Hidehiko Masuhara. Inner array inlining for structure of arrays layout. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2018, pages 50–58, New York, NY, USA, 2018. ACM. doi:10.1145/3219753.3219760.

[Str11] Robert Strzodka. Abstraction for AoS and SoA layout in C++. In *GPU computing gems – Jade edition*, pages 429–441. Elsevier, 2011.

[TFD+20] Alexandros Tasos, Juliana Franco, Sophia Drossopoulou, Tobias Wrigstad, and Susan Eisenbach. Reshape your layouts, not your programs: A safe language extension for better cache locality. *Science of Computer Programming*, 197:102481, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0167642320300915, doi:https://doi.org/10.1016/j.scico.2020.102481.

[TFW+18] Alexandros Tasos, Juliana Franco, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. Extending SHAPES for SIMD architectures: An approach to native support for struct of arrays in languages. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pages 23–29, 2018.

[TLH94] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[UBSO15] Vlad Ureche, Aggelos Biboudis, Yannis Smaragdakis, and Martin Odersky. Automating ad hoc data representation transformations. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 801–820, New York, NY, USA, 2015. ACM. doi:10.1145/2814270.2814271.

[Whe04] David Wheeler. SLOCCount. https://dwheeler.com/sloccount/, Wayback Machine URL: https://web.archive.org/web/20190621211304/https://dwheeler.com/sloccount/, 2001–2004.

[WPM+09] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for Java. In *ECOOP 2009*, LNCS, pages 445–469. Springer, 2009. doi:10.1007/978-3-642-03013-0_21.

Appendices

Appendix A

Future work: Entering the SIMD territory

Modern CPU architectures have introduced *SIMD* instructions (*Single Instruction, Multiple Data*) in their instruction sets [Int11, Lim]. The premise of SIMD is twofold: SIMD introduces registers that can fit multiple units of data simultaneously (*e.g.,* k floating-point numbers). Additionally, SIMD introduces operations that can be simultaneously applied to such multiple units of data. That is, on a CPU where a SIMD register can hold k elements, a SIMD addition instruction will perform k additions simultaneously and produce k sums. This implies a speedup of up to k with respect to single-core performance, which makes SIMD an attractive approach in terms of maximising performance.

Indeed, before fully carving out our case studies (§ 7), we implemented handwritten versions of the OP2 `airfoil` application (§ 7.2), albeit running only on a single thread, and measured their performance *in a preliminary fashion*; these consisted of the implementations presented in Figure 7.3 (*AoS*, *Mixed* and *SoA* layouts), as well as an *SoA version that used SIMD wherever possible*. We ran this preliminary test on the machines of Figure 7.1.

Figure A.1 presents our results, which show our SIMD version outperforming all other implementations. While the speedup we obtain with SIMD in an SoA layout is certainly less than

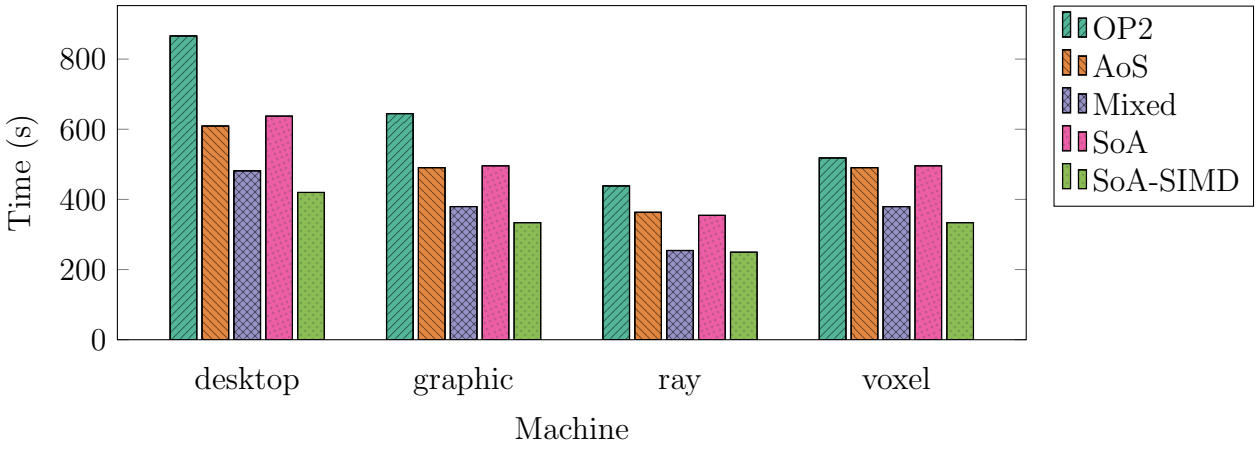


Figure A.1: OP2 `airfoil` single-threaded execution for the *original OP2* version, with our handwritten *AoS*, *Mixed*, and *SoA* versions, as well as an *SoA version that uses SIMD*, lower times are better.

2x[1], it is certainly a direction for future SHAPES extensions that deserves consideration.

However, even with that certainly enticing potential for a performance speedup, it appears that SIMD is underutilised by developers. We argue that this can be attributed to an extent to the following challenges:

Portability concerns Instruction set architectures can vary greatly with respect to their implementation of SIMD.

Mismatch between the scalar and SIMD paradigms We argue that a scalar language (*e.g.,* C) is not necessarily well-fitted to handle SIMD in what we consider to be an intuitive manner.

SIMD requires data layouts that are at odds with OOP As we will see in § A.2, compared to an Array-of-Structs (AoS) layout, a Struct-of-Arrays (SoA) data layout (*cf.,* § 2.1) makes it easier for the developer to incorporate SIMD into their business logic.

To tackle these challenges, we will present SHAPES$^{\text{SIMD}}$ in this chapter. SHAPES$^{\text{SIMD}}$ is our vision of extending SHAPES with concepts and semantics that make taking advantage of SIMD much easier compared to current practices.

[1] A further investigation showed us that `airfoil` spends a major portion of its execution on a loop that is excruciatingly hard to parallelise with SIMD manually, hence we did not attempt it.

```
#include <xmmintrin.h>
void square(float* arr, size_t n) {
    for (size_t i = 0; i < n; i += 4) {
        __m128 v = _mm_loadu_ps(&arr[i]);
        __m128 sq = _mm_mul_ps(v, v);
        _mm_storeu_ps(&arr[i], sq);
    }
}
```

Listing A.1: SSE example: Vector scaling by a constant

§ A.1 presents the tradeoffs of the current mainstream approaches to SIMD. § A.2 presents the challenges a developer faces with regards to memory layouts when incorporating SIMD. In § A.3 and § A.4, we introduce the ISPC and Sierra languages, respectively; in § A.5 reason why their designs are worth considering with respect to the design of SHAPES$^{\text{SIMD}}$.

A.1 Current mainstream approaches to SIMD

We now present an overview to SIMD through the SSE instruction set for the Intel x86 and x86-64 architectures [Int11].

SSE introduces *vector registers* to the x86 and x86-64 architectures, which are dubbed as `xmm` registers. SSE vector registers are 128-bit wide and can simultaneously hold, *e.g.,* two 64-bit integers, or four 32-bit floating-point numbers, *etc.* SSE also introduces SIMD instructions that operate on these vector registers; instruction `addps`, for instance, performs four simultaneous 32-bit floating-point additions and generates four 32-bit floating-point sums.

Use of SSE in languages such as C and C++ is usually done directly through *intrinsic functions* or indirectly through *autovectorisation* as part of optimisation.

Intrinsics In the case of intrinsics, the compiler provides a set of intrinsic functions that expose the instruction set to the language. During assembly code generation, calls to these intrinsic functions will be explicitly converted to the invocation of the specific CPU instruction they correspond to.

```
1  float sum_gt10(float* arr, int len)
2  {
3      float sum = 0;
4      for (int i = 0; i < len; i++) {
5          if (arr[i] > 10) {
6              sum += arr[i];
7          }
8      }
9      return sum;
10 }
```

Listing A.2: Autovectorisation example: Conditional sum

```
1  .LBB0_4:
2      movss xmm3, dword ptr [rdi + 4*rcx]
3      movaps xmm0, xmm1
4      cmpltss xmm0, xmm3
5      addss xmm3, xmm2
6      andps xmm3, xmm0
7      andnps xmm0, xmm2
8      orps xmm0, xmm3
9      add rcx, 1
10     movaps xmm2, xmm0
11     cmp rax, rcx
12     jne .LBB0_4
13
14
```

Listing A.3: Clang 7.0.1 output (main loop body)

```
1  .LBB0_7:
2      movups xmm2, xmmword ptr [rdi + 4*rdx]
3      movaps xmm3, xmm1
4      cmpltps xmm3, xmm2
5      addps xmm2, xmm0
6      andps xmm2, xmm3
7      andnps xmm3, xmm0
8      movaps xmm0, xmm3
9      orps xmm0, xmm2
10     add rdx, 4
11     cmp rcx, rdx
12     jne .LBB0_7
13
14
```

Listing A.4: Clang 8.0.0 output (main loop body)

Listing A.1 presents a C++ example that makes use of SSE intrinsics. Function `square()` squares each element of an array of **`float`**s. To use SSE intrinsics, we must include the `xmmintrin.h` header Line 1).

Intrinsic functions `_mm_loadu_ps()` and `_mm_storeu_ps()` in Lines 4 and 6 perform a contiguous load and store of 4 **`float`**s from memory, respectively. Intrinsic function `_mm_mul_ps()` performs 4 multiplications of **`float`**s simultaneously and yields 4 products. Thanks to these intrinsics, we can now square 4 elements of `arr` simultaneously; as such, we can modify the index `i` (Line 3) to be increased by 4 at each iteration, rather than 1.

Automatic vectorisation Another approach to using SIMD is by delegating its use to a compiler optimisation pass that detects pieces of code where SIMD can be used and transforms them accordingly (*e.g.,* SLP vectorisation [LA00]).

The benefit of autovectorisation is that the task of using SIMD is left to the compiler, hence developers need not modify their original, *scalar* code. This is also beneficial in terms of portability, as it is unnecessary to write architecture-specific variations of the same business logic.

In practice, however, it is not unusual for autovectorisation to yield varying results depending on the compiler vendor and/or version being used. Listing A.2 presents a function, `sum_gt10` that yields the sum of an array of **`float`**s, but only for elements greater than 10. Listing A.3 presents the x86-64 assembly generated for the loop body in Clang 7.0.1 (released in December 2018); this assembly has not been autovectorised. Listing A.4, on the other hand, presents the

(a) AoS representation (b) SoA representation

Figure A.2: In-memory representation of AoS and SoA layouts

x86-64 assembly generated for the main loop body by Clang 8.0.0 (released in December 2019); *this loop has been successfully vectorised*[2].

Indeed, if we observe the generated code, the significant difference between the two listings corresponds to Lines 2, 4 and 5. In Listing A.3, the instructions on these lines are *scalar*, hence they operate on one element at a time, whereas in Listing A.4 the instructions on these lines are SIMD instructions [Int11].

As such, although autovectorisation is certainly beneficial, we cannot necessarily rely on it if we want to ensure that performance guarantees regarding our code are explicitly met.

A.2 The impact of layouts

Another barrier to the wider usage of SIMD is the behaviour of memory access instructions. Typically, SIMD memory instructions access memory in a *contiguous manner*. This makes memory layouts such as Struct-of-Arrays (SoA) better suited for SIMD compared to an Array-of-Structs (AoS) layout (Figure A.2).

To mitigate this lack of flexibility, some instruction sets have recently introduced *scatter–gather* instructions (*e.g.,* the AVX extension on the x86-64), which allow much greater flexibility with respect to accessing memory. However, even with scatter–gather instructions taken into account, an SoA layout is bound to be better at exploiting spatial locality (§ 1.1), which implies better cache utilisation. As such, use of SoA layouts for SIMD is still worthwhile.

[2] Examined with `https://godbolt.org` and by passing the arguments `-O2 -ffast-math -fno-unroll-loops`.

A.3 Related work: ISPC

With regards to extending SHAPES with concepts and constructs tailored to SIMD, we will be drawing inspiration from the Intel ISPC project [PM12]. ISPC (Implicit SPMD Program Counter) is a compiled, statically typed language developed with the intent to allow the writing of code in a scalar, C-like language with concepts and semantics that make it easier to emit performant SIMD code. This makes it possible to write high-level code that is portable across various SIMD architectures (*e.g.,* SSE, ARM Neon [Lim]) while making it easier to predict what the generated assembly code will be, compared to autovectorisation.

Similar to SHAPES-z (§ 6), ISPC is also designed to be an embeddable language: Developers write their performance-critical portions in ISPC and then invoke ISPC code from within their existing C and C++ code.

A.3.1 ISPC execution model

The ISPC execution model consists of k instances or *lanes*[3] that are executing concurrently. The number k of instances is architecture-specific (*e.g.,* $k = 4$ for SSE, $k = 8$ for AVX). These lanes run in a lock-step manner: All lanes are always executing the exact same instruction at the exact same address; to execute the subsequent statement, all lanes must have completed execution of the previous statement.

ISPC variables and expressions are either `uniform` or `varying`:

- `uniform` variables or expressions are guaranteed to evaluate to the same value across all lanes.
- `varying` variables or expressions, on the other hand, can evaluate to different values across different lanes.

Variables without a `uniform` or `varying` declaration are implicitly treated as `varying`.

[3] Also called "gangs" in the ISPC documentation.

Listing A.5: Handling branches with an execution mask

The concepts of **uniform** and **varying** variables and expressions are borrowed from shader languages (*e.g.,* GLSL for OpenGL [KBR14] and HLSL for DirectX [Cor15]), as well as Pixar's Renderman shader language [HL90].

A.3.2 Execution mask: Handling divergence

The *"multiple lanes running in lockstep"* execution model of § A.3.1 handles sequential code trivially. To handle branching and looping constructs, these languages introduce the concept of an *execution mask*. An execution mask indicates which lanes are active or inactive at any given time; a statement executed by an inactive lane is effectively turned into a no-op.

Listing A.5 presents an execution mask example, with respect to an **if** statement. Initially, the condition is evaluated for all lanes and an execution mask is generated (Line 2). This execution mask determines which lanes correspond to either the **then** or **else** branch, respectively.

The **then** statement is executed afterwards (Line 3). During execution, only the currently active lanes (*i.e.,* the ones where the condition was evaluated to **true**) are running; the other lanes are inactive and they cannot, therefore, produce side effects. To execute the **else** branch, all lanes that were active now become inactive and vice-versa (Line 4). Then, the **else** branch is executed only for the newly active lanes (Line 5). On Line 6, all lanes have now converged, hence the execution mask indicates that all lanes are active.

The execution mask model can be used to implement a plethora of constructs with diverging execution such as **if**, **while**, **break**/**continue**, and **switch**. For instance, within a **while** loop, the

```
1  export void example(uniform float varr[], uniform int count) {
2    foreach (i = 0 ... count) {
3      float v = varr[i];
4      if (v < 3.)
5        v = v * v;
6      else
7        v = sqrt(v);
8      varr[i] = v;
9    }
10 }
```

Listing A.6: ISPC code that traverses and modifies an array

execution mask expresses which lanes are still executing the loop and which ones have exited it; the loop will be exited only when the execution mask indicates that all lanes have exited the loop.

A noteworthy observation is that despite the usefulness of the execution mask model, it cannot be used to implement constructs such as calls to **varying** function pointers (*e.g.,* virtual function tables) or a **goto** where divergence may be possible.

A.3.3 An ISPC example

We will now demonstrate the syntax and semantics of ISPC with an example. Listing A.6 presents an example method written in ISPC. Function `example()` is declared as **export** (Line 1), which indicates that `example()` is intended to be called from external C/C++ code.

Parameters `varr` and `count` (Line 1) are declared as **uniform**, hence they are guaranteed to have the same value across all lanes. Local variable `v` (Line 3) is implicitly defined as **varying**; its value can differ between different lanes.

The **foreach** statement (Line 2) allows the developer to, among other things, write code that iterates over an array of N elements. At every loop iteration, k elements will be processed (where k is the number of lanes), hence the **foreach** loop will be executed N/k times. If k does not evenly divide N, then an additional iteration corresponding to the "leftover" $N \bmod k$ elements will be executed.

```
template<int L>
class Vec3 {
public:
    simd(L) Vec3(float varying(L) xx,
                 float varying(L) yy,
                 float varying(L) zz): x(xx), y(yy), z(zz) {}
    simd(L) Vec3 <L> operator+(Vec3 <L> v) {
        Vec3 <L> result;
        result.x = x + v.x;
        result.y = y + v.y;
        result.z = z + v.z;
        return result;
    }
    float varying(L) x, y, z;
};
```

Listing A.7: Sierra future extensions suggested in [LHH14].

A.4 Related work: Sierra

Similar to ISPC, Sierra [LHH14] is an extension on top of the C++ syntax and semantics with the aim of writing code that better exploits SIMD capabilities.

Sierra also provides a SIMD environment, wherein, similar to ISPC, execution consists of k lanes executing in a lockstep manner. The main difference with respect to Sierra is that a SIMD environment is annotated explicitly as `simd(k)`, with k being the number of SIMD lanes.

In § 3.3.3, we presented the layout transformation capabilities of the `varying` keyword on `struct` types. These capabilities can be also extended: Two future extensions that have been discussed [LHH14] for Sierra are the possibility of allowing the `varying` keyword to also affect classes, as well as the possibility of allowing an explicit template parameter to specify the constant `k` in a `varying(k)` declaration.

Listing A.7 presents an example that combines these two extensions.

A.5 SHAPES^{SIMD}

From our examination of the ISPC and Sierra syntax and semantics, we argue that both ISPC and Sierra are worthwhile languages in terms of concepts and semantics to expand upon. To that extent, and given the data placement constructs that SHAPES exhibits, we believe that incorporating the concept of SIMD lanes of execution into SHAPES is a worthwhile endeavour. We will call this enrichment of SHAPES with SIMD constructs SHAPES^{SIMD}.

Because SHAPES is primarily a scalar language, we argue that an explicit `simd` execution environment should be added, similar to that of Sierra. Within this execution environment, SHAPES^{SIMD} exits scalar mode and enters SIMD mode (and the semantics of the code within the `simd` block are changed accordingly).

We envision that our syntax will mostly resemble that of ISPC, due to our expectation that ISPC will "feel" more familiar to developers who are accustomed to shader languages.

A feature of ISPC which we argue that it should be an essential part of SHAPES^{SIMD} is a construct similar to that of ISPC's `foreach` statement. For instance, the `foreach` statement as implemented currently in SHAPES-z (§ 6.1) allows the traversal of all elements within a pool. However, as it stands right now, we introduced the pool traversal introduced in SHAPES-z in an ad hoc fashion, so as to primarily support the implementation of our case studies.

As such, we argue that it is necessary to formalise the concept of pool-backed arrays within the formal model of SHAPES. One possible approach we envision this could be performed is as follows:

Pool-backed arrays A common implementation of arrays with respect to managed languages (*e.g.,* Java [GJS+14]) is that arrays are implemented as arrays to *references* of objects, hence it is possible to have two array elements that point to the same object. Considering that this would introduce an additional level of indirection and our aim is to eliminate unnecessary scatter–gather SIMD operations, such an implementation detail would be undesirable for SHAPES^{SIMD}.

A possible solution would be to implement value semantics: Indexing into the array yields a copy of the object and assigning into it replaces the object to be assigned with a copy of the value assigned. This way, each array element contains a unique object. However, this approach may not be always suitable and it may incur additional complexity and overhead. Instead, we make use of *unique pointers* to array elements.

Unique pointers A *unique pointer* [Hog91] is a pointer that takes ownership of an object. Uniqueness can be leveraged to, *e.g.,* achieve automatic memory management in an unmanaged language (*e.g.,* `std::unique_ptr` in C++11 [ISO12]). We exploit this property for our implementation: An array of n unique pointers to objects can be simply implemented via a section of memory with enough capacity to accommodate n objects.

Since assigning to a unique pointer drops the currently present object, one possible way of implementing assignment to unique pointers would be to incorporate move semantics [ISO12]: The unique pointer effectively takes ownership of the value assigned to it.

Borrowing A downside of unique pointers is the fact that due to move semantics, reading the underlying value will cause the unique pointer to release ownership of that value. Borrowing [Boy01] relaxes the constraints of unique pointers for a specific (usually a lexical) scope and is intended to alleviate the issue of move semantics of unique pointers. We use borrowing to access the array elements and obtain references to both objects themselves and fields of objects.

Pools exclusive to arrays In some cases, it can be vital from a performance perspective to ensure that an array spans the entire pool. This bears a lot of resemblance to the current implementation of `foreach` in SHAPES-z. As such, we argue that formalising this approach of array representation within SHAPES is something worth considering.

A.5.1 Discussion

The combination of arrays and pools allows a developer to specify an array object that uses the already familiar syntax of accessing/modifying array elements, and also gives them the ability to change the underlying representation from AoS to SoA by simply changing a pool's layout.

Now, an array A with an exclusive pool P whose elements have type `unique` T, can *automatically* be represented in memory as a contiguous storage of objects (laid out according to the layout specification of P) in the same order as they are held by the array. (Unique pointers even make it possible to obtain the same layout for a singly linked list.)

Inside a **`simd`** block, the borrowing construct allows us to directly manipulate elements in arrays, voiding the need to assign to all of the fields of an object simultaneously.

Gaps in the Data The implementation details of pools in SHAPES are abstracted away from the developer. One such detail is the possibility of gaps between objects inside a pool. That is, these gaps can be filled in later when a new object is constructed inside a pool. We believe that making pool-backed arrays expose this feature is beneficial to the developer. The ability to guarantee that no gaps exist in the array (*e.g.,* by leveraging moving GC) can be added as a later extension.

Moreover, despite the possibility of gaps in our implementation, we expect that the code generated will not be suboptimal, as we can exploit the fact that reading from and writing into the fields of empty slots should not affect the semantics of well-behaving programs and that conditional statements will not be translated into suboptimal code.

A.6 Conclusion

All of the extensions to SHAPES we proposed in § A.5 are useful on their own, regardless of whether or not we are taking SIMD into account. By combining them and the **`simd`** environment,

we hope that these features will provide an environment that, thanks to intersectionality, allows better exploitation of SIMD compared to existing approaches.

Moreover, despite the possibility of gaps in our implementation, we expect that the code generated will not be suboptimal, as we can exploit the fact that reading from and writing into the fields of empty slots should not affect the semantics of well-behaving programs and that conditional statements will not be translated into suboptimal code.

Appendix B

The SHAPES-z Grammar

We now present the SHAPES-z grammar. Note that, for the sake of conciseness, we omit parts such as the precedence and associativity of binary operators, as well as features such as allowing trailing commas in method call arguments. As such, the grammar presented is deliberately incomplete.

⟨*program*⟩ ::= (⟨*program-definition*⟩)* EOF

⟨*program-definition*⟩ ::= ⟨*class-definition*⟩
| ⟨*layout-definition*⟩

⟨*class-definition*⟩ ::= '`class`' ⟨*IDENT*⟩ '`<`' ⟨*identifier-list*⟩ '`>`' '`where`' ⟨*pool-bound-list*⟩ '`{`' (⟨*class-member*⟩)* '`}`'

⟨*layout-definition*⟩ ::= '`layout`' ⟨*IDENT*⟩ '`:`' ⟨*IDENT*⟩ '`=`' ⟨*cluster-list*⟩ '`;`'

⟨*pool-bound-list*⟩ ::= ⟨*pool-bound*⟩ ('`,`' ⟨*pool-bound*⟩)*

⟨*pool-bound*⟩ ::= ⟨*IDENT*⟩ '`:`' ⟨*bound-type*⟩

⟨*bound-type*⟩ ::= '`[`' ⟨*IDENT*⟩ '`<`' ⟨*identifier-list*⟩ '`>`' '`]`'

⟨*layout-type*⟩ ::= ⟨*IDENT*⟩ '`<`' ⟨*layout-ident-list*⟩ '`>`'

⟨*layout-ident-list*⟩ ::= ⟨*layout-ident*⟩ ('`,`' ⟨*layout-ident*⟩)*

⟨*layout-ident*⟩ ::= ⟨*IDENT*⟩ | '`none`'

⟨*object-type*⟩ ::= ⟨*IDENT*⟩ '<' ⟨*identifier-list*⟩ '>'

⟨*primitive-type*⟩ ::= 'bool' | 'i8' | 'u8' | 'i16' | 'u16' | 'i32' | 'u32' | 'i64' | 'u64'
| 'f32' | 'f64'

⟨*type*⟩ ::= ⟨*object-type*⟩ | ⟨*primitive-type*⟩

⟨*class-member*⟩ ::= ⟨*field-declarations*⟩ | ⟨*method-declaration*⟩

⟨*cluster-list*⟩ ::= ⟨*cluster*⟩ ('+' ⟨*cluster*⟩)*

⟨*cluster*⟩ ::= 'rec' '{' ⟨*identifier-list*⟩ '}'

⟨*field-declarations*⟩ ::= ⟨*declaration-list*⟩ ';'

⟨*method-declaration*⟩ ::= 'fn' ⟨*IDENT*⟩ '(' (⟨*declaration-list*⟩)? ')' (':' ⟨*type*⟩)? ⟨*block-stmt*⟩

⟨*declaration*⟩ ::= ⟨*IDENT*⟩ ':' ⟨*type*⟩

⟨*expr*⟩ ::= '(' ⟨*expr*⟩ ')'
| ⟨*un-op*⟩ expr
| ⟨*expr*⟩ ⟨*bin-op*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ '.' ⟨*IDENT*⟩
| ⟨*expr*⟩ '.' ⟨*IDENT*⟩ '(' (⟨*method-call-args*⟩)? ')'
| ⟨*IDENT*⟩ '(' (⟨*method-call-args*⟩)? ')'
| ⟨*IDENT*⟩
| ⟨*IDENT*⟩ '[' ⟨*expr*⟩ ']'
| 'new' ⟨*object-type*⟩
| ⟨*expr*⟩ 'as' ⟨*primitive-type*⟩
| 'this' | 'null'
| ⟨*INT-CONST*⟩ | ⟨*DOUBLE-CONST*⟩ | 'true' | 'false'

⟨*un-op*⟩ ::= '+' | '-' | '!'

⟨*bin-op*⟩ ::= '+' | '-' | '*' | '/' | '<<' | '>>' | '&' | '|' | '^' | '&&' | '||'
| '<' | '>' | '<=' | '>=' | '==' | '!='

⟨*method-call-args*⟩ ::= ⟨*expr*⟩ (',' ⟨*expr*⟩)*

⟨*stmt*⟩ ::= ⟨*block-stmt*⟩
| ‘;’
| ‘`let`’ ⟨*declaration-list*⟩ ‘;’
| ‘`pools`’ ⟨*pool-declaration-list*⟩ ‘;’
| ‘`pool`’ ⟨*pool-declaration*⟩ ‘;’
| ⟨*expr*⟩ ‘=’ ⟨*expr*⟩ ‘;’
| ⟨*expr*⟩ ⟨*op-assign*⟩ ⟨*expr*⟩ ‘;’
| ‘`if`’ ⟨*expr*⟩ ⟨*block-stmt*⟩ (‘`else`’ ⟨*block-stmt*⟩)?
| ‘`while`’ ⟨*expr*⟩ ⟨*block-stmt*⟩
| ‘`foreach`’ ⟨*IDENT*⟩ ‘=’ ⟨*expr*⟩ ‘..’ ⟨*expr*⟩ ⟨*block-stmt*⟩
| ‘`foreach`’ ⟨*IDENT*⟩ ‘:’ ⟨*IDENT*⟩ ⟨*block-stmt*⟩
| ‘`break`’ ‘;’
| ‘`continue`’ ‘;’
| ‘`return`’ (⟨*expr*⟩)? ‘;’
| ⟨*expr*⟩ ‘;’

⟨*block-stmt*⟩ ::= ‘{’ (⟨*stmt*⟩)* ‘}’

⟨*op-assign*⟩ ::= ‘+=’ | ‘-=’ | ‘*=’ | ‘/=’ | ‘<<=’ | ‘>>=’ | ‘&=’ | ‘|=’ | ‘^=’

⟨*pool-declaration*⟩ ::= ⟨*IDENT*⟩ ‘:’ ⟨*layout-type*⟩

⟨*declaration-list*⟩ ::= ⟨*declaration*⟩ (‘,’ ⟨*declaration*⟩)*

⟨*pool-declaration-list*⟩ ::= ⟨*pool-declaration*⟩ (‘,’ ⟨*pool-declaration*⟩)*

⟨*identifier-list*⟩ ::= ⟨*IDENT*⟩ (‘,’ ⟨*IDENT*⟩)*

Appendix C

Proof sketches

C.1 Paths

In order to express the definition of reachable objects, we make use of paths. We define paths as follows:

$$path \in Path ::= x \mid path.f$$

Given a specialised typing context Δ, we would like to require that our paths are well-formed with respect to Δ. That is, $\exists t.\ \Delta \vdash path : t$ for a given path $path$. Because for simplicity reasons the syntax of SHAPES^h as defined in § 4 does not permit complex paths, *i.e.*, $x.f.g$, we define the following typing rules for paths:

$$\frac{}{\Delta \vdash x : \Delta(x)} \qquad \frac{\Delta \vdash path : C\langle ps \rangle}{\Delta \vdash path.f : \mathcal{F}(C, f)[\mathcal{P}s(C)/ps]}$$

C.1.1 High-Level Paths

To evaluate paths, we define the following variant of the operational semantics, wherein a specialised context, a high-level configuration and a path reduce to an object address. This variant is of the form $\Delta, \mathcal{X}, \Phi, \mathit{path} \rightsquigarrow \beta$.

$$\textsc{Variable Path (HL)}\quad \frac{}{\Delta, \mathcal{X}, \Phi, x \rightsquigarrow \Phi(x)} \qquad \textsc{Null Path (HL)}\quad \frac{\Delta, \mathcal{X}, \Phi, \mathit{path} \rightsquigarrow \mathbf{null}}{\Delta, \mathcal{X}, \Phi, \mathit{path}.f \rightsquigarrow \mathbf{null}}$$

$$\textsc{Object Path (HL)}\quad \frac{\Delta \vdash \mathit{path} : C\langle\_\rangle \quad \Delta, \mathcal{X}, \Phi, \mathit{path} \rightsquigarrow \omega \quad \mathcal{X}(\omega) = (C, \_, \rho)}{\Delta, \mathcal{X}, \Phi, \mathit{path}.f \rightsquigarrow \rho(f)}$$

C.1.2 Low-Level Paths

To evaluate paths in SHAPES_ℓ, we define the following variant of the operational semantics, wherein a specialised context, a low-level configuration and a path reduce to either an address (in the case of a standalone object), or an index (in the case of a pool-allocated object) corresponding to an object in the pool (and we can determine the address of the pool in question by inferring the type of *path* under Δ). This variant is, similarly, of the form $\Delta, \chi, \phi, \mathit{path} \rightsquigarrow \gamma$.

Variable Path (LL)

$$\frac{}{\Delta, \chi, \phi, x \rightsquigarrow \sigma(x)}$$

Null Path (LL)

$$\frac{\Delta, \chi, \phi, path \rightsquigarrow \textbf{null}}{\Delta, \chi, \phi, path.f \rightsquigarrow \textbf{null}}$$

Heap Object Path (LL)

$$\frac{\Delta \vdash path : C\langle np \cdot \_ \rangle \quad \Delta \vdash np : \textbf{None} \quad \Delta, \chi, \phi, path \rightsquigarrow \alpha \quad \mathcal{O}(C, f) = i}{\Delta, \chi, \phi, path.f \rightsquigarrow \chi(\alpha + i)}$$

Pool Object Path (LL)

$$\frac{\Delta \vdash path : C\langle np \cdot \_ \rangle \quad \Delta \vdash np : L\langle \_ \rangle \quad \Delta, \chi, \phi, path \rightsquigarrow k \quad \mathcal{O}(L, f) = (i, j) \quad N = |\mathcal{C}l(L)[i]|}{\Delta, \chi, \phi, path.f \rightsquigarrow \chi(\chi(\phi(np) + i + 2) + N * k + j)}$$

Apart from the Null Path, the rest of the rules are standard. The rationale for Null Path is to handle the case of a possible **null** dereference whilst traversing a path.

To define equivalence between a high-level and a low-level configuration, we will make use of an injection $\mathcal{I} : Address \rightarrow Address^{\ell} \cup (Address^{\ell} \times Index^{\ell})$

For a given sequence of typing contexts Δs, equivalence between a high level and a low level configuration via an injection $\mathcal{I}$ is represented using the notation $\mathcal{X}, \Sigma \simeq_{\Delta s, \mathcal{I}} \chi, \sigma$ and is defined as follows:

C.2 Correctness of Compilation Theorems

We will now prove that translation is meaning preserving.

Definition C.1. *Equivalence between high-level and low-level addresses under an injection* $\mathcal{I} : Address \mapsto Address^{\ell} \cup (Address^{\ell} \times Index^{\ell})$ *is defined as:*

$$\beta \simeq_{\mathcal{I},ps,\sigma} \gamma \textit{ iff}$$
$$[\beta = \gamma = \textbf{null}] \ \vee \ [\sigma(ps[0]) = \textbf{null} \ \wedge \ \mathcal{I}(\beta) = \gamma] \ \vee \ [\sigma(ps[0]) = \alpha \neq \textbf{null} \ \wedge \ \mathcal{I}(\beta) = (\alpha, \gamma)]$$

Equivalence between a high-level and a low-level configuration is defined as:

$$\mathcal{X}, \Sigma \simeq_{\Delta s,\mathcal{I}} \chi, \sigma \textit{ iff}$$
$$[\forall p, i.\ (\Sigma[i](p) = \textbf{none} \ \wedge \ \sigma[i](p) = \textbf{null}) \ \vee \ \mathcal{I}(\Sigma[i](p)) = \sigma[i](p)] \ \wedge$$
$$[\forall np, i, path, C, \beta, \gamma.\ [\Delta s[i] \vdash path : C\langle np \cdot \_\rangle \ \wedge$$
$$\Delta s[i], \mathcal{X}, \Sigma[i], path \rightsquigarrow \beta \ \wedge \ \Delta s[i], \chi, \sigma[i], path \rightsquigarrow \gamma \rightarrow \beta \simeq_{\mathcal{I},np,\sigma} \gamma]$$

Garbage collection

We use the notation $\chi, \sigma \simeq_{\Delta s,\mathcal{J}} \chi', \sigma'$ to indicate that two low-level configurations χ, σ and χ', σ' are equivalent under the injection $\mathcal{J}$ for a given sequence of specialised contexts. The definition of equivalence under $\mathcal{J}$ is presented in Definition 5.3 (found in § 5.5.1).

The definition ensures that if a path *path* yields two standalone or pool-allocated object addresses in the two configurations, then a mapping between them must exist and if both are allocated in a pool, then a mapping between the corresponding pool addresses must also exist. Furthermore, we require that all variables in a stack frame corresponding to pools must have a mapping between them (assuming they point to pools).

We now define the following theorem, which states that evaluating the same statement sequence under two equivalent low-level configurations reaches two low-level configurations that are also equivalent. This theorem allows us to reason that performing a GC on SHAPES_ℓ will preserve the execution semantics. It shows that as long as two configurations are equivalent with respect to reachable objects, then the resulting configurations and values yielded will also be equivalent.

Theorem C.2 (Equivalent low-level configurations will transition into new equivalent low-level configurations)**.** *Let χ_1, σ_1 and χ_2, σ_2 be two low level configurations, let Δs be their corresponding typing contexts and let $\mathcal{J}$ be an injection such that $\chi_1, \sigma_1 \simeq_{\Delta s,\mathcal{J}} \chi_2, \sigma_2$. Then, for a sequence of statements stmts such that $\Delta s[0] \vdash stmts : C\langle p_1 \,..\, p_k\rangle$, if $\chi_1', \sigma_1', \gamma_1$ and $\chi_2', \sigma_2', \gamma_2$ exist such*

that:

$$\chi_1, \sigma_1, [\![stmts]\!]_{\Delta s[0]} \rightsquigarrow \chi_1', \sigma_1', \gamma_1 \qquad \textit{and} \qquad \chi_2, \sigma_2, [\![stmts]\!]_{\Delta s[0]} \rightsquigarrow \chi_2', \sigma_2', \gamma_2$$

And $\Delta' = \mathbf{this}\colon C\langle np_1\ ..\ np_k\rangle, np_1\colon \Delta s[0](np_1),\ ..\ , np_n\colon \Delta s[0](np_n)$*, then there exists an injection* $\mathcal{J}'$ *such that* $\chi_1', \phi_1' \cdot \sigma_1' \simeq_{\Delta' \cdot \Delta s, \mathcal{J}'} \chi_2', \cdot\sigma_2'$*. where*

$$\phi_1' = [\mathbf{this} \mapsto \gamma_1, np_1\ ..\ np_k \mapsto \sigma_1'(np_1\ ..\ np_k)]$$
$$\phi_2' = [\mathbf{this} \mapsto \gamma_2, np_1\ ..\ np_k \mapsto \sigma_2'(np_1\ ..\ np_k)]$$

C.3 Proof sketches

Proof of Lemma 4.3. For this proof, we will make use of the utility predicate $pbds_1 \simeq pbds_2$ that is defined as follows:

$$\frac{pbds_1 \simeq pbds_2}{pbds_2 \simeq pbds_1} \qquad \frac{pbds_1 \simeq pbds_2 \quad pbd_1 \simeq pbd_2}{pbd_1 \cdot pbds_1 \simeq pbd_2 \cdot pbds_2} \qquad \frac{}{\epsilon \simeq \epsilon}$$

$$\frac{}{[C\langle ps\rangle] \simeq [C\langle ps\rangle]} \qquad \frac{}{\mathbf{None} \simeq \mathbf{None}} \qquad \frac{}{\mathbf{None} \simeq [C\langle ps\rangle]} \qquad \frac{\mathcal{C}l(L) = C}{L\langle ps\rangle \simeq [C\langle ps\rangle]}$$

Thus we can redefine $\forall i.\ \Gamma \vdash ps[i] :\ pbds[i]$ as $\Gamma(ps) \simeq pbds$. Let:

$$qs = \mathcal{P}s(C) \quad qbs = \mathcal{P}b(C)$$
$$rs = \mathcal{P}s(C') \quad rbs = \mathcal{P}b(C')$$

From Rule ObjTypeWF and from the above definition of $\simeq$, to show that $\Gamma \vdash C'\langle ps'[qs/ps]\rangle$, we need to show that:

$$\Gamma(ps'[qs/ps]) \simeq rbs[rs/ps'[qs/ps]] \tag{C.1}$$

Because $\Gamma \vdash C\langle ps \rangle$, we have that

$$\Gamma(ps) \simeq qbs[qs/ps] \tag{C.2}$$

Because our program *prog* is well-formed, if Γ_C is the environment used to compile class C (*cf.*, Definition 4.6), then:

$$\Gamma_C(ps') \simeq rbs[rs/ps'] \tag{C.3}$$

We now define the following:

$$\begin{aligned} Qs &= qs \cdot \mathbf{none} & QBs &= qbs \cdot \mathbf{None} \\ Rs &= rs \cdot \mathbf{none} & RBs &= rbs \cdot \mathbf{None} \\ Ps &= ps \cdot \mathbf{none} \\ Ps' &= ps' \cdot \mathbf{none} \end{aligned}$$

For convenience, let $\Gamma(\mathbf{none}) = \mathbf{None}$. Because **none** is the last pool parameter in Ps, Ps', Qs, Rs (hence the substitution *e.g.*, $[Qs/Ps]$ will replace **none** with **none**), it will also hold from Equations (C.2) and (C.3) that:

$$\Gamma(Ps) \simeq QBs[Qs/Ps] \tag{C.4}$$

$$\Gamma_C(Ps') \simeq RBs[Rs/Ps'] \tag{C.5}$$

Furthermore, if we can show that:

$$\Gamma(Ps'[Qs/Ps]) \simeq RBs[Rs/Ps'[Qs/Ps]] \tag{C.6}$$

Then this suffices to show that Equation (C.1) holds.

Let $\operatorname{Im} f$ be the image of function f.

Well-formedness of *prog* implies that each of the parameters in ps' will be either **none** or originate from rs, therefore there exists a function $\sigma : \{0, \ldots, |Ps'| - 1\} \mapsto \{0, \ldots, |Qs| - 1\}$ such that $\forall i.\ Ps'[i] = Qs[\sigma(i)]$.

Because *prog* is well-formed, each pool parameter from Ps' will also agree to their pool bound defined in C, hence:

$$\Gamma(Ps') \simeq QBs[\operatorname{Im} \sigma]$$

Thus:

$$RBs[Rs/Ps'][\operatorname{dom}(\sigma)] \simeq QBs[\operatorname{Im} \sigma] \tag{C.7}$$

Therefore:

$$
\begin{aligned}
\Gamma(Ps'[Qs/Ps]) &= \Gamma(Ps[\operatorname{Im}\sigma]) && \text{From the definition of } \sigma \\
&\simeq QBs[Qs/Ps][\operatorname{Im}\sigma] && \text{From Equation (C.4)} \\
&\simeq RBs[Rs/Ps'][\operatorname{dom}(\sigma)][Qs/Ps] && \text{From Equation (C.7)} \\
&= RBs[Rs/Ps'][Qs/Ps][\operatorname{dom}(\sigma)] && \text{By structure} \\
&= RBs[Rs/Ps'][Qs/Ps] && \text{Since } |\operatorname{dom}(\sigma)| = |Ps'| = |Rs| \\
&= RBs[Rs/Ps'[Qs/Ps]] && \text{By structure}
\end{aligned}
$$

Hence Equation (C.6) holds.

□

Proof of Lemma 4.4. We prove this theorem by structural induction over the derivation of e:

- Rule Value (**null**): Trivial.
- Rules Variable and New Object (x/**this** and **new** t, respectively): From Definition 4.6 (well-formed program).
- Rule Object Read ($x.f$): Let $\Gamma \vdash x : C\langle ps \rangle$. Then we apply Lemma 4.4 given that $\vdash \Gamma$, $\Gamma \vdash C\langle ps \rangle$, and $\Gamma_C \vdash \mathcal{F}(C, f)$, where Γ_C is the typing environment class C was typechecked against.
- Rule Object Write ($x.f = x$): The type of the variable x is well-formed from Definition 4.6 (well-formed program).
- Rule MethodCall ($x.m(x'')$): Shown in a manner similar to that of Rule Object Read, except with respect to the return type of method m in class C.
- Rule Assignment ($x = e$): By structural induction over the derivation of e.

□

Proof of Theorem 4.11. We prove the theorem by cases:

- Rule Value: Configuration is not mutated and **null** weakly agrees with any object type.
- Rule Variable: Configuration is not mutated and it holds that $\mathcal{X} \vDash \Sigma(x) : C\langle\Sigma(ps)\rangle$ from the definition of well-formed configuration.
- Rule Assignment: By structural induction over the derivation of e, we obtain a new well-formed configuration $\mathcal{X}', \Sigma'$ and a value β such that $\mathcal{X}' \vDash \beta : C\langle\Sigma'(ps)\rangle$. Variable x corresponds to an object and from the definition of well-formed configuration it holds that $\mathcal{X}' \vDash \Sigma'(x) : C\langle\Sigma'(ps)\rangle$, therefore assigning β to x will not break well-formedness of the configuration or break the weak agreement requirement for β.
- Rule New Object: We augment the heap $\mathcal{X}$ with a new object ω; we only need to show that $\mathcal{X}' \vDash \omega \triangleleft C\langle\Sigma(ps))\rangle$ in the augmented heap $\mathcal{X}'$. Since we set the object's type to C, its pool parameters to $\Sigma(ps))$ and initialise its fields $fs = \mathcal{F}s(C)$ to **null**, we only need to show that $\mathcal{X} \vDash \Sigma(ps[0]) : [C\langle\Sigma(ps)\rangle]$.

  Because $\Gamma s[0] \vdash C\langle ps\rangle$, it holds from Rule ObjTypeWF that $\Gamma s[0] \vdash ps[0] :: [C\langle ps\rangle]$, hence from the definition of well-formed configuration, $\mathcal{X} \vDash \Sigma(ps[0]) : [C\langle\Sigma(ps)\rangle]$ will hold, regardless of whether $ps[0]$ is **none**, corresponds to a pool with layout L such that $\mathcal{C}l(L) = C$ or a formal pool parameter from $\mathcal{P}s(C)$. And because $\mathcal{X}'$ is an augmentation of $\mathcal{X}$, it will also hold that $\mathcal{X}' \vDash \Sigma(ps[0]) : [C\langle\Sigma(ps)\rangle]$.
- Rule Object Read: Configuration is not mutated, hence well-formedness of the configuration is preserved.

  Suppose that $\Gamma \vdash x : C'\langle ps'\rangle$, hence $\Gamma \vdash x.f : \mathcal{F}(C', f)[\mathcal{P}s(C')/ps']$ and suppose that $C\langle ps\rangle = \mathcal{F}(C', f)[\mathcal{P}s(C')/ps']$. If $\mathcal{X}(\omega) = (C, \pi s, \rho)$ and $\rho(f) = \beta$, then we want to show that $\mathcal{X}' \vDash \beta : C\langle\Sigma(ps)\rangle$.

  From the definition of well-formed configuration, it holds that $\mathcal{X} \vDash \Sigma(x) \triangleleft C'\langle\Sigma(ps')\rangle$, hence $\mathcal{X} \vDash \beta : \mathcal{F}(C', f)[\mathcal{P}s(C')/\Sigma(ps')]$.

  Because $C\langle ps\rangle = \mathcal{F}(C', f)[\mathcal{P}s(C')/ps']$, each pool parameter $ps[i]$ is either **none** or it comes from ps' (*i.e.,* there exists j such that $ps[i] = ps'[j]$). This is because each pool

parameter of the type $\mathcal{F}(C', f)$ can be **none** or come from $\mathcal{P}s(C')$ and the pool parameters $\mathcal{P}s(C')$ are substituted by ps'. Therefore $C\langle\Sigma(ps)\rangle = \mathcal{F}(C', f)[\mathcal{P}s(C')/\Sigma(ps')]$, hence $\mathcal{X}' \vDash \beta\colon C\langle\Sigma(ps)\rangle$.

- Rule Object Write: Similar to Rule Object Read, it can be shown that if $\Gamma \vdash x.f : C\langle ps\rangle$, $\mathcal{X}(\omega) = (C, \pi s, \rho)$, then $\mathcal{X} \vDash \rho(f)\colon C\langle\Sigma(ps)\rangle$. From the definition of well-formed configuration, it holds that $\mathcal{X} \vDash \Sigma(x')\colon C\langle\Sigma(ps)\rangle$. Because $\rho(f)$ and $\Sigma(x')$ have the same runtime type $C\langle\Sigma(ps)\rangle$, the assignment will maintain the well-formedness of the new configuration $\mathcal{X}', \Sigma$ and it will hold that $\mathcal{X}' \vDash \Sigma(x')\colon C\langle\Sigma(ps)\rangle$.

- Rule Statement sequence: By structural induction over the derivation of *stmts* and, subsequently, the derivation of e.

- Rule MethodCall: When invoking a method, we construct a new stack Σ' and augment the heap $\mathcal{X}$ with new pools, thus resulting in the heap $\mathcal{X}'$. We will show that $\Gamma'\cdot\Gamma s \vDash \mathcal{X}', \Sigma'$ (where Γ' is the context of the function being called); because we push a new stack frame and augment the heap, we only need to show that all object and pool variables in the new stack frame in Σ' weakly agree to their "appropriate" runtime types and that the new pools in $\mathcal{X}'$ strongly agree to their "appropriate" runtime types.

  Let Φ and Φ' be the top stack frame of Σ and Σ', respectively, and that Γ' corresponds to the environment of Φ. Then, with respect to the local variables of Φ':

  - For *the* **this** *parameter*, it holds that $\mathcal{X}' \vDash \Phi'(\mathbf{this})\colon C\langle\Phi'(\mathcal{P}s(C))\rangle$, given that it holds that $\Gamma' \vdash \mathbf{this} : C\langle\mathcal{P}s(C)\rangle$. Suppose that the variable x corresponding to **this** in Φ' has the type $C\langle ps\rangle$ under Γ. Then, since $\mathcal{X}'$ augments $\mathcal{X}$, it holds that $\mathcal{X}' \vDash \Phi(x)\colon C\langle\Phi(ps)\rangle$. From the operational semantics of Rule MethodCall, we have that $\Phi(x) = \Phi'(\mathbf{this})$ and $\Phi(ps) = \Phi'(\mathcal{P}s(C))$, hence we conclude that $\mathcal{X}' \vDash \Phi'(\mathbf{this})\colon C\langle\Phi'(\mathcal{P}s(C))\rangle$.

  - For *the method parameter* x'', let x, x' be the arguments corresponding to **this**, x'', respectively, in the method call $x.m(x')$. Assume that $\Gamma \vdash x : C\langle ps\rangle$, $\Gamma' \vdash \mathbf{this} : C\langle\mathcal{P}s(C)\rangle$, and $\Gamma' \vdash x'' : C'\langle ps'\rangle$. Then, $\Gamma \vdash x' : C'\langle ps'\rangle[\mathcal{P}s(C)/ps]$ holds.

Since our program is well-formed, each pool parameter $ps'[i]$ can be either **none** or originate from $\mathcal{P}s(C)$ (*i.e.,* there exists j such that $ps'[i] = \mathcal{P}s(C)[j]$), each of the pool parameters of the type of x'' is either **none** or originates from ps, therefore it holds that $\mathcal{X}' \vDash \Phi(x') \colon C'\langle ps' \rangle[\mathcal{P}s(C)/\Phi(ps)]$. Thus, it holds that $\mathcal{X}' \vDash \Phi'(x'') \colon C'\langle ps' \rangle[\mathcal{P}s(C)/\Phi'(\mathcal{P}s(C))]$, given that $\Phi(x') = \Phi'(x'')$, $\Phi(ps) = \Phi'(\mathcal{P}s(C))$. And given the aforementioned limitation on ps', we reach the conclusion that it holds that $\mathcal{X}' \vDash \Phi'(x'') \colon C'\langle \Phi'(ps') \rangle$.

- Weak agreement on pool bounds is shown in a manner similar to that of the method argument.
- *Object local variables* (declared via **locals**) are initialised to **null**, hence weak agreement holds.
- If a *Pool local variable* p (declared via **pools**) has type $L\langle ps \rangle$ under Γ', then $\mathcal{X}' \vDash \Phi'(p) \colon C\langle \Phi'(ps) \rangle$ holds by construction (we set the pool's pool parameters to $\Phi'(ps)$).

We now show that $\vDash \mathcal{X}'$. Let p be a pool variable such that $\Gamma' \vdash p :: L\langle ps \rangle$ and let $C = \mathcal{C}l(L)$. In order to show that $\mathcal{X}' \vDash \Phi'(p) \triangleleft L\langle \Phi'(ps) \rangle\rangle$, we must show that for every i it holds that $\mathcal{X}' \vDash \Phi'(ps[i]) \colon \mathcal{B}(C, \mathcal{P}s(C)[i])[\mathcal{P}s(C)/\Phi'(ps)]$.

If $\Phi'(ps[i]) = \textbf{none}$, then weak agreement easily holds. Otherwise, since $\Gamma' \vdash L\langle ps \rangle$, it holds that $\Gamma' \vdash ps[i] :: \mathcal{B}(C, \mathcal{P}s(C)[i])[\mathcal{P}s(C)/ps]$. Because $\Phi'(ps[i]) \neq \textbf{none}$, it holds that $ps[i] \neq \textbf{none}$, hence from the definition of Γ', $ps[i]$ can only adhere to the bound $\mathcal{B}(C, \mathcal{P}s(C)[i])[\mathcal{P}s(C)/ps]$.

Since we have shown that all variables in Φ' weakly agree to their "appropriate" runtime type, it holds that $\mathcal{X}' \vDash \Phi'(ps[i]) \colon \mathcal{B}(C, \mathcal{P}s(C)[i])[\mathcal{P}s(C)/\Phi'(ps)]$.

Therefore $\Gamma' \cdot \Gamma s \vDash \mathcal{X}', \Sigma'$. By structural induction over the statement sequence *stmts*, we deduce that $\Gamma' \cdot \Gamma s \vDash \mathcal{X}'', \Sigma''$, where $\mathcal{X}'', \Sigma''$ is the configuration resulting from the evaluation *stmts*.

Let β be the value yielded by evaluating *stmts*. Then if $C'\langle ps' \rangle$ is the return type of method m, then by structural induction over *stmts* it will also hold that $\mathcal{X}'' \vDash$

$\beta\colon C'\langle\Sigma''(ps')\rangle$. We want to show that if $\Gamma \vdash x.m(x'') : C'\langle ps''\rangle$, then it holds that $\mathcal{X}'' \vDash \beta\colon C'\langle\Sigma''[1](ps'')\rangle$.

Indeed, each pool parameter in ps' can be either **none** or originate from $\mathcal{P}s(C)$. As such, each pool parameter in ps'' can be either **none** or originate from ps. Following a similar reasoning to the one used in proving weak agreement for method arguments and given that $\Sigma''(\mathcal{P}s(C)) = \Sigma''(ps)$, we conclude that $\mathcal{X}'' \vDash \beta\colon C'\langle\Sigma''[1](ps'')\rangle$.

□

We will use the following lemma to prove part of Theorem 5.5:

Lemma C.3 (GC and high-level – low-level equivalence)**.** *If it holds that* $\mathcal{X}, \Sigma \simeq_{\Delta s,\mathcal{I}} \chi, \sigma$ *and it holds that* $\chi, \sigma \simeq_{\Delta s,\mathcal{J}} \chi', \sigma'$*, then* $\mathcal{X}, \Sigma \simeq_{\Delta s,\mathcal{I}'} \chi, \sigma$*, where* $\mathcal{I}' = \mathcal{J} \circ \mathcal{I}$*.*

Proof. By structural induction over the *path*s that are well-typed under each of the specialised contexts Δs. □

Proof of Theorem 5.4. We prove this theorem on a case-by-case basis:

- Rule Value: High-level configuration need not be altered, hence $\mathcal{I}' = \mathcal{I}$.
- Rule Variable: High-level configuration need not be altered, hence $\mathcal{I}' = \mathcal{I}$.
- Rule Assignment: We obtain a new high-level configuration and injection $\mathcal{I}''$ by structural induction over the derivation of e. $\mathcal{I}''$ will also map the high-level value yielded (β) to its low-level counterpart (γ), thus assignment will not violate equivalence, hence $\mathcal{I}' = \mathcal{I}''$.
- Rule New Object: Assume that p is the variable of the pool the object is being allocated into. We distinguish two cases:
  - *New Standalone Object* (*i.e.*, $p =$ **none** or $\Delta(p) =$ **None**): $\mathcal{I}'$ extends $\mathcal{I}$ by mapping the address of the new high-level object to the address of the low-level one.

  - *New Pooled Object* (*i.e.*, $\Delta(p) = L\langle\_\rangle$): $\mathcal{I}'$ extends $\mathcal{I}$ by mapping the address of the new high-level object to the low-level address of p and the offset of the new low-level object.

- Rule Object Read: We assume no **null** dereference (which would cause the low-level to get stuck). High-level configuration need not be altered, hence $\mathcal{I}' = \mathcal{I}$. Equivalence between values yielded holds because $x.f$ is a reachable path from the top stack frame.

- Rule Object Write: We assume no **null** dereference (which would cause the low-level to get stuck). Because the addresses of x, x' are already equivalent between the high-level and low-level, mutation of the object pointed to by x will not break equivalence, hence it holds that $\mathcal{I}' = \mathcal{I}$.

- Rule Statement sequence: Before the low-level statements are executed, it is possible that the GC is invoked. In such a case, according to Lemma C.3, we will be able to obtain a new injection $\mathcal{I}'$ for the new low-level configuration. Then the theorem holds, by structural induction over the derivation of e and then by structural induction over the derivation of *stmts*.

- Rule MethodCall: High-level configuration will have n new empty pools constructed; $\mathcal{I}'$ will extend $\mathcal{I}$ to map them to their low-level counterparts. Moreover, to obtain the new high-level configuration, we have to create the new stack frame Φ and populate it with the method's parameters and initialise the local variables. The values of these variables are already equivalent under $\mathcal{I}$, hence $\mathcal{I}'$ as well. Then, by structural induction over the body of the method, we obtain a new injection $\mathcal{I}''$. $\mathcal{I}''$ will also map the high-level value yielded (β) to its low-level counterpart (γ).

□

Proof of Theorem 5.5. Theorem 5.5 is proven in a manner similar to that of Theorem 5.4, with the exception that we derive a new low-level configuration instead. The following rules need to be considered in more detail:

- Rule New Object (when constructed inside a pool): Construction of a new object in the high-level will never get "stuck", regardless of whether this object is standalone or pool-allocated. In the case of the low-level, however, if the capacity of a has been exhausted prior to allocation, execution will get stuck.

  Because pool allocation is an instruction, it is a statement, hence a statement sequence, thus Rule Garbage Collection can run before it and reduce the low-level configuration to one where the pool's capacity has not been exhausted. Thus, there exists a low-level execution wherein the execution will not have become stuck by the time we construct the object, hence there exists an injection $\mathcal{I}'$, which we can derive in a manner similar to the one given in Theorem 5.4 (Rule New Object).

- Rule Statement sequence: Although the operational semantics allow the GC to run infinitely many times before the execution of a statement sequence, we can select one such configuration where the GC has run at most once and permits statements such as Rule Pool Alloc to not get stuck due to the lack of capacity. Then, by Lemma C.3, we will obtain an injection $\mathcal{I}'$ from the high-level configuration to the configuration obtained by GC. The proof is then followed by structural induction over the derivation of e and then by structural induction over the derivation of *stmts* in a typical fashion.

- Rule MethodCall: Translation of the method call $x.m(x'')$ into SHAPES_ℓ involves invoking a specialisation m' of m. Suppose that $\Delta s[0] \vdash x : C\langle ps \rangle$. Then, the types pool parameters ps will be specialised (*i.e.,* they will be $L\langle \_ \rangle$ or **None**). Method m' is compiled under an environment Δ such that for all i, if $\Delta s[0](ps[i]) = L\langle \_ \rangle$ or $\Delta s[0](ps[i]) = \textbf{None}$, then $\Delta(\mathcal{P}s(C)[i]) = L\langle \_ \rangle$ or $\Delta(\mathcal{P}s(C)[i]) = \textbf{None}$, respectively. Therefore, if Φ is the stack frame corresponding to m, then $\Delta \cdot \Delta s \vDash \mathcal{X}, \Phi \cdot \Sigma$. The proof continues by structural induction on *stmts*, in a similar manner to that of Theorem 5.4 for Rule MethodCall.

□

Proof of Theorem C.2. We prove this theorem on a case-by-case basis:

- Rules Value, Variable and Object Read: It holds that $\mathcal{J}' = \mathcal{J}$, since they do not modify the configuration and the references returned are either **null** or they both correspond to reachable objects.

- Rule Assignment: We obtain two new low-level configurations that are equivalent over an injection $\mathcal{J}$ by structural induction over the derivation of e. $\mathcal{J}$ will also map the reference to the object yielded for the first configuration maps to its counterpart in the second configuration.

- Rule New Object: We distinguish two cases:

  - *New Standalone Object* (*i.e.,* Rule Alloc): We extend $\mathcal{J}$ to map the address of the object in the first configuration to the one in the second configuration.

  - *New Pooled Object* (*i.e.,* Rule Pool Alloc): We extend $\mathcal{J}$ to map the address of pool p and the index of the object in the first configuration to the address of pool p and the index of the object in the second configuration.

- Rule Object Write: It holds that $\mathcal{J}' = \mathcal{J}$. This is because $\mathcal{J}$ maps the objects referenced by x, x' in first configuration to their counterparts in the second configuration, thus field assignment will not break the isomorphic property of the configurations.

- Rule Statement sequence: Garbage collection is performed on two initial configurations, yielding two new configurations that are each equivalent to the initial ones. Thus, the two new configurations are transitively equivalent. The proof is then completed by structural induction over the sequence of statements.

- Rule MethodCall: n pools are created in each case; we extend $\mathcal{J}$ so that it maps the newly constructed pools one by one (and according to construction order). The proof is completed by structural induction over the method's body.

□