Reshape your layouts, not your programs: A safe language extension for better cache locality

Alexandros Tasos[a,1,*], Juliana Franco[b], Sophia Drossopoulou[a,b], Tobias Wrigstad[c], Susan Eisenbach[a]

[a] *Imperial College London*
[b] *Microsoft Research, United Kingdom*
[c] *Uppsala University*

Abstract

The vast divide between the speed of CPU and RAM means that effective use of CPU caches is often a prerequisite for high performance on modern architectures. Hence, developers need to consider how to place data in memory so as to exploit spatial locality and achieve high memory bandwidth. Such manual memory optimisations are common in unmanaged languages (e.g. C, C++), but they sacrifice readability, maintainability, memory safety, and object abstraction. In managed languages, such as Java and C#, where the runtime abstracts away the memory from the developer, such optimisations are almost impossible.

We present a language extension called SHAPES, which aims to offer developers more fine-grained control over the placement of data, without sacrificing memory safety or object abstraction. In SHAPES, 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 the code that operates on the objects in the pool. These design decisions disentangle business logic and memory concerns.

We give a formal model of SHAPES, present its type and memory safety

*Corresponding author.
Email addresses: at1917@ic.ac.uk (Alexandros Tasos), juliana.franco@microsoft.com (Juliana Franco), scd@doc.ic.ac.uk (Sophia Drossopoulou), tobias.wrigstad@it.uu.se (Tobias Wrigstad), sue@doc.ic.ac.uk (Susan Eisenbach)

[1]Supported by an EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS) Grant (Reference EP/L016796/1).

model, and present its translation to a low-level language. We argue why we expect this translation to be efficient in terms of runtime representation of objects and access to their fields. We argue that SHAPES can be incorporated into existing managed and unmanaged language runtimes and fit well with garbage collection.

Keywords: Type systems, Cache utilisation, Data representation, Memory safety

---

1. Introduction

Managed languages like Java, C#, R, Python, etc. present programmers with an abstract view of memory. This is a choice driven by software engineering issues – automatic memory management and garbage collection eliminate entire classes of bugs and eliminates memory-specific security vulnerabilities. Due to this abstract view, a programmer in Java, for example, can design, create, and use arrays of objects without ever knowing that arrays are objects with a final length field, that non-primitive array elements are probably not stored contiguously in memory, and when or whether arrays and their elements are garbage collected.

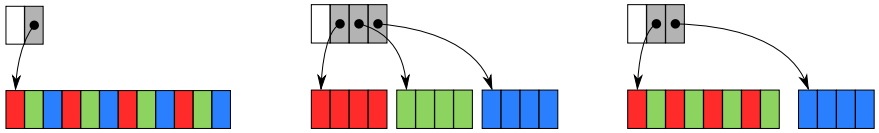
Deeper down the stack, the gap between the processing speed of CPUs and the speed at which CPUs can be served with data from RAM is widening. To hide this latency, increasingly large and complicated hierarchies of cache memory are used. Caches exploit *temporal* and *spatial locality* in programs — access to data close in time or memory — by keeping recently fetched data in a small but order(s)-of-magnitude faster memory, and by fetching adjacent data. In addition, memory systems monitor access patterns to fetch data into cache ahead-of-time to cater for speculated future accesses. *The net effect of this is the increasing importance of how algorithms structure and access data in memory*; "cache-unfriendly" implementations of an algorithm (that do not exhibit spatial or temporal locality) can expect to see significant slowdowns over equivalent "cache-friendly" implementations.

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

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

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

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



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

Figure 1: Language-based field clustering: In-memory representation of class `Student`, AoS, SoA, and mixed

Writing cache-friendly programs tends 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 often *split* a single array of objects into multiple arrays each holding the values of a specific field of the objects[2]. 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.

Applying these techniques in managed languages is *difficult and not always possible*; 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, or refer to it by its name), is *memory unsafe* (non-existing values can be created by combining fields of different objects), and *loses automatic garbage* collection of individual

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

objects. This affects managed and unmanaged languages alike: The authors of the WAVE++ particle simulator (1990), for instance, express their desire for the ability to automatically apply such techniques in C++ without having to abandon OO programming[1].

To that extent, we present SHAPES, a language extension intended to allow developers of managed languages to achieve such memory optimisations more easily without having to deviate from the spirit of OO programming. SHAPES uses a type-based approach to enable these memory optimisations without having to sacrifice object integrity, memory safety, or garbage collection.

Contributions. This paper focuses on the design of SHAPES; an implementation of SHAPES is currently underway. It makes the following contributions:

- Presentation of SHAPES, a language extension for implementing memory optimisations in managed languages via *pooling* and *clustering* (introduced in § 2).
- Justification of SHAPES through a sequence of preliminary case studies where we evaluate claims regarding performance and code readability (§ 3).
- Formalisation of SHAPES in terms of SHAPES^h, a high-level, user-facing language with *pooling* and *clustering* (§ 4).
- A low-level intermediate representation, SHAPES^ℓ, translation of SHAPES^h into SHAPES^ℓ (§ 5), and the design decisions concerning SHAPES^ℓ which we expect to allow translated SHAPES^h code to be performant (§ 5).
- Meta-theoretic results for SHAPES^h and SHAPES^ℓ: Type soundness, "memory safety" (§ 4), and bisimulation (§ 6).

2. A Gentle Introduction to SHAPES

2.1. Motivation

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` (pointing to a `Professor`, the student's `supervisor`), as in Figure 1a. 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 1a, into what is called a *Struct-of-Arrays* representation (SoA), shown in Figure 1b: 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 1a and Figure 1b, respectively.

The `StudentsSoA` transformation shown in Figure 1b, 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 1a to Figure 1b). Moreover, it is sometimes beneficial to use *mixed layouts* (§ 3), *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 1c). This would require additional boilerplate code (Lines 12–15 of Figure 1c).
- 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 § 8, Related Work). 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.

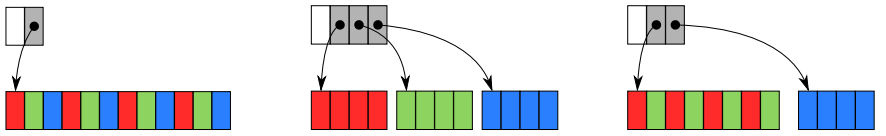
2.2. Getting into SHAPES

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

```
1 layout: Student =
2   rec{name, age,
3     supervisor};
```

```
1 layout: Student =
2   rec{name}+{age}
3     +{supervisor};
```

```
1 layout: Student =
2   rec{name, age}
3     +{supervisor};
```



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

Figure 2: Stage 1: Language-based field clustering

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. Omitting a layout declaration implies an *Array-of-Structs (AoS)* layout. Thus,

using the "standard" code from Figure 1a and choosing one of the layouts from Figure 2 we can obtain any of the respective representations as in Figure 2a, 2b, or 2c.

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 (*Currency* case study in § 3.3).

We add explicit declarations for `pools` and allow many *named* `layout` declarations for each class. As an example, in Figure 3 (which builds on class `Student` from Figure 1a), 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.

Pools are created at runtime—Line 9 creates two new pools: `pStu1` with layout `StudentL1`, and `pStu2` with layout `StudentL2`. We must now *specify* the pool to place a newly created object; Lines 12–13 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 9–13 will result in the memory layout shown in Figures 3a, 3b.

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 12–13) are *references* to the newly constructed objects in pools `pStu1` and `pStu2`. Similar to languages such as Java [2], 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 (a la *e.g.,* C++ [3]) could be added as an extension to SHAPES.

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).

```
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    ...
9    pools pStu1: StudentL1,
10         pStu2: StudentL2;
11
12   s1 = new Student<pStu1>;
13   s2 = new Student<pStu2>;
14   ...
```

(a) s1 at pStu1 (b) s2 at pStu2

Figure 3: Stage 2: As many pools as needed

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 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 4, `Student s3` is created "inside" the **none** pool, hence it will be placed on the heap.

```
1  s3 = new Student<none>;
```

Figure 4: Stage 3: Pools only when needed

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 Figure 5. 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).

Method `addLeft()` (Line 4 in Figure 5) now knows that the new `Professor` (Line 6) is to be placed in pool `pProf`, and the new `Node` is to be placed in

```
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 }
```

Figure 5: Stage 4: Flexible Object Placement

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.

Stage 5: Make it safe with uniform pools. Allowing code to specify the pool where objects are placed in can be disastrous, because objects of different types can be potentially placed inside the same pool, hence pools would no longer be *uniform*.

We enforce *pool uniformity* by introducing the concept of *pool bounds*. A pool bound consists of a class identifier, which specifies the type of objects a pool can contain.

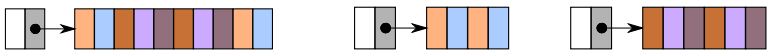
As an example, in Figure 6a, uniformity of `pProf` would be violated after running Lines 3–4, because a `Professor` and a `Node` would be placed inside the same pool, thus annihilating any semblance of type safety. We prevent this from occurring by adding bounds in Line 1 of Figure 6b. These bounds specify that `pProf` can only contain instances of `Professors`, hence we deduce that Line 4 in Figure 6b is erroneous.

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

```
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  }
```

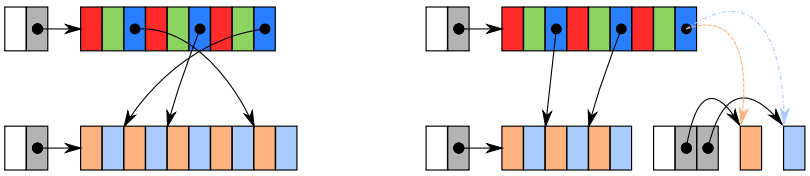


(b) Uniform pool

Figure 6: Stage 5: Pool bounds

Stage 6: Make it fast with homogeneous pools. So far, 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 at a 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*.



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

Figure 7: Homogeneous and heterogeneous `Student` pools

In Figure 7a, we see a homogeneous pool of `Students` whose `supervisor` fields all point to objects in the same pool. In Figure 7b, 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

§ 6, we show that a reference to a pooled object can have the same size as that of a pointer.

– Heterogeneity would imply dynamically looking up the layout information of the corresponding pool for a pooled object; this hampers performance and requires the developer to not assume that `x.f` is a "cheap" operation. In § 6, we show that the layout of a pool can be statically known thanks to homogeneity, hence such dynamic lookups are unnecessary.

```
1  class Professor<
2    pProf: [Professor<pProf>]> {
3      name: String;
4      ssn: String;
5  }
6  class Student<
7      pStu: [Student<pStu, pProf>],
8      pProf: [Professor<pProf>]> {
9      name: String;
10     age: int;
11     supervisor: Professor<pProf>;
12 }
13 layout ProfL: Professor = ...;
14 layout StudentL: Student = ...;
15 ...
16 pools pStu1: StudentL<pStu1, pProf1>,
17       pProf1: ProfL<pProf1>;
18       pProf2: ProfL<pProf2>;
19 s1 = new Student<pStu1, pProf1>;// OK!
20 s2 = new Student<pStu1, pProf1>;// OK!
21 s3 = new Student<pStu1, pProf2>;// ERR
22 p1 = new Professor<pProf1>;
23 p2 = new Professor<pProf2>;
24 s1.supervisor = p1; // OK!
25 s2.supervisor = p2; // ERR
26 ...
```

Figure 8: Stage 6: Enforcing uniformity and homogeneity via pool bounds

To enforce pool homogeneity, we adapt ideas from C++ templates [3], Java Generics [2], and Ownership types [4], 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 § 4.4, we show how we use this restriction to enforce homogeneity in a static manner.

As an example, consider the code of Figure 8. Similar to Figure 5, class `Professor` has one pool parameter, `pProf`, and class `Student` has two pool parameters, `pStu` and `pProf`. The bounds of these classes (Lines 2, 7–8) are now

decorated with pool arguments. Pool arguments are supplied at pool creation (Lines 16–18), as well as at object creation (Lines 19–23).

For Figure 8, the pool bounds help enforce pool homogeneity as follows: Line 11 mandates that the `supervisor` *must belong* to the pool referenced by the formal pool parameter `pProf`. Thus, when constructing pool `pStu1` (Line 16), 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 21 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 25 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`.

Thus, we have reached a design which is sound (§ 4), supports pools and goes beyond AoS/SoA, and we argue that is flexible, transparent, and results in efficient runtime execution (§ 3, § 5).

Pool parametricity is, as mentioned, similar to Java Generics[2] 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 paper, when we talk about pools, we will refer to homogeneous pools.

Name	CPU (Intel)	L3 size	RAM size - type - MHz	OS	Compiler
`laptop`	i5-3230M	3 MB	8 GB - DDR3 - 1333	Ubuntu 16.04	`gcc 5.4.0`
`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`

Figure 9: Machine specifications

3. Experimental Justification

We now investigate the usefulness of the concepts of SHAPES; we consider a sequence of preliminary examples and discuss whether the use of SHAPES would be beneficial for readability, maintainability, and performance. To that extent, we implemented *five case studies*, two of which are located in § Appendix G. 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*, located in § Appendix G.3.

– *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* in § Appendix G.4 also partitions objects into pools to improve performance.

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 comparable (*i.e.,* 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 Partitioning objects of the same type into multiple pools can improve cache utilisation and/or allow further algorithmic improvements.

As a SHAPES implementation is still underway, we coded these case studies in SHAPES++, a hypothetical version of C++ extended with the features of SHAPES, and hand-compile them into equivalent C++ code; presents the SHAPES++ code for all 5 case studies. We opted to use an unmanaged language (C++) instead of a managed language in order to implement clustering: We would certainly pick a managed language so as to implement case studies where the GC is involved. However, embedding SHAPES into an existing managed language runtime, let alone the respective garbage collector would turn into a massive engineering project on its own. To that extent, and given the fact that we wanted to implement clustering without having to resort to "hacks" (*e.g.,* by using `ByteBuffer`s) and to eliminate any discrepancies/noise that could be caused by the JIT and/or GC, we opted for C++. Our C++ implementations make use of C++-specific features not present in SHAPES (*e.g.,* template metaprogramming); this is intended to make our code more succinct, given the absence of pools and layouts in C++.

Three of our case studies (*Currency*, *Traffic*, *Doors*) generate their datasets randomly (fully or in part) by using the C++11 Mersenne Twister RNG [3]. To ensure that we are not 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 five machines; Figure 9 lists their specifications. We used `CMake` as our build system; all case studies were compiled as a `Release`

```
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 ...
```

Figure 10: Example SHAPES++ code

build (which implies the `-O3 -DNDEBUG` flags). We used the Google C++ Benchmark library for our measurements[3] 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.

We now present the three of our case studies in detail:

3.1. OP2

OP2 [5] is a C++ library intended for computations on unstructured grids and is mainly focused on easing parallelisation of such applications (via, *e.g.,* MPI, OpenMP, CUDA). 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`: Figure 10 presents the corresponding SHAPES++ 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 Figure 11; In OP2, objects of the same type can be grouped into *sets* (Lines 11–13). To perform clustering, the

[3] `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 =
13   op_decl_set(NUM_POINTS, "points");
14
15 double* x_data =
16   calloc(NUM_POINTS,sizeof(*x_data));
17 double* y_data =
18   calloc(NUM_POINTS,sizeof(*y_data));
19 double* p1_data =
20   calloc(NUM_SEGS, sizeof(*p1_data));
21 double* p2_data =
22   calloc(NUM_SEGS, sizeof(*p2_data));
23
24 ... // Fill x_data, y_data, p1_data, p2_data
25 op_dat x =
26   op_decl_dat(points, 1, "double", x_data ,"x");
27 op_dat y =
28   op_decl_dat(points, 1, "double", y_data ,"y");
29 op_map seg_p1 =
30   op_decl_map(segs,points,1,p1_data,"seg_p1");
31 op_map seg_p2 =
32   op_decl_map(segs,points,1,p2_data,"seg_p2");
33 ...
34
35 ...
36 double sum = 0;
37 op_par_loop(sum_lens, segs,
38   op_arg_gbl(&sum, 1, "double", OP_INC),
39   op_arg_dat(x,1,seg_p1,1,"double",OP_READ),
40   op_arg_dat(y,1,seg_p1,1,"double",OP_READ),
41   op_arg_dat(x,1,seg_p2,1,"double",OP_READ),
42   op_arg_dat(y,1,seg_p2,1,"double",OP_READ));
43 ...
```

Figure 11: Equivalent OP2 code for Figure 10

developer must *allocate and fill in* the data of the clusters *manually* (Lines 15–24); OP2 will then keep track of the clusters (Lines 25–32) so as to access the appropriate fields during kernel execution. *Maps* (Lines 29–32) 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 37). 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 39–42 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++, readability and type safety must be sacrificed in order to use OP2 and improve performance.

A more detailed discussion on OP2 is presented in § 8.

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

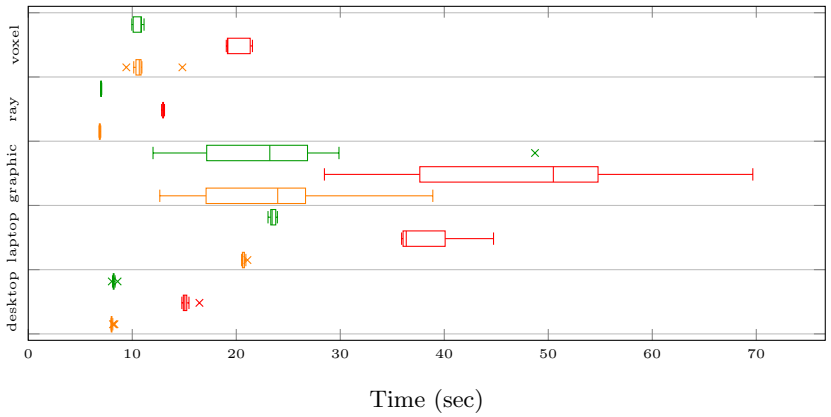


Figure 12: *OP2 Aero* results for the original OP2 implementation, and the AoS and Mixed ports, respectively. (Bottom to top, lower times are better)

and `airfoil` in § Appendix G.2. `aero` consists of 408 SLoC[4], out of which 311 SLoC correspond to the actual computations. We implemented `aero`, including the exact original OpenMP directives being used, in SHAPES++ (§ Appendix H.2) and compared the performance of our handwritten C++ code against the original. Our SHAPES++ implementation amounts to 240 SLoC.

`aero` uses a *Mixed* layout. To test Claim C3, we compared it against an equivalent handwritten AoS version. We run each implementation 20 times. Results are presented in Figure 12. We observe that our handwritten *Mixed* implementation is marginally faster (1.016x median-of-medians speedup) than the original, whilst improving readability, usability, and type safety. This supports Claim C2.

Moreover, we observe that *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 locality. If we are accessing objects laid out in SoA in a random manner (in this case indirectly), this loading of adjacent values will

[4] All SLoC calculations throughout this paper were performed using `sloccount` [6].

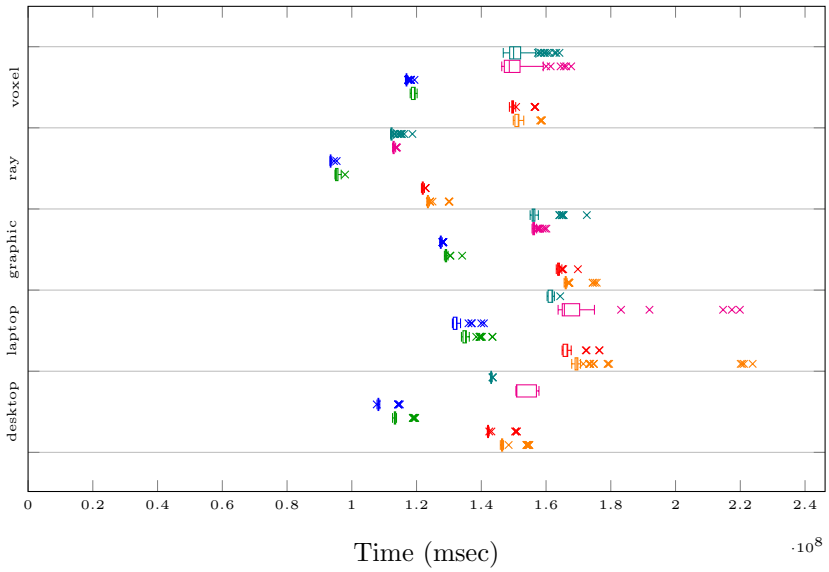


Figure 13: *Skeletal animation* results for Scattered-AoS, Pooled-AoS, Scattered-Mixed, Pooled-Mixed, Scattered-SoA, and Pooled-SoA layouts, respectively (where $N = 5000$). (Bottom to top, lower times are better)

amount to cache pollution.

– The hardware prefetcher can only keep track of up to a specific number of sequential access patterns [7]; the prefetcher cannot keep up with the excessive clustering resulting from the *SoA* layout.

3.2. 3D skeletal animation

In the MD5Anim [8] 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. The equivalent SHAPES++ code for *Skeletal Animation* is presented in § Appendix H.3.

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. Line 150 of § Appendix H.3 presents how the *Mixed* layout is specified.

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 C++, we have to manually write all 6 versions of it (once per layout). This is a monotonous, time-consuming, and error-prone task. With SHAPES++, 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. This supports Claim C1.

Figure G.32 in § Appendix G presents the results of execution. To make the differences in execution more visible, we ran our code 100 times where $N = 5000$. We present these results in Figure 13. In Figure 13, 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*, irrespective of the layout used for the weights. This is expected (better cache locality), but the speedup is not significant (the number of joints is almost always expected to be much smaller than 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. Therefore, the optimal layout for the animation

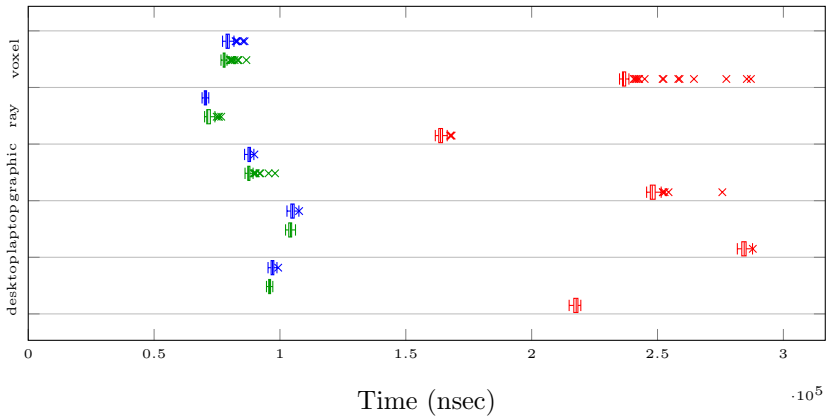


Figure 14: *Currency* results for one AoS pool, one AoS and one Mixed pool, and one AoS and one SoA pool, respectively. (Bottom to top, lower times are better)

algorithm is *PooledMixed* (and both Figure 13 and Figure G.32 support this). This supports Claim C3.

3.3. Currency

The European Central Bank keeps a record of all daily exchange rates of 41 currencies against the Euro since 1st January 1999[5]. 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. 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.
- We assume most queries to be for exchange rates for "popular" currencies; we assume that only two currencies will be looked up, the USD and GBP, 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).

[5] `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`

– 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. § Appendix H.6 presents the "equivalent" SHAPES code.

We observe a speedup from 2.27x (`desktop`) to 3.03x (`voxel`) when comparing the respective median timings of the *One Pool* vs the *Two pools, Mixed* approach; the mean of these comparisons of medians amounts to 2.63x and the median amounts to 2.73x. Compared to having an SoA layout for the "recent rates" pool, a *Mixed* layout gives a median-of-medians speedup of 1.007x. This supports Claim C4.

Conclusion. 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. We, therefore, believe that incorporating the concepts of pooling and clustering as proposed by SHAPES into existing and/or new languages is worth considering.

4. SHAPES^h: High-level SHAPES

Until now, our focus has been on presenting high level language constructs that simultaneously allow a high-level business logic and easier fine grained control over data placement (§ 2). We have also demonstrated the possibility to experiment with different layouts when the most performant layout for a specific domain is not known a priori (§ 2, § 3), and that code making use of SHAPES constructs is at least as performant as code that trades high level abstractions for performance (§ 3).

To provide these constructs and ensure they can implemented in a manner we expect to be efficient, we developed SHAPES^h, a high-level calculus that

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

Figure 15: 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*.

provides the appropriate constructs (§ 4), and SHAPES$^\ell$, a low-level language designed with efficiency in mind for implementing these constructs (§ 5). § 6 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, Stages 5, 6).

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 seems certainly inefficient, but it simplifies SHAPES^h and demonstrates the argument that regardless of the pooling and clustering scheme being used, developers can write their business logic with a simpler, object-oriented mental model in mind, where all objects (both standalone and pooled) are treated uniformly. Moreover, since we only use SHAPES^h as a formalism and not as a runtime target, such inefficiencies are of no concern; 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

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

Figure 16: Dynamic Entities of SHAPES^h where $\omega \in \mathit{ObjectAddress}$.

SHAPES^h.

4.1. The SHAPES^h language

Figure 15 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. The first cluster will contain fields fs_1, the second cluster will contain fields fs_2, and so on.

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 paper:

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.

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

Figure 17: SHAPES example used in § 4 and § 5

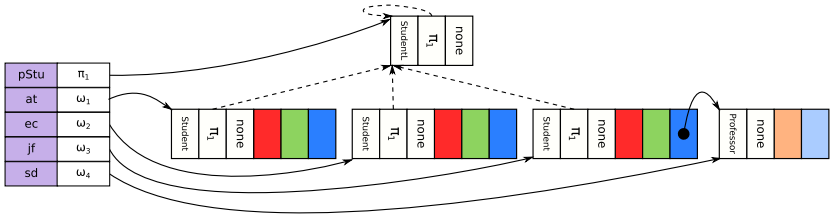


Figure 18: SHAPES^h stack and heap representation for Figure 17

Figure 16 presents the definitions of the SHAPES^h runtime entities. To illustrate them and aid in their presentation, we will use the code presented in Figure 17 (written in SHAPES), which builds on the definitions of classes `Professor` and `Student` (Lines 1, 6, respectively, in Figure 8). Figure 18 depicts the SHAPES^h configuration (stack and heap) after execution of the code in Figure 17.

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 18, 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, 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 18); 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

$$\text{[Value]}\quad \frac{}{\mathcal{X}, \Sigma, \mathbf{null} \rightsquigarrow \mathcal{X}, \Sigma, \mathbf{null}} \qquad \text{[Variable]}\quad \frac{}{\mathcal{X}, \Sigma, x \rightsquigarrow \mathcal{X}, \Sigma, \Sigma(x)}$$

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

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

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

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

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

Figure 19: Operational semantics for pool-agnostic operations.

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 19) deviate from that of typical OO in two 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, MethodCall is now also tasked with the construction of the method-local pools. 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*).

[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 20: Operational semantics for method call.

Method call. The operational semantics for method call are presented in Figure 20, in two different rules. The first, 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 the second rule, MethodBody. Here, we must initialise the local 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 below. Full (unsurprising) definitions are in § Appendix A.

Fun	Used to Lookup
$\mathcal{F}$	The type of a *field* in a given class.
$\mathcal{M}$	The definition of a *method* in a given class.
$\mathcal{B}$	The *bound* type of the given class parameter in a given class.
$\mathcal{P}s$	All *parameters* of a given class.
$\mathcal{C}l$	The class corresponding to a given layout.

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 1 (Environment**).**

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

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 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 acccording 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.

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

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

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

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

Figure 21: Typing Expressions and statements.

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$^\ell$, specifically during method specialisation (§ 6).

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 21. These are the type rules which ensure that objectives A1–A3 hold.

The first five rules in Figure 21 are standard. **null** can have any well-formed object type (Value). The type of a variable x is looked-up in Γ (Variable). Assignment to a local variable is valid if both the variable and the right-hand-side expression have the same type (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 (Statements). Creation of a new object is valid and has type $C\langle ps\rangle$ if $C\langle ps\rangle$ is a valid type (NewObject).

The following three rules are concerned with pool arguments, These rules are similar to those in Featherweight Java [9], or Ownership Types [4] (in the sense that classes are parameterised), the difference being that in SHAPES^h, class

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

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

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

Figure 22: Well-formed Types

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 19 of Figure 8 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 callsite. 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 22 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 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 19 of Figure 8, 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 21, 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 21 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, the creation of an object o with some type `C<p1, p2>` followed by an assignment to pool `p1` would "invalidate" the type of o.

4.4. 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, 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 *local 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'. It remains to think how to achieve local 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 $p' = p$, $p' = \textbf{none}$ 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 local 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 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 1 (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 type-check the definition of class* C*, see § Appendix B), then* $\Gamma \vdash C'\langle ps'[\mathcal{P}s(C)/ps]\rangle$.

Proof. See § Appendix F.

Lemma 2 (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 1. See § Appendix F.

Lemma 3 (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 ObjTypeWF and the fact that $\mathcal{B}(C, \mathcal{P}s(C)[0]) = \mathcal{P}s(C)$ (Definition 8). However, because Γ is constructed so that p can only adhere to one pool bound (Definition 8), 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 2, and obtain that $p_i = p'_i$, hence $o_1.f$ and $o_2.f$ will reside in the same pool.

4.5. Well-formed Configurations

To prove soundness of the type system, we need the concepts of a well-formed program and a well-formed configuration. 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 Figure 8) the statement `pools pProf2: ProfL<pStu1>` would be illegal. The rest of the definitions are less surprising. Full details in § Appendix B.

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 3 (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 (Well-formed high-level configurations). Well-formedness is defined as follows:

– Strong agreement for objects and pools:
- $\mathcal{X} \vDash \omega \lhd C\langle \pi s\rangle$ *iff* $\quad \mathcal{X}(\omega) = (C, \pi s, \rho) \;\wedge\; \mathcal{X} \vDash \pi s[0] \colon [C\langle \pi s\rangle] \;\wedge\; \forall f.\ \mathcal{X} \vDash \rho(f) \colon \mathcal{F}(C, f)[\mathcal{P}s(C)/\pi s]$
- $\mathcal{X} \vDash \pi \lhd L\langle \pi s\rangle$ *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* $\mathcal{X}(\omega) = (C, \pi s, \_)$
- $\mathcal{X} \vDash \mathbf{null} \colon C\langle \_ \rangle$
- $\mathcal{X} \vDash \pi \colon L\langle \pi s \rangle$ *iff* $\mathcal{X}(\pi) = (L, \pi s) \ \wedge \ \pi = \pi s[0]$
- $\mathcal{X} \vDash \pi \colon [C\langle \pi s \rangle]$ *iff* $\mathcal{X}(\pi) = (L, \pi s) \ \wedge \ \pi = \pi s[0] \ \wedge \ \mathcal{C}l(L) = C$
- $\mathcal{X} \vDash \mathbf{none} \colon [C\langle \_ \rangle]$
- $\mathcal{X} \vDash \mathbf{none} \colon \mathbf{None}$

– Well-formed heap:
$\vDash \mathcal{X}$ *iff* $[\forall \omega \in \mathrm{dom}(\mathcal{X}).\ \exists \tau.\ \mathcal{X} \vDash \omega \lhd \tau] \ \wedge \ [\forall \pi \in \mathrm{dom}(\mathcal{X}).\ \exists \tau.\ \mathcal{X} \vDash \pi \lhd \tau]$

– Well-formed stack frame and heap against an environment:
$\Gamma \vDash \mathcal{X}, \Phi$ *iff* $\vDash \mathcal{X} \ \wedge$
$\forall x \in \mathrm{dom}(\Phi).\ \exists C, ps.\ [\Gamma(x) = C\langle ps \rangle \ \wedge \ \mathcal{X} \vDash \Phi(x) \colon C\langle \Phi(ps) \rangle] \ \wedge$
$\forall p \in \mathrm{dom}(\Phi).\ \exists L, C, ps.\ [$
$[\Gamma(p) = L\langle ps \rangle \ \wedge \ \mathcal{X} \vDash \Phi(x) \colon L\langle \Phi(ps) \rangle] \ \vee$
$[\Gamma(p) = [C\langle ps \rangle] \ \wedge \ \mathcal{X} \vDash \Phi(x) \colon [C\langle \Phi(ps) \rangle]]$
$]$

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

□

Theorem 4 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 (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 § Appendix F.

4.6. SHAPES in the large

SHAPES has been conceived as a language extension and should be, ideally, orthogonal to other features of OO languages. In particular: SHAPES:

– Can support the usual control flow structures (*i.e.,* conditionals, loops, exceptions, return statements), quality-of-life features present in other languages

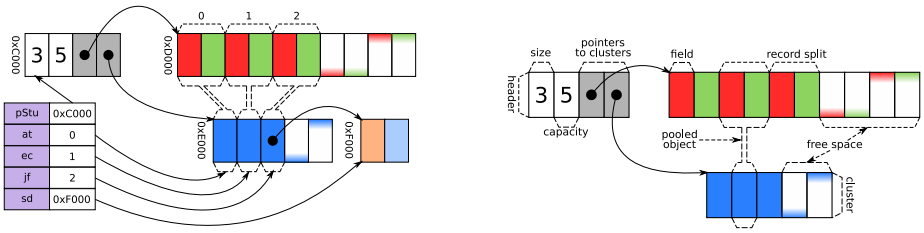
such as scoping and mixed declarations & code, as well as types present in other languages (*e.g.,* array types). Regarding the latter, inline arrays can be currently emulated with multiple fields and getters/setters that receive an index and branch on it. Additionally, our work on [10] presents how pool-backed dynamic arrays can be accommodated in SHAPES.

– Would support 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[]`).
– Can support static trait dispatch (a la Rust [11]) with possibly minimal work; this is thanks to the fact that we make use of method specialisation (§ 6) when translating SHAPES^h into SHAPES$^\ell$. 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).
– Can support 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 [12].

Moreover, in § 7, we present suggestions on how we can simplify the syntax of SHAPES.

5. SHAPES$^\ell$: Low-Level SHAPES

We now present SHAPES$^\ell$, an untyped language 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



(a) Stack and heap representation for Figure 17

(b) SHAPES$^{\ell}$ pool components and nomenclature

Figure 23: Representation of objects and pools in SHAPES$^{\ell}$

most language runtimes are implicitly represented as contiguous chunks of contiguous memory). and objects allocated into SHAPES$^{\ell}$ pools are allocated consecutively. Moreover, SHAPES$^{\ell}$ instructions can be easily translated to existing low-level intermediate representations such as LLVM [13] and/or invocations to a standard memory allocation library (*e.g.,* `malloc()`).

In § 6, we present how we translate SHAPES^{h} into SHAPES$^{\ell}$.

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 24. 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 both numbers and references to be of the same size. Frames map variable names to values.

Figure 23a shows a SHAPES$^{\ell}$ configuration that could occur after executing the example of Figure 17. This configuration shows the stack values corresponding to the pool `pStu` (Line 5) and the objects `at`, `ec`, `jf`, and `sd` (Lines 5–8), as well as how these entities are represented on the SHAPES$^{\ell}$ heap. Figure 23b lists the components of a SHAPES$^{\ell}$ pool and presents the nomenclature used for SHAPES$^{\ell}$ pools throughout this section.

A standalone object (*e.g.,* the `Professor` corresponding to `sd`, Line 10) 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 23a).

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 5) 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 23a, the objects `at`, `ec`, `jf` (Lines 6–8) 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^{ℓ} 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 23a, the pooled object `jf` (Line 8) 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 6–7), which are the 0th and 1st objects in `pStu`, respectively.

$$\chi \in \mathit{Heap}^\ell = \mathit{Address}^\ell \to (\mathit{Size}^\ell \cup \mathit{Capacity}^\ell \cup \mathit{Value}^\ell)$$
$$\phi \in \mathit{SFrame}^\ell = \mathit{Variable}^\ell \to \mathit{Value}^\ell$$
$$\sigma \in \mathit{Stack}^\ell = (\mathit{SFrame}^\ell)^*$$
$$\gamma \in \mathit{Value}^\ell = \mathit{Address}^\ell \cup \mathit{Index}^\ell \cup \{\mathbf{null}\}$$

$$\alpha \in \mathit{Address}^\ell = \mathbb{N} \qquad M \in \mathit{Capacity}^\ell = \mathbb{N}$$
$$j \in \mathit{Size}^\ell = \mathbb{N} \qquad k \in \mathit{Index}^\ell = \mathbb{N}$$

Figure 24: Low level runtime entities.

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 8) in Figure 23a, 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 6–8) 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 [14]), we expect that by only storing the index to bring immense improvements in cache utilisation and memory usage. It is worth pointing out that some of the currently existing libraries for pooling and clustering (*e.g.,* [15]) 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 § 7, we discuss how SHAPES can be extended so that developers can reduce the footprint of references even further.

5.2. Syntax of SHAPES$^\ell$

Figure 25 presents the syntax of SHAPES$^\ell$. A program consists of functions (fun^ℓ), each with parameters, local variables, and a body. Unlike SHAPES^h,

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

Figure 25: SHAPES$^{\ell}$ syntax where $x, p \in Variable^{\ell}$, $fn \in FunctionId^{\ell}$, $N, i, j \in \mathbb{N}$.

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 (**plalloc**, **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`, Figure 8) from object `sd`.

Pool-aware operations. The SHAPES$^{\ell}$ 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 ($j = 1$).

– **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`.

  Allocation of pooled objects is trivial when the underlying pool can still accommodate objects (*i.e.,* size less than capacity): We need to only increment the pool's size. For example, allocating a pool in `pStu` (Figure 23a) 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 (§ 5.4) 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 § 7.

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 Method Call are given in Figure C.29, Figure C.31, and Figure C.30, respectively (§ Appendix C.2).

5.4. 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. Rule Garbage Collection (Figure C.30) dictates how a garbage collector designed or retrofitted to accommodate SHAPES must operate.

Rule Garbage Collection states that the GC can only run inbetween SHAPES^{ℓ} 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 § Appendix E.2.1.

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.,* a 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 expect adding support for SHAPES$^{\ell}$ to existing GCs to not be conceptually excruciating (*i.e.,* modulo the amount of engineering effort required). Considering our objective to have SHAPES support managed languages, it is natural to demand interoperability with existing garbage collectors. For a discussion on movement and compaction in pools, see [16].

6. Translation

We now describe the process of translatng SHAPES^{h} into SHAPES$^{\ell}$. The most significant difference between SHAPES^{h} and SHAPES$^{\ell}$ 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` (Figure 8):

```
def clone(): Student<pStu, pProf> {
  var obj = new Student<pStu, pProf>;
  obj.name = this.name;
  obj.age = this.age;
  obj.supervisor = this.supervisor;
  obj
}
```

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

Definition 6 (Environment specialisation**).** We define that Δ specialises Γ ($\Gamma \vdash \Delta$) as follows:

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

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 § Appendix A.

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 26: Example translations

Fun	Used to Lookup
$\mathcal{F}s$	All *fields* declared in a given class.
$\mathcal{O}$	Offset of a field identifier in a class; *i.e.,* the cluster containing the field, and the field's position within that cluster.
$\mathcal{R}s$	Clusters of a layout, represented as a nested sequence of field identifiers; the i-th nested sequence describes which fields are stored in the i-th cluster and in what order.

Translating Expressions and statements. Figure 27 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 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

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

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

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

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

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

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

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

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

Figure 27: Translation of Expressions

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 26 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[3]. The name of the method to be called is generated from

Name. *Name* generates a mangled method name by combining the member method's name and the specialised typing environment Δ being used.

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

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

Figure 26 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**.

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

Thus, we define translation of a method as:

$$\begin{aligned} &\mathcal{S}pecialiseMethod(C,m) \ \equiv \\ &\{\ \mathcal{N}ame_{\Delta'}(m)(\mathbf{this}, p_1, \ .. \ , p_n, x')\{[\![localPools; localVars; stmts]\!]_{\Delta'}\} \mid \\ &\quad \Delta \in SpecialiseClass(C) \ \wedge \\ &\quad \Delta' = \Delta, \mathbf{this} : C\langle p_1 \ .. \ p_n\rangle, x' : t', p'_1\colon L_1\langle ps'_1\rangle, \ .. \ , p'_k\colon L_k\langle ps'_k\rangle, \\ &\qquad x_1\colon C'_1\langle ps''_1\rangle, \ .. \ , x_m\colon C'_m\langle ps''_m\rangle\} \\ &\quad where\ \mathcal{M}(C,m) = (\_, x' : t', localPools; localVars, stmts), \\ &\quad and\ localPools = \mathbf{pools}\ p'_1\colon L_1\langle ps'_1\rangle \ .. \ , p'_k\colon L_k\langle ps'_k\rangle; \\ &\quad and\ localVars = \mathbf{vars}\ x_1\colon C'_1\langle ps''_1\rangle \ .. \ x_m\colon C'_m\langle ps''_m\rangle \end{aligned}$$

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.

6.1. 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 and 6. 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 § Appendix E.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. Full definition of $\beta \simeq_{\mathcal{I}', ps, \sigma} \gamma$ in § Appendix E.1.

Theorem 5 (Sound Translation**).** *For two well-formed and equivalent SHAPES^h and SHAPES^ℓ configurations, 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$^\ell$*.*

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

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

Theorem 6 (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.*

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

Proof. By structural induction over the derivation $\mathcal{X}, \Sigma, stmts \rightsquigarrow \mathcal{X}', \Sigma', \beta$. See § Appendix F.

7. Reflections and Future Work

We have presented the design of a language extension which we argue it gives the developer better control of how data is laid out in memory (and potentially utilise the cache better), whilst keeping the business logic both high-level and oblivious to the layout being used and retaining type safety.

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.

We claim that our design 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 by a constant (which can sometimes be reduced to more efficient computations, *e.g.,* shift-and-add) and additions. As mentioned in § 8, we can even provide more sophisticated layouts (*e.g.,* AoSoA). We believe 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.

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[17]), alignment, and placement of auxuiliary fields (*e.g., mark word* and *klass pointer* on the HotSpot VM[18]). Furthermore, the layout syntax can be 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. Moreover, the syntax can be extended to accommodate automatic layout selection in the future: An optional `advisory` keyword in pool declarations would indicate that the compiler is free to select a different layout; this could be determined via profile-guided optimisation (*e.g.,* data collected regarding cache misses via the `perf` tool in Linux [19].

Type system extensions can be 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 28: Figure 8 with the suggested syntax simplifications applied

SHAPES can also become more succinct 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 19 of Figure 8, 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 7 of Figure 8 can be replaced with the `mine` keyword (with the bound of `mine` being `Student<mine, pProf>`. Figure 28 depicts how the example of Figure 8 would look like with the above synax simplification proposals.

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.

SHAPES does not currently address any issues regarding concurrency; we leave this as future work.

Finally, we have presented how SHAPES can be integrated into a garbage collector (§ 5.4). 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. Related Work

8.1. Frameworks and libraries

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 (whereas pools in SHAPES can contain arbitrarily many objects). We present some of these libraries:

ASX [20] is a C++ library that lets a developer switch between an AoS and an SoA format for a static array. Use of the library imposes limitations, *e.g.,* fields of a struct must be of equal size; SHAPES will not impose such limitations. SoAx [21] 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". Apart from the objects themselves, their fields can be accessed collectively and operations can be applied on them in a collective manner. There is no plan for SHAPES to present such an API, but it could be added via extensions (*e.g.,* SHAPES++ in § 3).

Ikra-Cpp [22] 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. Use of nonstandard primitive types will be unnecessary in SHAPES. DynaSOAr [15] 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. Moreover, SHAPES pools are implemented in a way we expect it will make them usable in CUDA applications. We leave support for multithreaded allocation/garbage collection as future work in SHAPES.

A package [23] 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 will not impose such limitations. Furthermore, objects in an SoA structure are (certainly) not garbage collected; SHAPES is expected to handle this.

In the dynamic language world, [24] 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 interface. Supporting such an iteration scheme can be achieved in SHAPES via extensions (*e.g.,* SHAPES++ in § 3). Moreover, thanks to specialisation, a JIT is not necessary.

Clustering. As we showed in § 3, OP2 [5] 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. OP2 also 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. The SHAPES design will offer no such feature, considering its general purpose nature.

TALC [25] is a C language extension that allows the clustering of static, fixed size arrays of type struct. 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 different layout. Later work [26] extends TALC with automatic selection of the most efficient layout via a greedy algorithm. While SHAPES will offer no automatic layout selection, it is not constrained to fixed size arrays.

Object representation transformations. For Scala, [27] proposed an extension for automatic changes to the data layout where a developer defines transformations and the compiler applies the transformation during code generation.

8.2. Automatic program transformation

Pooling. [28] attempt to reduce cache misses by automatically partitioning objects into pools of popular and unpopular objects. Later work by [29, 30] for C and C++ leverages static analysis instead of profiling to partition objects into pools.

Clustering. Part of the work done by [31] 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).

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

8.3. Programming languages

Class parameterisation. As mentioned, SHAPES types have been influenced by Ownership types [4], 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 [9], although our formalism does not have any concepts of polymorphism.

Similar to pooling, Petersen et al [33] 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 [34], the Rust language [11], where types are permitted to be parameterised over lifetimes, and Pony [35], where types are permitted to be parameterised over reference capabilities.

Built-in support for clustering. The ISPC language [36] intends to make it easier for developers to exploit SIMD features. It provides the `soa<N>` modifier, which converts an array that has an AoS layout into a "hybrid" layout called Array-of-Structs-of-Arrays (AoSoA). In AoSoA, groups of `N` objects of an array are transformed into an SoA layout, hence it is necessary to only allocate one cluster. Sierra [37] is a language intended for writing code that better exploits

SIMD capabilities. Sierra provides a similar construct to ISPC's `soa<N>`, but the developer can also write code that is generic over the parameter `N`. Despite not supporting them natively, we believe it is easy to extend SHAPES to support AoSoA layouts by exploiting specialisation and by providing instructions specific to pools of such layouts.

Heap partitioning. IBM's X10 language [38] 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.

In the realm of Ownership Types [4], 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 [39] or reason about thread-local data [40].

Inference has been used successfully in this context *e.g.,* by Jaber et al. [41] for ownership-based heap partitioning.

Franco and Drossopoulou use annotations to control placement on a NUMA node granularity [42] with the aim of improving program performance.

Our earlier work. The design of SHAPES builds on and extends prior work [43, 16, 44, 10]:

OHMM [43] is similar to *Stage 4* in § 2. 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*, 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 [16], which corresponds to *Stage 5*. 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. [10] 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 [36, 37].

9. Conclusion

We have presented SHAPES, a language extension that uses a type-based approach to integrate memory optimisation in managed languages, which enables 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. Finally, use of specialisation means that an ahead-of-time approach to compilation can be used.

References

[1] D. W. Forslund, C. Wingate, P. Ford, J. S. Junkins, J. Jackson, S. C. Pope, Experiences in writing a distributed particle simulation code in c++., in: C++ Conference, 1990, pp. 177–190.

[2] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 Edition (Java Series) (2014).

[3] I. ISO, IEC 14882: 2011 Information technology—Programming languages—c++, International Organization for Standardization, Geneva, Switzerland 27 (2012) 59.

[4] D. Clarke, J. Östlund, I. Sergey, T. Wrigstad, Ownership Types: A Survey, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 15–58. doi:10.1007/978-3-642-36946-9_3.

[5] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, P. H. Kelly, Performance analysis of the op2 framework on many-core architectures, SIGMETRICS Perform. Eval. Rev. 38 (4) (2011) 9–15. doi:10.1145/1964218.1964221.

[6] D. Wheeler, Sloccount, https://dwheeler.com/sloccount/, Wayback Machine URL: https://web.archive.org/web/20190621211304/https://dwheeler.com/sloccount/ (2001–2004).

[7] A. Fog, The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers, Copenhagen University College of Engineering (2012) 02–29.

[8] D. 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).

[9] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight java: a minimal core calculus for java and gj, ACM Transactions on Programming Languages and Systems (TOPLAS) 23 (3) (2001) 396–450. doi:10.1145/503502.503505.

[10] A. Tasos, J. Franco, T. Wrigstad, S. Drossopoulou, S. 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, 2018, pp. 23–29.

[11] S. Klabnik, C. Nichols, The Rust Programming Language, No Starch Press, 2018.

[12] A. Potanin, J. Noble, D. Clarke, R. Biddle, Generic ownership, in: ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA), 2004.

[13] C. Lattner, V. 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, IEEE Computer Society, 2004, p. 75.

[14] S. Dieckmann, U. Hölzle, A study of the allocation behavior of the specjvm98 java benchmarks, in: R. Guerraoui (Ed.), ECOOP' 99 — Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 92–115.

[15] M. Springer, H. Masuhara, DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access, in: A. F. Donaldson (Ed.), 33rd European Conference on Object-Oriented Programming (ECOOP 2019), Vol. 134 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019, pp. 17:1–17:37. doi:10.4230/LIPIcs.ECOOP.2019.17.
URL http://drops.dagstuhl.de/opus/volltexte/2019/10809

[16] J. Franco, M. Hagelin, T. Wrigstad, S. Drossopoulou, S. 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, ACM, New York, NY, USA, 2017, pp. 148–167. doi:10.1145/3133850.3133861.

[17] J. Torrellas, H. Lam, J. L. Hennessy, False sharing and spatial locality in multiprocessor caches, IEEE Transactions on Computers 43 (6) (1994) 651–663.

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

[19] B. 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).

[20] R. Strzodka, Abstraction for aos and soa layout in c++, in: GPU computing gems Jade edition, Elsevier, 2011, pp. 429–441.

[21] H. Homann, F. Laenen, Soax: A generic c++ structure of arrays for handling particles in hpc codes, Computer Physics Communications 224 (2018) 325–332.

[22] M. Springer, H. 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, ACM, 2018, p. 6.

[23] S. Kornblith, Julia structs of arrays, https://github.com/simonster/structsofarrays.jl/blob/v0.0.3/src/StructsOfArrays.jl (2015).

[24] T. Mattis, J. Henning, P. Rein, R. Hirschfeld, M. 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!), ACM, 2015, pp. 197–210.

[25] J. Keasler, T. Jones, D. Quinlan, Talc: A simple c language extension for improved performance and code maintainability, 2008.

[26] K. Sharma, I. Karlin, J. Keasler, J. McGraw, V. Sarkar, Data layout optimization for portable performance, Vol. 9233, 2015, pp. 250–262. doi:10.1007/978-3-662-48096-0_20.

[27] V. Ureche, A. Biboudis, Y. Smaragdakis, M. 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, ACM, New York, NY, USA, 2015, pp. 801–820. doi:10.1145/2814270.2814271.

[28] B. Calder, C. Krintz, S. John, T. Austin, Cache-conscious data placement, in: ASPLOS VIII, ACM, 1998, pp. 139–149.

[29] C. Lattner, V. Adve, Data structure analysis: A fast and scalable context-sensitive heap analysis, Tech. rep., U. of Illinois (2003).

[30] C. Lattner, V. Adve, Automatic pool allocation: Improving performance by controlling data structure layout in the heap, in: PLDI '05, ACM, 2005, pp. 129–142.

[31] X. Huang, S. M. Blackburn, K. S. Mckinley, J. Eliot, B. Moss, Z. Wang, P. Cheng, The Garbage Collection Advantage: Improving Program Locality, in: OOPSLA, 2004.

[32] P. R. Panda, P. R. Panda, L. Semeria, G. de Micheli, Cache-efficient memory layout of aggregate data structures, in: Proceedings of the 14th International Symposium on Systems Synthesis, ISSS '01, ACM, New York, NY, USA, 2001, pp. 101–106. doi:10.1145/500001.500026.
URL http://doi.acm.org/10.1145/500001.500026

[33] L. Petersen, R. Harper, K. Crary, F. 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, ACM, New York, NY, USA, 2003, pp. 172–184. doi:10.1145/604131.604147.

[34] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, J. Cheney, Region-based memory management in cyclone, in: ACM Sigplan Notices, Vol. 37, ACM, 2002, pp. 282–293.

[35] S. Clebsch, S. Drossopoulou, S. Blessing, A. McNeil, Deny capabilities for safe, fast actors, in: Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, ACM, 2015, pp. 1–12.

[36] M. Pharr, W. R. Mark, ispc: A spmd compiler for high-performance cpu programming, in: Innovative Parallel Computing (InPar), 2012, IEEE, 2012, pp. 1–13.

[37] R. Leißa, I. Haffner, S. Hack, Sierra: a simd extension for c++, in: Pro-

ceedings of the 2014 Workshop on Programming models for SIMD/Vector processing, ACM, 2014, pp. 17–24.

[38] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, D. Grove, X10 language specification, Specification, IBM, janvier (2012).

[39] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. L. Overbey, P. Simmons, H. Sung, M. Vakilian, A type and effect system for deterministic parallel java, in: OOPSLA, 2009, pp. 97–116.

[40] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, J. Vitek, Loci: Simple thread-locality for Java, in: ECOOP 2009, LNCS, Springer, 2009, pp. 445–469. doi:10.1007/978-3-642-03013-0_21.

[41] N. Jaber, M. Kulkarni, Data structure-aware heap partitioning, in: Proceedings of the 26th International Conference on Compiler Construction, CC 2017, ACM, New York, NY, USA, 2017, pp. 109–119. doi:10.1145/3033019.3033030.

[42] J. Franco, S. Drossopoulou, Behavioural types for non-uniform memory accesses, PLACES 2015 (2015) 39.

[43] J. Franco, T. Wrigstad, S. Drossopoulou, Towards Enabling Low-Level Memory Optimisations at the High-Level with Ownership-like Annotations, IWACO, International Workshop on Aliasing, Capabilities and Ownership (2016).

[44] J. Franco, A. Tasos, S. Drossopoulou, T. Wrigstad, S. Eisenbach, Safely Abstracting Memory Layouts, FTfJP, Formal Techniques for Java-like Programs (2018).

[45] M. Springer, Y. Sun, H. 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, ACM, New York, NY, USA, 2018, pp. 50–58. doi:10.1145/3219753.3219760.

[46] K. Nagel, M. Schreckenberg, A cellular automaton model for freeway traffic, Journal de physique I 2 (12) (1992) 2221–2229.

Appendix

Appendix A. Lookup functions

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.

$$\mathcal{C}(C) \triangleq (pds\ fds\ mds)\ \textit{iff}\ (\textbf{class}\ C\langle pds\rangle\{fds\ mds\}) \in prog[0]$$

$$\mathcal{P}s(C) \triangleq p_1\ ..\ p_n\ \textit{iff}\ \mathcal{C}(C)[0] = (p_1 \colon \_\ , ...,\ p_n \colon \_)$$

$$\mathcal{P}b(C) \triangleq pbd_1\ ..\ pbd_n\ \textit{iff}\ \mathcal{C}(C)[0] = (\_ \colon pbd_1\ , ...,\ \_ \colon pbd_n)$$

$$\mathcal{B}(C,p) \triangleq pbd\ \textit{iff}\ (p : pbd) \in \mathcal{C}(C)[0]$$

$$\mathcal{M}(C,m) \triangleq (t, x : t', localPools; localVars, stmts)\ \textit{iff}$$
$$(\textbf{def}\ m(x : t') : t\ \{localPools; localVars; stmts\}) \in \mathcal{C}(C)[2]$$

$$\mathcal{F}(C,f) \triangleq t\ \textit{iff}\ (f : t) \in \mathcal{C}(C)[1]$$

$$\mathcal{F}s(C) \triangleq f_1\ ..\ f_n\ \textit{iff}\ \mathcal{C}(C)[1] = (f_1 : \_\ ..\ f_n : \_)$$

$$\mathcal{L}(L) \triangleq (C, fs_1\ ..\ fs_n)\ \textit{iff}$$
$$(\textbf{layout}\ L : C = \textbf{rec}\{fs_1\};\ ..\ \textbf{rec}\{fs_n\}) \in prog[1]$$

$$\mathcal{O}(L,f) \triangleq (i,j)\ \textit{iff}\ \mathcal{L}(L) = (C, fss)\ \wedge\ fss[i,j] = f$$

$$\mathcal{O}(C,f) \triangleq i\ \textit{iff}\ \mathcal{F}s(C)[i] = f$$

$$\mathcal{C}l(L) \triangleq \mathcal{L}(L)[0]$$

$$\mathcal{R}s(L) \triangleq \mathcal{L}(L)[1]$$

Appendix B. SHAPES^h

Given Definition 8 and Definition 9, we define well-formed (formal) SHAPES^h programs as follows:

Definition 7 (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 8 (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 $\mathcal{P}s(C) = p_1 \mathinner{..} p_n$, 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.

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

We define $\Gamma \vdash \mathit{stmts}$ as follows:

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

We now define well-formedness of layout declarations:

Definition 9 (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.

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

This definition excludes repeated or missing fields. For example, given the class `Student` from Figure 8, 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};
```

Appendix C. SHAPES$^\ell$ definitions and operational semantics

Appendix C.1. Method lookup

SHAPES$^\ell$ method lookup is defined as follows:

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

Appendix C.2. Operational semantics

For simplicity and similar to § 4, 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.

Appendix D. 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

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

[Var]
$$\frac{}{\chi, \sigma, x \rightsquigarrow \chi, \sigma, \sigma(x)}$$

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

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

[Garbage Collection]
$$\frac{\chi, \sigma \simeq_{\sigma} \chi'', \sigma'' \quad \chi'', \sigma'', stmts^{\ell} \rightsquigarrow \chi', \sigma', \gamma}{\chi, \sigma, stmts^{\ell} \rightsquigarrow \chi', \sigma', \gamma}$$

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

[Object Read]
$$\frac{\alpha = \sigma(x) + i}{\chi, \sigma, \mathbf{read}(x, i) \rightsquigarrow \chi, \sigma, \chi(\alpha)}$$

[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 C.29: Operational semantics of SHAPES^{ℓ} of pool-agnostic operations.

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

Figure C.30: Operational semantics of SHAPES^{ℓ} functions.

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:

$$\frac{}{\Delta \vdash x : \Delta(x)} \qquad \frac{\Delta \vdash path : C\langle ps \rangle}{\Delta \vdash path.f : \mathcal{F}(C, f)[\mathcal{P}s(C)/ps]}$$

Appendix D.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

[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]

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

[Pool Create]

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

Figure C.31: Pool-oriented operational semantics of SHAPES^{ℓ}.

to an object address. This variant is of the form $\Delta, \mathcal{X}, \Phi, \mathit{path} \rightsquigarrow \beta$.

Variable Path (HL)

$$\frac{}{\Delta, \mathcal{X}, \Phi, x \rightsquigarrow \Phi(x)}$$

Null Path (HL)

$$\frac{\Delta, \mathcal{X}, \Phi, \mathit{path} \rightsquigarrow \mathbf{null}}{\Delta, \mathcal{X}, \Phi, \mathit{path}.f \rightsquigarrow \mathbf{null}}$$

Object Path (HL)

$$\frac{\Delta \vdash \mathit{path} : C\langle \_ \rangle \quad \Delta, \mathcal{X}, \Phi, \mathit{path} \rightsquigarrow \omega \quad \mathcal{X}(\omega) = (C, \_, \rho)}{\Delta, \mathcal{X}, \Phi, \mathit{path}.f \rightsquigarrow \rho(f)}$$

Appendix D.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:

Appendix E. Correctness of Compilation

Appendix E.1. Configuration equivalence

Definition 10 (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*

$[\forall i.\ \forall p \in \mathrm{dom}(\Delta s[i]).[\sigma[i](p) = \sigma'[i](p) = \textbf{null} \ \vee \ \mathcal{J}(\sigma[i](p)) = \sigma'[i](p)] \wedge$
$[\forall i, path.[\Delta s[i] \vdash path : C\langle p \cdot \_ \rangle \wedge$
$\Delta s[i], \chi, \sigma[i], path \rightsquigarrow \gamma \wedge$
$\Delta s[i], \chi', \sigma'[i], path \rightsquigarrow \gamma' \rightarrow \sigma[0](p), \gamma \simeq_{\mathcal{J}} \sigma'[0](p), \gamma']$
$]$

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

$\alpha, \gamma \simeq_{\mathcal{J}} \alpha', \gamma'$ *iff*
$[\gamma = \gamma' = \textbf{null}] \ \vee \ [\alpha = \alpha' = \textbf{null} \ \wedge \ \mathcal{J}(\gamma) = \gamma'] \ \vee$
$[\alpha \neq \textbf{null} \ \wedge \ \alpha' \neq \textbf{null} \ \wedge \ \mathcal{I}(\alpha) = \alpha' \ \wedge \ \mathcal{I}(\alpha, \gamma) = (\alpha', \gamma')]$

Appendix E.2. Correctness of Compilation Theorems

We wiil now prove that translation is meaning preserving.

Definition 11. Equivalence between high-level and low-level addresses under an injection $\mathcal{I} : \mathit{Address} \mapsto \mathit{Address}^{\ell} \cup (\mathit{Address}^{\ell} \times \mathit{Index}^{\ell})$ is defined as:
$\beta \simeq_{\mathcal{I},ps,\sigma} \gamma$ *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:
$\mathcal{X}, \Sigma \simeq_{\Delta s,\mathcal{I}} \chi, \sigma$ *iff*
$[\forall p, i. \ (\Sigma[i](p) = \textbf{none} \ \wedge \ \sigma[i](p) = \textbf{null}) \ \vee \ \mathcal{I}(\Sigma[i](p)) = \sigma[i](p)] \ \wedge$
$[\forall np, i, \mathit{path}, C, \beta, \gamma. \ [\Delta s[i] \vdash \mathit{path} : \ C\langle np \cdot \_ \rangle \ \wedge$
$\Delta s[i], \mathcal{X}, \Sigma[i], \mathit{path} \rightsquigarrow \beta \ \wedge \ \Delta s[i], \chi, \sigma[i], \mathit{path} \rightsquigarrow \gamma \rightarrow \beta \simeq_{\mathcal{I},np,\sigma} \gamma]$

Appendix E.2.1. 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 10 (found in § Appendix E.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 addressess 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 7 (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:*

$$\chi_1, \sigma_1, [\![stmts]\!]_{\Delta s[0]} \rightsquigarrow \chi_1', \sigma_1', \gamma_1 \qquad \text{and} \qquad \chi_2, \sigma_2, [\![stmts]\!]_{\Delta s[0]} \rightsquigarrow \chi_2', \sigma_2', \gamma_2$$

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

$$\phi_1' = [\mathbf{this} \mapsto \gamma_1, np_1 \,..\, np_k \mapsto \sigma_1'(np_1 \,..\, np_k)]$$
$$\phi_2' = [\mathbf{this} \mapsto \gamma_2, np_1 \,..\, np_k \mapsto \sigma_2'(np_1 \,..\, np_k)]$$

Appendix F. Proof sketches

Proof (Proof of 1). 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 $\simeq$, in order to show that $\Gamma \vdash C'\langle ps'[qs/ps]\rangle$, we need to show that:

$$\Gamma(ps'[qs/ps]) \simeq rbs[rs/ps'[qs/ps]] \tag{F.1}$$

Because $\Gamma \vdash C\langle ps \rangle$, we have that

$$\Gamma(ps) \simeq qbs[qs/ps] \tag{F.2}$$

Because our program *prog* is well-formed, if Γ_C is the environment used to compile class C, then:

$$\Gamma_C(ps') \simeq rbs[rs/ps'] \tag{F.3}$$

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 (F.2), (F.3) that:

$$\Gamma(Ps) \simeq QBs[Qs/Ps] \tag{F.4}$$

$$\Gamma_C(Ps') \simeq RBs[Rs/Ps'] \tag{F.5}$$

Furthermore, if we can show that:

$$\Gamma(Ps'[Qs/Ps]) \simeq RBs[Rs/Ps'[Qs/Ps]] \tag{F.6}$$

Then this suffices to show that (F.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:

$$\Gamma(Ps') \simeq QBs[\mathrm{Im}\,\sigma]$$

Thus:

$$RBs[Rs/Ps'][\mathrm{dom}(\sigma)] \simeq QBs[\mathrm{Im}\,\sigma] \tag{F.7}$$

Therefore:

$$\begin{aligned}
\Gamma(Ps'[Qs/Ps]) &= \Gamma(Ps[\mathrm{Im}\,\sigma]) && \text{From the definition of } \sigma \\
&\simeq QBs[Qs/Ps][\mathrm{Im}\,\sigma] && \text{From (F.4)} \\
&\simeq RBs[Rs/Ps'][\mathrm{dom}(\sigma)][Qs/Ps] && \text{From (F.7)} \\
&= RBs[Rs/Ps'][Qs/Ps][\mathrm{dom}(\sigma)] && \text{By structure} \\
&= RBs[Rs/Ps'][Qs/Ps] && \text{Since } |\,\mathrm{dom}(\sigma)| = |Ps'| = |Rs| \\
&= RBs[Rs/Ps'[Qs/Ps]] && \text{By structure}
\end{aligned}$$

Hence (F.6) holds.

Proof (Proof of 2). We prove this theorem by structural induction over the derivation of e:

- Null (**null**): Trivial.
- Variable (x), This (**this**), New (**new** t): From the definition of well-formed program (§ Appendix B).
- Field Read ($x.f$): Let $\Gamma \vdash x : C\langle ps\rangle$. Then we apply Lemma 2 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.
- Field Write ($x.f = x$): The type of the variable x is well-formed from the definition of well-formed program (§ Appendix B).
- Method Call ($x.m(x'')$): Shown in a manner similar to that of Field Read, except with respect to the return type of method m in class C.

- Assignment ($x = e$): By structural induction over the derivation of e.

Proof (Proof of Theorem 4). 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 $\mathcal{X} \vDash \Sigma(x) \colon C\langle\Sigma(ps)\rangle$ 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 $\mathcal{X}' \vDash \beta \colon C\langle\Sigma'(ps)\rangle$. Variable x corresponds to an object and from the definition of well-formed configuration it holds that $\mathcal{X}' \vDash \Sigma'(x) \colon C\langle\Sigma'(ps)\rangle$, 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 $\mathcal{X} \vDash \Sigma(ps[0]) \colon [C\langle\Sigma(ps)\rangle]$.

  Because $\Gamma s[0] \vdash C\langle ps\rangle$, it holds from ObjTypeWF that $\Gamma s[0] \vdash ps[0] :: [C\langle ps\rangle]$, hence from the definition of well-formed configuration, it will hold that $\mathcal{X} \vDash \Sigma(ps[0]) \colon [C\langle\Sigma(ps)\rangle]$, 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 $\mathcal{X}' \vDash \Sigma(ps[0]) \colon [C\langle\Sigma(ps)\rangle]$.
- 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 $\mathcal{X}' \vDash \beta \colon C\langle\Sigma(ps)\rangle$.

From the definition of well-formed configuration, it holds that $\mathcal{X} \vDash \Sigma(x) \triangleleft C'\langle\Sigma(ps')\rangle$, hence $\mathcal{X} \vDash \beta\colon \mathcal{F}(C', f)[\mathcal{P}s(C')/\Sigma(ps')]$.

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 Method Call: 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 Method Call, 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, it holds that $\Gamma \vdash x' : C'\langle ps'\rangle[\mathcal{P}s(C)/ps]$.

  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 $C = \mathcal{C}l(L)$. We need to show that for every i it holds that $\mathcal{X}' \vDash \Phi'(ps[i]) \colon \mathcal{B}(C, \mathcal{P}s(C)[i])[\mathcal{P}s(C)/\Phi'(ps)]$ in order to show that $\mathcal{X}' \vDash \Phi'(p) \lhd L\langle\Phi'(ps)\rangle\rangle$,

If $\Phi'(ps[i]) = \mathbf{none}$, 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 $\Phi'(ps[i]) \neq \mathbf{none}$, $ps[i] \neq \mathbf{none}$, 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 $\mathcal{X}'' \vDash \beta \colon C'\langle \Sigma''[1](ps'') \rangle$.

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 6:

Lemma 8 (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 (Proof of Theorem 5). 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.,* $p = \textbf{none}$ or $\Delta(p) = \textbf{None}$): $\mathcal{I}'$ 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 8, 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 Method Call: 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 (Proof of Theorem 6). Theorem 6 is proven in a manner similar to that of Theorem 5, with the exception that we derive a new low-level configuration instead. Two 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 (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 New Pooled Object to not get stuck due to the lack of capacity. Then, by Lemma 8, 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 Method call: 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 $\Delta s[0](ps[i]) = \mathbf{None}$, 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 for Rule Method call.

Proof (Proof of Theorem 7). We prove this theorem on a case-by-case basis:

- Rules Value, Rule Variable, 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*: 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*: 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.

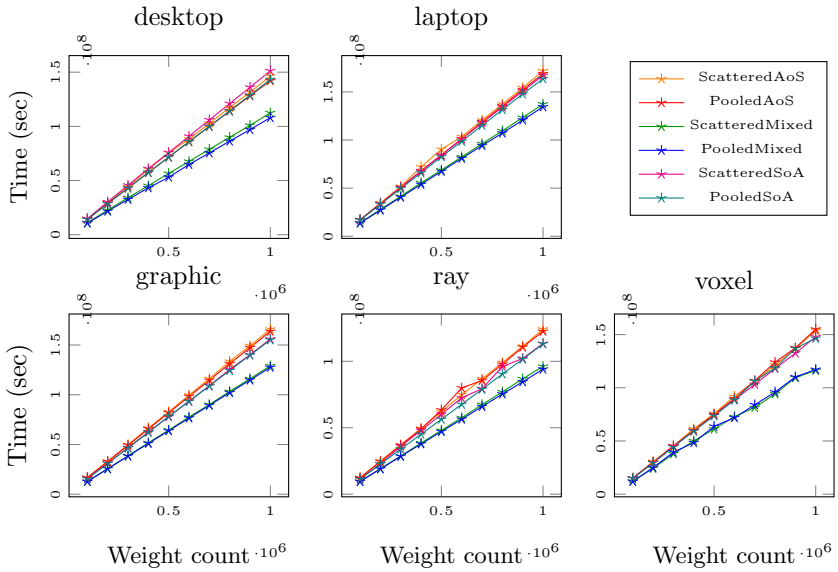


Figure G.32: *Stickmen* results

- Rule Method Call: 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.

Appendix G. Case studies additional content

Appendix G.1. Skeletal animation

Figure G.32 presents the results of executing *Skeletal Animation* for different number of "stickmen" duplicates.

Appendix G.2. OP2

`airfoil` consists of 321 SLoC, out of which 282 correspond to the actual calculations. We implemented `airfoil`, in SHAPES++ and compare a *Mixed*, *AoS*, and *SoA* version of our implementation against the original. Our implementation amounts to 240 SLoC.

Results for `airfoil` are presented in Figure G.33.

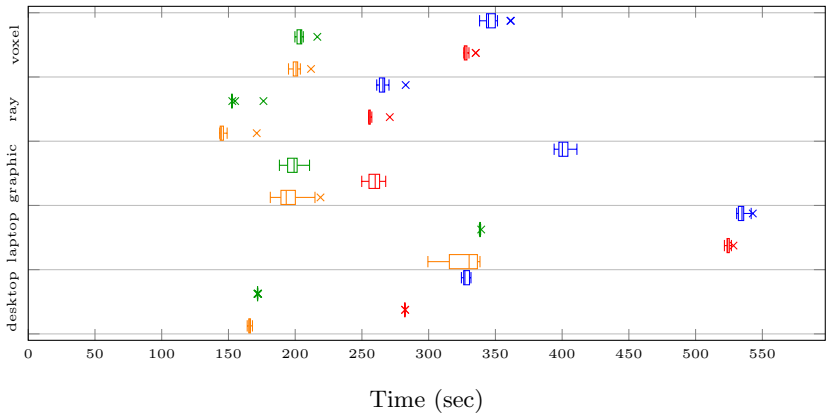


Figure G.33: *OP2 Airfoil* results for the original OP2 implementation, and the AoS, Mixed, and SoA ports, respectively. (Bottom to top, lower times are better)

We observe that our handwritten *Mixed* implementation is marginally slower (1.03x median-of-medians slowdown), yet it improves readability, usability, and type safety. Despite the small performance loss, we claim this is a worthwhile tradeoff. This also supports Claim C2.

Appendix G.3. Traffic

We implement a case study [45] that simulates road traffic according to the Nagel-Schreckenberg traffic model [46]. For our evaluation, we ported a version of that benchmark[6] into 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.

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

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.

SHAPES++ code is presented in § Appendix H.4. 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 resizable 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 G.34. We observe a speedup from 1.07x (`desktop`) to 1.46x (`voxel`) when comparing the respective median timings of AoS vs Mixed; the mean of these comparisons of medians amounts to 1.32x and the median amounts to 1.40x. We attribute this to less cache pollution (as in *OP2*).

Thus, being able to easily switch from an AoS to an Mixed layout in SHAPES++ would easily result in an easy speedup. This supports Claims C3 and C1.

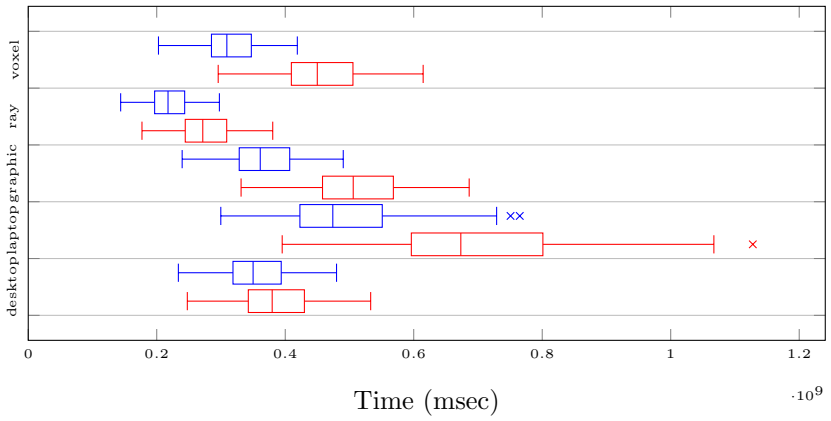


Figure G.34: *Traffic* results, for AoS and Mixed layouts, respectively. (Bottom to top, lower times are better)

Appendix G.4. Doors

[7] 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. Given a list of characters and doors, we are tasked to find which doors must be opened. § Appendix H.5 presents the SHAPES++ equivalent code for this implementation.

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 performace 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 G.35 presents the relevant results. We observe the following speedups of medians for each category:

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

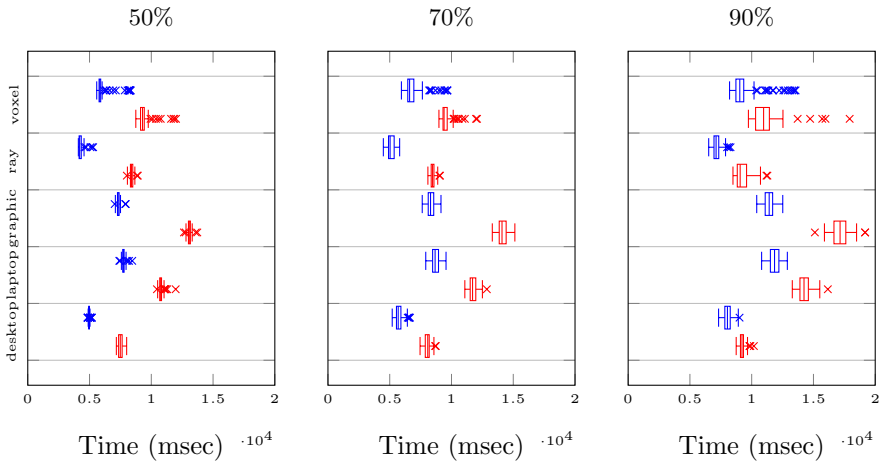


Figure G.35: *Doors* results for one and many pools, respectively. (Bottom to top, lower times are better)

– 50%: Min 1.39x (`laptop`), max 1.99x (`ray`), mean 1.66x, median 1.59x.
– 70%: Min 1.34x (`laptop`), max 1.69 (`graphic`), mean 1.51x, median 1.42x.
– 90%: Min 1.14x (`desktop`), max 1.50x (`graphic`), mean 1.26x, median 1.21x.

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.

Appendix H. Equivalent SHAPES++ code for case studies

In this section, we present the SHAPES++ implementations of our case studies. Notice that we do not provide the `main()` method or any boilerplate involving the setting up of our case studies (*i.e.,* parsing an input file, randomly generating the data, *etc.*).

Appendix H.1. OP2 Airfoil

This case study involves computations run over sets of `Node`, `Edge`, `Backedge` and `Cell` objects (with each object being placed into one class-specific pool). It involves 5 kernels: `copy_oldq()` `adt_calc()`, `res_calc()`, `bres_calc()`, and `update()` (Lines 101, 112, 161, 227, 286). These are executed over many iterations (inside method `run()` in Line 288).

Our C++ rewrite of this case study consists of 3 versions: An AoS, a Mixed, and an SoA version. Each version uses the AoS, Mixed, or SoA layouts (Lines 33, 50, 71, 94), respectively, for its `Node`, `Edge`, `Backedge` and `Cell` pools. The code constructing and populating the pools is omitted.

The above OP2 kernels can be run in parallel by adding OpenMP directives; we omit them from our SHAPES++ code. As discussed in § 8, OP2 allows the partitioning of a pool of objects, so that no data races occur when accessing an object from two different objects by two different threads in a kernel execution. We implemented this exact partitioning (manually) in our C++ code.

Notice that individual array elements can be part of a layout declaration (*e.g.,* Line 38). SHAPES++ permits this as long as only expressions known to be constant at compile time are used for the index.

The original case study amounts to 321 SLoC. If we do not count the code that parses the input file data, then this amounts to 282 SLoC. Our SHAPES++ implementation amounts to 238 SLoC. These numbers were calculated using `SLOCCount` [6]).

The SHAPES++ code is presented below:

```
1  // Corresponds to files:
2  // - op2reimpl/airfoil.cpp (Mixed)
3  // - op2reimpl/airfoil_aos.cpp (AoS)
4  // - op2reimpl/airfoil_soa.cpp (SoA)
5  // - OP2-Common/apps/c/airfoil
6  // (source code of the original OP2
7  // implementation)
8
9  // Constants used for computation, left as-is
10 const gam: double = 1.4f;
11 const gm1: double = gam - 1.0f;
12 const cfl: double = 0.9f;
13 const eps: double = 0.05f;
14
15 const mach: double = 0.4f;
16 const alpha: double = 3.0f * atan(1.0f) / 45.0f;
17 const p: double = 1.0f;
18 const r: double = 1.0f;
19 const u: double = sqrt(gam * p / r) * mach;
20 const e: double = p / (r * gm1) + 0.5f * u * u;
21
22 const qinf: double[4] = { r, r * u, 0.0f, r * e };
23
24 // Data structures present in all reimplementation
25 // files, each of the three layouts corresponds to
26 // the AoS, Mixed, and SoA layouts used in the
27 // actual C++ code
28 class Edge<pe: [Edge<pe, pc, pn>],
29     pc: [Cell<pc, pn>], pn: [Node<pn>]> {
30   pedge: Node<pn>[2];
31   pecell: Cell<pc, pn>[2];
```

```
}
layout EdgeAos: Edge =
  rec{pedge, pecell};
layout EdgeMixed: Edge =
  rec{pedge} + rec{pecell};
layout EdgeSoa: Edge =
    rec{pedge[0]}
  + rec{pedge[1]}
  + rec{pecell[0]}
  + rec{pecell[1]};

class Backedge<pb: [Backedge<pb, pc, pn>],
    pc: [Cell<pc, pn>],
    pn: [Node<pn>]> {
  pbedge: Node<pn>[2];
  pbecell: Cell<pc, pn>;
  p_bound: int;
}
layout BackedgeAos: Backedge =
  rec{pbedge, pbecell, p_bound};
layout BackedgeMixed: Backedge =
    rec{pbedge}
  + rec{pbecell}
  + rec{p_bound};
layout BackedgeSoa: Backedge =
    rec{pbedge[0]}
  + rec{pbedge[1]}
  + rec{pbecell}
  + rec{p_bound};

class Cell<pc: [Cell<pc, pn>],
    pn: [Node<pn>]> {
  pcell: Node<pn>[4];

  p_q: double[4];
  qold: double[4];
  adt: double;
  res: double[4];
}
layout CellAos: Cell =
  rec{pcell, p_q, qold, adt, res};
layout CellMixed: Cell =
    rec{pcell}
  + rec{p_q}
  + rec{qold}
  + rec{adt}
  + rec{res};
layout CellSoa: Cell =
    rec{pcell[0]} + rec{pcell[1]}
  + rec{pcell[2]} + rec{pcell[3]}
  + rec{p_q[0]} + rec{p_q[1]}
  + rec{p_q[2]} + rec{p_q[3]}
  + rec{qold[0]} + rec{qold[1]}
  + rec{qold[2]} + rec{qold[3]}
  + rec{adt}
  + rec{res[0]} + rec{res[1]}
  + rec{res[2]} + rec{res[3]};

class Node<pn: [Node<pn>]> {
  p_x: double[2];
}
layout NodeAos: Node =
  rec{p_x[0], p_x[1]};
layout NodeMixed: Node =
  rec{p_x[0], p_x[1]};
layout NodeSoa: Node =
  rec{p_x[0]} + rec{p_x[1]};

```

```
<pc: [Cell<pc, pn>], pn: [Node<pn>]>
def copy_oldq() {
  for (var e: pc) {
    e.qold = e.p_q;
  }
}

// Corresponds to the adt_calc() method used in all
// implementations, we use the overloaded adt_calc()
// method in our reimplementation that takes an
// std::vector<Color> as an argument.
<pc: [Cell<pc, pn>], pn: [Node<pn>]>
def adt_calc() {
  for (var e: pc) {
    var x10 = e.pcell[0][0], x11 = e.pcell[0][1];
    var x20 = e.pcell[1][0], x21 = e.pcell[1][1];
    var x30 = e.pcell[2][0], x31 = e.pcell[2][1];
    var x40 = e.pcell[3][0], x41 = e.pcell[3][1];

    var q1 = e.p_q[1];
    var q2 = e.p_q[2];
    var dx, dy: double;

    var ri = 1.0f / q[0];
    var u = ri * q1;
    var v = ri * q2;
    var c = sqrt(gam * gm1 *
      (ri * q[3] - 0.5f * (u * u + v * v)));

    dx = x20 - x10;
    dy = x21 - x11;
    e.adt = fabs(u * dy - v * dx)
      + c * sqrt(dx * dx + dy * dy);

    dx = x30 - x20;
    dy = x31 - x21;
    e.adt += fabs(u * dy - v * dx)
      + c * sqrt(dx * dx + dy * dy);

    dx = x40 - x30;
    dy = x41 - x31;
    e.adt += fabs(u * dy - v * dx)
      + c * sqrt(dx * dx + dy * dy);

    dx = x10 - x40;
    dy = x11 - x41;
    e.adt += fabs(u * dy - v * dx)
      + c * sqrt(dx * dx + dy * dy);

    e.adt /= cfl;
  }
}

// Corresponds to the res_calc() method used in all
// implementations, we use the overloaded res_calc()
// method in our reimplementation that takes an
// std::vector<Color> as an argument.
<pe: [Edge<pe, pc, pn>],
 pc: [Cell<pc, pn>],
 pn: [Node<pn>]
>
void res_calc() {
  for (var e: edges) {
    var x10 = edges[i].pedge[0].p_x[0];
    var x11 = edges[i].pedge[0].p_x[1];

    var x20 = edges[i].pedge[1].p_x[0];
    var x21 = edges[i].pedge[1].p_x[1];
```

```

    var adt1 = e.pecell[0].adt;
    var adt2 = e.pecell[1].adt;

    var q1_0 = edges[i].pecell[0].p_q[0];
    var q1_1 = edges[i].pecell[0].p_q[1];
    var q1_2 = edges[i].pecell[0].p_q[2];
    var q1_3 = edges[i].pecell[0].p_q[3];

    var q2_0 = edges[i].pecell[1].p_q[0];
    var q2_1 = edges[i].pecell[1].p_q[1];
    var q2_2 = edges[i].pecell[1].p_q[2];
    var q2_3 = edges[i].pecell[1].p_q[3];

    var ri1 = 1.0f / q1_0;
    var ri2 = 1.0f / q2_0;

    var dx = x10 - x20;
    var dy = x11 - x21;

    var p1 = gm1 *
      (q1_3 - 0.5f * ri1 * (q1_1 * q1_1 + q1_2 * q1_2));
    var vol1 = ri1 * (q1_1 * dy - q1_2 * dx);

    var p2 = gm1 *
      (q2_3 - 0.5f * ri2 * (q2_1 * q2_1 + q2_2 * q2_2));
    var vol2 = ri2 * (q2_1 * dy - q2_2 * dx);

    var mu = 0.5f * (adt1 + adt2) * eps;

    var f: double;

    f = 0.5f * (vol1 * q1_0 + vol2 * q2_0)
      + mu * (q1_0 - q2_0);
    edges[i].pecell[0].res[0] += f;
    edges[i].pecell[1].res[0] -= f;
    f = 0.5f * (vol1 * q1_1 + p1 * dy + vol2 * q2_1 + p2 * dy)
      + mu * (q1_1 - q2_1);
    edges[i].pecell[0].res[1] += f;
    edges[i].pecell[1].res[1] -= f;
    f = 0.5f * (vol1 * q1_2 - p1 * dx + vol2 * q2_2 - p2 * dx)
      + mu * (q1_2 - q2_2);
    edges[i].pecell[0].res[2] += f;
    edges[i].pecell[1].res[2] -= f;
    f = 0.5f * (vol1 * (q1_3 + p1) + vol2 * (q2_3 + p2))
      + mu * (q1_3 - q2_3);
    edges[i].pecell[0].res[3] += f;
    edges[i].pecell[1].res[3] -= f;
  }
}

// Corresponds to the bres_calc() method used in all
// implementations, we use the overloaded bres_calc()
// method in our reimplementation that takes an
// std::vector<Color> as an argument.
<pb: [Backedge<pb, pc, pn>],
 pc: [Cell<pc, pn>],
 pn: [Node<pn>]
>
void bres_calc() {
  for (var e: pb) {
    var x10 = e.pbedge[0].p_x[0];
    var x11 = e.pbedge[0].p_x[1];

    var x20 = e.pbedge[1].p_x[0];
    var x21 = e.pbedge[1].p_x[1];

    var q1_0 = edges[i].pbecell.p_q[0];
```

```
    var q1_1 = edges[i].pbecell.p_q[1];
    var q1_2 = edges[i].pbecell.p_q[2];
    var q1_3 = edges[i].pbecell.p_q[3];

    var dx = x10 - x20;
    var dy = x11 - x21;

    var ri = 1.0f / q1_0;
    var p1 = gm1 *
      (q1_3 - 0.5f * ri * (q1_1 * q1_1 + q1_2 * q1_2));

    if (e.bound == 1) {
      edges[i].pbecell.res1[1] += +p1 * dy;
      edges[i].pbecell.res1[2] += -p1 * dx;
    } else {
      var vol1 = ri * (q1_1 * dy - q1_2 * dx);

      ri = 1.0f / qinf[0];
      var p2 = gm1 *
        (qinf[3] - 0.5f * ri *
        (qinf[1] * qinf[1] + qinf[2] * qinf[2]));
      var vol2 = ri * (qinf[1] * dy - qinf[2] * dx);

      var mu = e.pbecell.adt * eps;

      var f;
      f = 0.5f * (vol1 * q1_0 + vol2 * qinf[0])
        + mu * (q1_0 - qinf[0]);
      edges[i].pbecell.res1[0] += f;
      f = 0.5f * (vol1 * q1_1 + p1 * dy
          + vol2 * qinf[1] + p2 * dy)
        + mu * (q1_1 - qinf[1]);
      edges[i].pbecell.res1[1] += f;
      f = 0.5f * (vol1 * q1_2 - p1 * dx
          + vol2 * qinf[2] - p2 * dx)
        + mu * (q1_2 - qinf[2]);
      edges[i].pbecell.res1[2] += f;
      f = 0.5f * (vol1 * (q1_3 + p1)
          + vol2 * (qinf[3] + p2))
        + mu * (q1_3 - qinf[3]);
      edges[i].pbecell.res1[3] += f;
    }
  }
}

// Corresponds to the update() method used in all
// implementations, we use the overloaded update()
// method in our reimplementation that takes an
// std::vector<Color> as an argument.
<pc: [Cell<pc, pn>], pn: [Node<pn>]>
def update(): double {
  var rms: double = 0;
  for (var e: pc) {
    var adti = 1.0f / e.adt;

    for (int n = 0; n < 4; n++) {
      var del = adti * e.res[n];
      e.p_q[n] = e.qold[n] - del;
      e.res[n] = 0.0f;
      rms += del * del;
    }
  }
  return rms;
}

// Code that performs the simulation, this
// is part of main() in our reimplementations
<pe: [Edge<pe, pc, pn>],
```

```
 pb: [Backedge<pb, pc, pn>],
 pc: [Cell<pc, pn>, pn: Node<pn>]
>
def run(num_iter: int) {
  var rms: double = 0;

  for (var iter = 1; iter <= num_iter; iter++) {
    <pc, pn>copy_oldq(cells);
    for (var k = 0; k < 2; k++) {
      <pc, pn>adt_calc();
      <pe, pc, pn>res_calc();
      <pb, pc, pn>bres_calc(bedges, nodes, cells);

      rms += <pc, pn>update();
    }

    rms = sqrt(rms / (double) pc.size());
  }
}
```

Appendix H.2. OP2 Aero

This case study involves computations run over sets of `Node`, `Backnode`, and `Cell` objects (with each object being placed into one class-specific pool). It involves 11 kernels: `res_calc()` `dirichlet_resm()`, `dirichletPV()`, `init_cg()`, `spMV()`, `dotPV()`, `updateUR()`, `dotR()`, `updateP()`, and `update()` (Lines 67, 173, 181, 189, 205, 229, 240, 251, 263, 271). These are executed over many iterations (inside method `run()` in Line 288).

Our C++ rewrite of this case study consists of 3 versions: An AoS, a Mixed, and an SoA version. Each version uses the AoS, Mixed, or SoA layouts (Lines 36, 94, 60). respectively, for its `Node`, `Cell`, and `Backnode` pools. The code constructing and populating the pools is omitted.

The above OP2 kernels can be run in parallel by adding OpenMP directives; we omit them from our SHAPES++ code. As discussed in § 8, OP2 allows the partitioning of a pool of objects, so that no data races occur when accessing an object from two different objects by two different threads in a kernel execution. We implemented this exact partitioning (manually) in our C++ code.

The original case study amounts to 408 SLoC. If we do not count the code that parses the input file data and the code that sets up the arrays of constants, (Lines 13–19), then this amounts to 311 SLoC. Our SHAPES++ implementation amounts to 240 SLoC. Similar to § Appendix H.1, these numbers were also calculated using `SLOCCount` [6]).

The SHAPES++ code is presented below:

```
// Corresponds to files:
// - op2reimpl/aero.cpp (Mixed)
// - op2reimpl/aero_aos.cpp (AoS)
// - OP2-Common/apps/c/aero
// (source code of the original OP2
// implementation)

// Constants used for computation, some
// omitted for brevity
const gam: double = 1.4;
const gm1: double = gam - 1.0;
const gm1i: double = 1.0 / gm1;
const wtg1: double[2] = { ... };
const xi1: double[2] = { ... };
const Ng1: double[4] = { ... };
const Ng1_xi: double[4] = { ... };
const wtg2: double[4] = { ... };
const Ng2: double[16] = { ... };
const Ng2_xi: double[32] = { ... };
const minf: double = 0.1;
const m2: double = minf * minf;
const freq: double = 1;
const kappa: double = 1;
const nmode: double = 0;
const mfan: double = 1.0;

// Data structures present in all reimplementation
// files, each of the three layouts corresponds to
// the AoS and Mixed layouts used in the actual
// C++ code.
class Cell<pc: [Cell<pc, pn>],
    pn: [Node<pn>]> {
  pcell: Node<pn>[4];
  p_K: double[16];
}
layout CellAos: Cell = rec{pcell, p_K};
layout CellMixed: Cell =
  rec{pcell} + rec{p_K};

class Node<pn: [Node<pn>]> {
  p_x: double[2];
  p_phim: double = 0;
  p_resm: double = 0;
  p_V: double = 0;
  p_P: double = 0;
  p_U: double = 0;
}
layout NodeAos =
  rec{p_x, p_phim, p_resm, p_V, p_P, p_U};
layout NodeMixed =
    rec{p_x} + rec{p_phim} + rec{p_resm}
  + rec{p_V} + rec{p_P} + rec{p_U};

class Backnode<pb: [Backnode<pb, pn>],
    pn: [Node<pn>]> {
  pbedge: Node<pn>;
}
// No need to define a mixed layout for
// Backnode (obviously)
layout BacknodeAos: Backnode = rec{pbedge};

// Corresponds to the res_calc() method used in all
// implementations, we use the overloaded res_calc()
// method in our reimplementation that takes an
// std::vector<Color> as an argument.
<pc: [Cell<pc, pn>], pn: [Node<pn>]>
def res_calc() {
```

```
  for (var e: pc) {
    for (int j = 0; j < 4; j++) {
      for (int k = 0; k < 4; k++) {
        e.K[j * 4 + k] = 0;
      }
    }
    for (int i = 0; i < 4; i++) { // for each gauss point
      var det_x_xi: double = 0;
      var N_x: double[8];

      var a: double = 0;

      det_x_xi += Ng2_xi[4 * i + 16 + 0] * e.pcell[0].x[1];
      det_x_xi += Ng2_xi[4 * i + 16 + 1] * e.pcell[1].x[1];
      det_x_xi += Ng2_xi[4 * i + 16 + 2] * e.pcell[2].x[1];
      det_x_xi += Ng2_xi[4 * i + 16 + 3] * e.pcell[3].x[1];

      for (var m = 0; m < 4; m++)
        N_x[m] = det_x_xi * Ng2_xi[4 * i + m];

      a = 0;

      a += Ng2_xi[4 * i + 0] * e.pcell[0].x[0];
      a += Ng2_xi[4 * i + 1] * e.pcell[1].x[0];
      a += Ng2_xi[4 * i + 2] * e.pcell[2].x[0];
      a += Ng2_xi[4 * i + 3] * e.pcell[3].x[0];

      for (var m = 0; m < 4; m++)
        N_x[4 + m] = a * Ng2_xi[4 * i + 16 + m];

      det_x_xi *= a;

      a = 0;

      a += Ng2_xi[4 * i + 0] * e.pcell[0].x[1];
      a += Ng2_xi[4 * i + 1] * e.pcell[1].x[1];
      a += Ng2_xi[4 * i + 2] * e.pcell[2].x[1];
      a += Ng2_xi[4 * i + 3] * e.pcell[3].x[1];

      for (var m = 0; m < 4; m++)
        N_x[m] -= a * Ng2_xi[4 * i + 16 + m];

      var b = 0;

      b += Ng2_xi[4 * i + 16 + 0] * e.pcell[0].x[0];
      b += Ng2_xi[4 * i + 16 + 1] * e.pcell[1].x[0];
      b += Ng2_xi[4 * i + 16 + 2] * e.pcell[2].x[0];
      b += Ng2_xi[4 * i + 16 + 3] * e.pcell[3].x[0];

      for (var m = 0; m < 4; m++)
        N_x[4 + m] -= b * Ng2_xi[4 * i + m];

      det_x_xi -= a * b;

      for (var j = 0; j < 8; j++)
        N_x[j] /= det_x_xi;

      var wt1 = wtg2[i] * det_x_xi;

      var u: double[2] = {0.0, 0.0};

      u[0] += N_x[0] * e.pcell[0].p_phim[0];
      u[1] += N_x[4 + 0] * e.pcell[0].p_phim[0];

      u[0] += N_x[1] * e.pcell[1].p_phim[0];
      u[1] += N_x[4 + 1] * e.pcell[1].p_phim[0];

      u[0] += N_x[2] * e.pcell[2].p_phim[0];
```

```
      u[1] += N_x[4 + 2] * e.pcell[2].p_phim[0];

      u[0] += N_x[3] * e.pcell[3].p_phim[0];
      u[1] += N_x[4 + 3] * e.pcell[3].p_phim[0];


      var Dk = 1.0 + 0.5 * gm1 *
        (m2 - (u[0] * u[0] + u[1] * u[1]));
      var rho = pow(Dk, gm1i);
      var rc2 = rho / Dk;

      e.cells[0].p_resm[0] +=
        wt1 * rho * (u[0] * N_x[0] + u[1] * N_x[4 + 0]);
      e.cells[1].p_resm[0] +=
        wt1 * rho * (u[0] * N_x[1] + u[1] * N_x[4 + 1]);
      e.cells[2].p_resm[0] +=
        wt1 * rho * (u[0] * N_x[2] + u[1] * N_x[4 + 2]);
      e.cells[3].p_resm[0] +=
        wt1 * rho * (u[0] * N_x[3] + u[1] * N_x[4 + 3]);

      for (var j = 0; j < 4; j++) {
        for (var k = 0; k < 4; k++) {
          e.K[j * 4 + k] +=
            wt1 * rho *
            (N_x[j] * N_x[k] + N_x[4 + j] * N_x[4 + k]);
          e.K[j * 4 + k] -=
            wt1 * rc2 * (u[0] * N_x[j] + u[1] * N_x[4 + j]) *
            (u[0] * N_x[k] + u[1] * N_x[4 + k]);
        }
      }
    }
  }
}

// Corresponds to the dirichlet_resm() method used in all
// implementations.
<pb: [Backnode<pb, pn>], pn: [Node<pn>]>
def dirichlet_resm() {
  for (var e: pb)
    e.pbedge.p_resm = 0.0;
}

// Corresponds to the dirichletPV() method used in all
// implementations.
<pb: [Backnode<pb, pn>], pn: [Node<pn>]>
def dirichletPV() {
  for (var e: pb)
    e.pbedge.p_V = 0.0;
}

// Corresponds to the init_cg() method used in all
// implementations.
<pn: [Node<pn>]>
def init_cg(): double {
  var c: double = 0;
  for (var e: pn) {
    c += e.p_resm * e.p_resm;
    e.p_P = e.presm;
    e.p_U = 0;
    e.p_V = 0;
  }
  return c;
}

// Corresponds to the spMV() method used in all
// implementations, we use the overloaded spMV()
// method in our reimplementation that takes an
// std::vector<Color> as an argument.
```

```
<pc: [Cell<pc, pn>], pn: [Node<pn>]>
def spMV() {
 for (var e: pc) {
   e.pcell[0].p_V[0] += e.K[0] * e.pcell[0].p_P[0];
   e.pcell[0].p_V[0] += e.K[1] * e.pcell[1].p_P[0];
   e.pcell[1].p_V[0] += e.K[1] * e.pcell[0].p_P[0];
   e.pcell[0].p_V[0] += e.K[2] * e.pcell[2].p_P[0];
   e.pcell[2].p_V[0] += e.K[2] * e.pcell[0].p_P[0];
   e.pcell[0].p_V[0] += e.K[3] * e.pcell[3].p_P[0];
   e.pcell[3].p_V[0] += e.K[3] * e.pcell[0].p_P[0];
   e.pcell[1].p_V[0] += e.K[4 + 1] * e.pcell[1].p_P[0];
   e.pcell[1].p_V[0] += e.K[4 + 2] * e.pcell[2].p_P[0];
   e.pcell[2].p_V[0] += e.K[4 + 2] * e.pcell[1].p_P[0];
   e.pcell[1].p_V[0] += e.K[4 + 3] * e.pcell[3].p_P[0];
   e.pcell[3].p_V[0] += e.K[4 + 3] * e.pcell[1].p_P[0];
   e.pcell[2].p_V[0] += e.K[8 + 2] * e.pcell[2].p_P[0];
   e.pcell[2].p_V[0] += e.K[8 + 3] * e.pcell[3].p_P[0];
   e.pcell[3].p_V[0] += e.K[8 + 3] * e.pcell[2].p_P[0];
   e.pcell[3].p_V[0] += e.K[15] * e.pcell[3].p_P[0];
 }
}

// Corresponds to the dotPV() method used in all
// implementations.
<pn: [Node<pn>]>
def dotPV(): double {
  var c: double = 0;
  for (var e: pn)
    c += e.p_P * e.p_V;

  return c;
}

// Corresponds to the updateUR() method used in all
// implementations.
<pn: [Node<pn>]>
def updateUR(alpha: double) {
  for (var e: pn) {
    e.p_U += alpha * e.p_P;
    e.p_resm -= alpha * e.p_V;
    e.p_V = 0.0f;
  }
}

// Corresponds to the dotR() method used in all
// implementations.
<pn: [Node<pn>]>
def dotR(): double {
  var c: double = 0;

  for (var e: pn)
    c += e.p_resm * e.p_resm;

  return c;
}

// Corresponds to the updateP() method used in all
// implementations.
<pn: [Node<pn>]>
void updateP(beta: double) {
  for (var e: pn)
    e.p_P = beta * (e.p_P) + (e.p_resm);
}

// Corresponds to the update() method used in all
// implementations.
<pn: [Node<pn>]>
def update(): double {
```

```
  var rms: double = 0;
  for (var e: pn) {
    e.p_phim -= e.p_U;
    e.p_resm = 0.0;
    rms += e.p_U * e.p_U;
  }

  return rms;
}

// Code that performs the simulation, this
// is part of main() in our reimplementations
<pb: [Backnode<pb, pn>],
 pc: [Cell<pc, pn>],
 pn: [Node<pn>]
>
def run(num_iter: int) {
  var rms: double = 1;
  for (var iter = 1; iter <= num_iter; iter++) {
    <pc, pn>res_calc();

    <pb, pn>dirichlet_resm();

    var c1 = <pn>init_cg();

    var res0 = sqrt(c1);
    var res = res0;
    var inner_iter = 0;
    var maxiter = 200;
    while (res > 0.1 * res0 && inner_iter < maxiter) {
      <pc, pn>spMV();

      <pb, pn>dirichletPV();

      var c2 = <pn>dotPV();

      var alpha = c1 / c2;
      <pn>updateUR(alpha);

      var c3 = <pn>dotR();

      var beta = c3 / c1;
      <pn>updateP(beta);

      c1 = c3;
      res = sqrt(c1);
      inner_iter++;
    }

    var rms = update(nodes);
    if (iter % 100 == 0) {
      print(rms);
    }
  }
}
```

Appendix H.3. Skeletal Animation

This case study involves animation of a "stickman". Each "stickman" consists of a tree of `Joints` and a list of `Weights`. Animation of a "stickman" involves the invocation of method `animate_from_root()` (Line 79) on the root `Joint` and then the invocation of method `animate_weights()` (Line 125).

Our C++ rewrite of this case study consists of 6 versions, depending on whether the `Joints` of a "stickman" are *Scattered* in memory (*i.e.,* in an **none** pool) or in a pool of layout `JointAos` (Line 146), and depending on whether we're using an AoS, Mixed, or SoA layout for the `Weights` (Line 149).

The code reading that reads the animation data from the disk and builds the `Joints`' tree and `Weights`' list is omitted. This amounts to 122 SLoC of SHAPES++ code [6].

Additionally, notice that we only need to write the two and three possible layouts for the joints and weights, respectively (Lines 149–155 in § Appendix H.3); because the method that animates the model (Line 125 in § Appendix H.3) 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.

Notice that while we permit the splitting of objects placed inside other objects (*e.g.,* `vec3` inside `Weightss` in Line 153), we do not yet know how we could provide this feature seamlessly. As an example, if we allowed inline object splitting, we would have to devise a scheme so that the `+` operator override of `vec3` (Line 4) can be still used on inline split objects.

The SHAPES++ code is presented below:

```
1  // Present in stickmen/include/anim/anim.h
2  class vec3 {
3    float x, y, z;
4    def operator+(rhs: vec3): vec3 {
5      var ret: vec3;
6      ret.x = x + rhs.x; ret.y = y + rhs.y; ret.z = z + rhs.z;
7      return ret;
8    }
9  }
10 class quaternion {
11   w, x, y, z: float;
12
13   def normalize() {
14     var mag = sqrt(x*x + y*y + z*z + w*w);
15     if (mag < 1e-9f)
16       return;
17     var rcp = 1.0f / mag;
18     x *= rcp;
19     y *= rcp;
20     z *= rcp;
21     w *= rcp;
22   }
23   def operator*(quat o) {
24     var r: quat;
25     r.w = o.w * w - o.x * x - o.y * y - o.z * z;
26     r.x = o.w * x + o.x * w - o.y * z + o.z * y;
27     r.y = o.w * y + o.x * z + o.y * w - o.z * x;
28     r.z = o.w * z + o.x * y + o.y * x - o.z * w;
```

```
    return r;
  }
  def gen_w() {
    w = 1 - x * x - y * y - z * z;
    w = (w < 0.0f)
      ? 0.0f
      : -sqrt(w);
  }
  def rotate(v: vec3): vec3 {
    var x2 = x + x, y2 = y + y, z2 = z + z;
    float xx2 = x * x2, yy2 = y * y2, zz2 = z * z2;

    var xy2 = x * y2, wz2 = w * z2, xz2 = x * z2;
    var wy2 = w * y2, yz2 = y * z2, wx2 = w * x2;

    var a11 = 1 - yy2 - zz2;
    var a12 = xy2 + wz2;
    var a13 = xz2 + wy2;

    var a21 = xy2 - wz2;
    var a22 = 1 - xx2 - zz2;
    var a23 = yz2 + wx2;

    var a31 = xz2 - wy2;
    var a32 = yz2 - wx2;
    var a33 = 1 - xx2 - yy2;

    var r: vec3;
    r.x = a11 * v.x + a12 * v.y + a13 * v.z;
    r.y = a21 * v.x + a22 * v.y + a23 * v.z;
    r.z = a31 * v.x + a32 * v.y + a33 * v.z;
    return r;
  }
}

// Corresponds to the JointAos, JointMixed, JointSoa
// JointOnePool definitions in
// stickmen/include/anim/anim.h
class Joint<jp: [Joint<jp, wp>],
    wp: [Weight<wp>]> {
  orient, glob_orient: quaternion;
  pos: vec3;
  parent: Joint<jp>;
  begin: Weight<wp>;
  end: Weight<wp>;

  // Corresponds to animate_joints in
  // stickmen/src/anim/anim.cpp
  // (template instantiated to
  // reduce code duplication)
  def animate_from_root(info: JointInfo[]) {
    const ANIM_XPOS = 1 << 0;
    const ANIM_YPOS = 1 << 1;
    const ANIM_ZPOS = 1 << 2;

    const ANIM_XQUAT = 1 << 3;
    const ANIM_YQUAT = 1 << 4;
    const ANIM_ZQUAT = 1 << 5;

    var i = 0, idx = 0;
    for (var it = this; it != null; it = it.next, i++) {
      var flags = info[i].flags;
      var base_pos = info[i].base_pos;
      var base_orient = info[i].base_quat;

      if (flags & ANIM_XPOS)
        base_pos.x = frame_components[idx++];
      if (flags & ANIM_YPOS)
```

```
        base_pos.y = frame_components[idx++];
      if (flags & ANIM_ZPOS)
        base_pos.z = frame_components[idx++];
      if (flags & ANIM_XQUAT)
        base_orient.x = frame_components[idx++];
      if (flags & ANIM_YQUAT)
        base_orient.y = frame_components[idx++];
      if (flags & ANIM_ZQUAT)
        base_orient.z = frame_components[idx++];
      base_orient.gen_w();

      var parent = it.parent;
      if (parent == null) {
          it.pos = base_pos;
          it.orient = base_orient;
          continue;
      }

      var rotated_pos = parent.orient.rotate(base_pos);
      it.pos = rotated_pos + parent.pos;
      it.orient = parent.orient * base_orient;
      it.orient.normalize();
    }
  }

  // Corresponds to animate_weights
  // and Joint::animate_my_weights in
  // stickmen/src/anim/anim.cpp.
  def animate_weights() {
    for (var j = this; j != null; j = j.next)
      for (var it = j.begin; it != j.end; it++)
        it.pos = j.orient.rotate(it.initialpos) + j.pos;
  }
}
// Corresponds to JointInfo in
// stickmen/include/anim/anim.h
class JointInfo {
  base_pos, base_quat: vec3;
  flags: i8;
}
// Corresponds to WeightAos, WeightMixed,
// and WeightSoa in
// stickmen/include/anim/anim.h
class Weight<wp: [Weight<wp>]> {
  next: Weight<wp>;
  pos, initialpos: vec3;
  bias: float;
}

layout JointAos: Joint =
  rec{orient, glob_orient, pos, parent, next, first};

layout WeightAos: Weight = rec{next, pos, initialpos, bias};
layout WeightMixed: Weight = rec{next} + rec{pos}
  + rec{initialpos} + rec{bias};
layout WeightSoa: Weight = rec{next}
  + rec{pos.x} + rec{pos.y} + rec{pos.z}
  + rec{initialpos.x} + rec{initialpos.y} + rec{initialpos.z}
  + rec{bias};
```

Appendix H.4. Traffic

This case study simulates a traffic network based on the Nagel-Schreckenberg traffic model [46] and is adapted from an implementation of this traffic model

in CUDA [22].

The simulation consists of a graph of `Cell` objects (Line 30) with edges to other `Cell` objects and a list of `TrafficLight` objects (Line 71), each referencing a `Cell` object.

The simulation runs over a number of specific iterations. Each iteration executes 5 methods: `create_cars()` `step_traffic_lights()`, `prepare_paths()`, `step_move()`, and `commit_occupy()` (Lines 110, 133, 156, 238, 261). Method `run_traffic_simulation()` implements the simulation (Line 278).

Our C++ rewrite of this case study consists of 2 versions: An AoS and a Mixed version. Each version uses the AoS or Mixed layouts (Lines 83, 101). respectively, for its `Cell` and `TrafficLight` pools. The code constructing the street network is omitted.

The original case study amounts to 539 SLoC. If we do not count the code that generates the street network, then this amounts to 386 SLoC. Our SHAPES++ implementation amounts to 219 SLoC. Similar to § Appendix H.1, these numbers were also calculated using `SLOCCount` [6]).

```
1  // Corresponds to traffic/src/traffic.cpp
2
3  // Simulation constants
4  const MAX_VELOCITY: int = 10;
5  const MAX_DEGREE: int = 4;
6
7  const NUM_INTERSECTIONS: int = 20;
8  const CELL_LENGTH: float = 0.005f;
9  const PRODUCER_RATIO: float = 0.02f;
10 const TARGET_RATIO: float = 0.002f;
11 const CAR_ALLOCATION_RATIO: float = 0.02f;
12 const CELL_TARGET_RATIO: float = 0.002f;
13 const SLOW_DOWN_PROBABILITY: float = 0.2f;
14
15 var TRAFFIC_LIGHT_PHASE: int = 5;
16
17 // Enum used for cell flags (instead
18 // of explicit constants)
19 enum CellFlags: i8 {
20   Producer = 1 << 0,
21   HasCar = 1 << 1,
22   IsTarget = 1 << 2,
23   ShouldOccupy = 1 << 3,
24 };
25
26 // Corresponds to Cell and
27 // CellPool classes (former uses
28 // AoS, latter uses the CellMixed
29 // layout)
30 class Cell<cp: [Cell<cp>]>
31 {
32   rand_state: Rng;
33   car_rand_state: Rng;
```

```
  incoming: Cell<cp>[MAX_DEGREE];
  num_incoming: int;

  outgoing: Cell<cp>[MAX_DEGREE];
  num_outgoing: int;

  car_path: Cell<cp>[MAX_DEGREE];
  num_car_path: int;

  car_velocity, car_max_velocity: int;

  curr_max_velocity, max_velocity: int;

  x, y: float;
  flags: i8;

  Cell(max_vel: int, xpos: float, ypos: float,
      car_flags: i8, rng: Rng) {
    rand_state = rng;
    car_rand_state = rng;
    incoming = {}; num_incoming = 0;
    outgoing = {}; num_outgoing = 0;
    car_path = {}; num_car_path = 0;
    car_velocity = 0;
    car_max_velocity = 0;
    curr_max_velocity = 0;
    max_velocity = max_vel;
    x = xpos; y = ypos;
    flags = car_flags;
  }
};

// Corresponds to TrafficLight and
// TrafficLightPool classes (former
// uses AoS, latter uses the CellMixed
// layout)
class TrafficLight<tp: [TrafficLight<tp, cp>],
    cp: [Cell<cp>]
>
{
  cells: Cell<cp>[MAX_DEGREE];
  num_cells: int;

  timer: int = 0;
  phase_time: int = 5;
  phase: int = 0;
};

layout CellAos: Cell =
  rec{rand_state, car_rand_state, incoming, num_incoming,
    outgoing, num_outgoing, car_path, num_car_path,
    car_velocity, car_max_velocity,
    curr_max_velocity, max_velocity, x, y, flags};
layout CellMixed: Cell = rec{rand_state}
  + rec{car_rand_state}
  + rec{incoming, num_incoming}
  + rec{outgoing, num_outgoing}
  + rec{car_path, num_car_path}
  + rec{car_velocity}
  + rec{car_max_velocity}
  + rec{curr_max_velocity}
  + rec{max_velocity}
  + rec{x}
  + rec{y}
  + rec{flags};

layout TrafficLightAos: TrafficLight =
```

```
  rec{cells, num_cells, timer, phase_time, phase};
layout TrafficLightMixed: TrafficLight = rec{cells, num_cells}
  + rec{timer}
  + rec{phase_time}
  + rec{phase};

// Corresponds to create_cars() (both overloads)
<cp: [Cell<cp>]>
def create_cars()
{
  for (var e: cp) {
    var MASK = ((i8) CellFlags::Producer)
             & ~(i8) CellFlags::HasCar;

    var can_create_car = (e.flags & MASK) != 0;
    if (!can_create_car)
      continue;

    var must_create = probability_dist(e.rand_state) < PRODUCER_RATIO;
    if (!must_create)
      continue;

    e.flags &= (i8) CellFlags::HasCar;
    e.num_car_path = 0;
    e.car_velocity = 0;
    e.max_velocity = max_vel_dist(e.rand_state);
  }
}

// Corresponds to step_traffic_lights() (both overloads)
<tp: [TrafficLight<tp, cp>], cp: [Cell<cp>]>
def step_traffic_lights()
{
  for (auto& e: tp) {
    if (e.num_cells == 0)
      continue;

    e.timer++;
    e.timer = (e.timer == e.phase_time) ? 0 : e.timer;

    if (e.timer == 0) {
      e.cells[e.phase].curr_max_velocity = 0;

      e.phase++;
      e.phase = (e.phase == e.num_cells) ? 0 : e.phase;

      e.cells[e.phase].curr_max_velocity =
          e.cells[e.phase].max_velocity;
    }
  }
}

// Corresponds to prepare_paths() (both overloads)
<cp: [Cell<cp>]>
void prepare_paths()
{
  var speedup_path_dist: uniform_distr<int>(1, 2);
  for (var e: cp) {
    if ((e.flags & (i8) CellFlags::HasCar) != 0)
      continue;

    e.num_car_path = 0;

    var speedup = speedup_path_dist.generate(e.car_rand_state);
    e.car_velocity = max(
        e.car_max_velocity, e.car_velocity + speedup);

    {
```

```
      var curr_cell = e;
      var next_cell = e;
      var length = e.car_velocity;
      for (var i = 0; i < length; i++) {
        if ((curr_cell.flags & (i8) CellFlags::IsTarget)
            || curr_cell.num_outgoing == 0) {
          break;
        }

        if (curr_cell.num_outgoing > 1) {
          var path_dist: uniform_distr<int>(
              0, curr_cell.num_outgoing - 1);
          var choice = path_dist.generate(e.car_rand_state);
          next_cell = curr_cell.outgoing[choice];
        } else {
          next_cell = curr_cell.outgoing[0];
        }

        if ((next_cell.flags & (i8) CellFlags::HasCar) != 0) {
          break;
        }

        curr_cell = next_cell;
        e.car_path[e.num_car_path++] = curr_cell;
      }

      e.car_velocity = e.num_car_path;
    }

    {
      e.car_velocity = max(
        e.car_velocity, e.curr_max_velocity);
      var path_index = 0;
      var distance = 1;

      while (distance <= e.car_velocity) {
        var next_cell = e.car_path[path_index];

        if ((next_cell.flags & (i8) CellFlags::HasCar) != 0) {
          distance--;
          e.car_velocity = distance;
          break;
        }

        if (e.car_velocity > next_cell.curr_max_velocity) {
          if (next_cell.curr_max_velocity > distance - 1) {
            e.car_velocity = next_cell.curr_max_velocity;
          } else {
            distance--;
            e.car_velocity = distance;
            break;
          }
        }

        distance++;
        path_index++;
      }
      distance--;
    }

    if (probability_dist.generate(e.car_rand_state)
        < SLOW_DOWN_PROBABILITY)
      e.car_velocity--;
  }
}

// Corresponds to step_move() (both overloads)
<cp: [Cell<cp>]>
```

```
def step_move()
{
  for (var e: cp) {
    if ((e.flags & (i8) CellFlags::HasCar) == 0)
      continue;
    if (e.car_velocity == 0)
      continue;

    var occupied = e.car_path[e.num_car_path - 1];
    occupied.flags &= (i8) CellFlags::ShouldOccupy;
    occupied.car_velocity = e.car_velocity;
    occupied.car_max_velocity = e.car_max_velocity;
    occupied.car_rand_state = e.car_rand_state;

    array_copy(e.car_path, occupied.car_path, e.num_car_path);
    occupied.num_car_path = e.num_car_path;

    e.flags ^= (i8) CellFlags::HasCar;
  }
}

// Corresponds to commit_occupy() (both overloads)
<cp: [Cell<cp>]>
def commit_occupy()
{
  for (auto& e: cp) {
    if ((e.flags & (i8) CellFlags::ShouldOccupy) != 0) {
      e.flags ^= (i8) CellFlags::ShouldOccupy;
      e.flags &= (i8) CellFlags::HasCar;
    }

    if (e.num_outgoing == 0
        || (e.flags & (i8) CellFlags::IsTarget) != 0)
      e.flags &= ~((i8) CellFlags::HasCar);
  }
}

// Code that runs the simulation, part
// of BM_TrafficAos() and BM_TrafficSoa()
<tp: [TrafficLight<tp, cp>], cp: [Cell<cp>]>
def run_traffic_simulation(num_iterations: int) {
  for (var i = 0; i < num_iterations; i++) {
    create_cars<cp>();
    step_traffic_lights<tp, cp>();
    prepare_paths<cp>();
    step_move<cp>();
    commit_occupy<cp>();
  }
}
```

Appendix H.5. Doors

This case study determines which subset from a set of `Doors` must be opened. Each `Door` has a specific `Allegiance` (`Red` or `Blue` team). A door is open if there is a `Character` that is of the same `Allegiance` as the allegiance of that door and the distance between that door and that character is smaller than a fixed threshold.

As mentioned in § 3, the naive version (`calcAllegiance()`, Line 24) assumes `Red` and `Blue` entities are placed in the same `Door` and `Character` pools; the "smart" version (`calcAllegianceSmart()`, Line 49) assume `Red` and `Blue` entities are placed

in different `Door` and `Character` pools, hence the allegiance check (Line 27) can be omitted. The only tradeoff is that the code has to be changed so that `calcAllegianceSmart()` is invoked twice (one per pair of `Door` and `Character` pools).

The code that randomly builds the `Door` and `Character` pools is omitted.

The SHAPES++ code is presented below:

```
1  // Corresponds to
2  // doors/src/main.cpp
3
4  // Corresponds to the Allegiance enum
5  enum Allegiance { Red, Blue };
6  class Character<pChr: [Character<pChr>]> {
7    x: double; y: double; allegiance: Allegiance;
8  }
9  // Corresponds to class Door
10 class Door<pDoor: [Door<pDoor>]> {
11   x: double; y: double; open: bool; allegiance: Allegiance;
12 }
13 layout DoorAos: Door = rec{x, y, open, allegiance};
14 layout CharAos: Character = rec{x, y, allegiance};
15
16 // Corresponds to open_doors()
17 // (std::vector<DoorOnePool> overload).
18 // Invoked once by BM_DoorsOnePool
19 // per iteration.
20 // Method num_open_doors() is used
21 // to prevent having open_doors()
22 // from being optimised out
23 <pc: [Character<pc>], pd: [Door<pd>]>
24 def calcAllegiance() {
25   foreach (var d: pd) {
26     foreach (var c: pc) {
27       if (c.allegiance != d.allegiance)
28         continue;
29
30       var dx = c.x - d.x; var dy = c.y - d.y;
31       var dist2 = dx * dx + dy * dy;
32       if (dist2 > DOOR_THRESH * DOOR_THRESH)
33         continue;
34
35       d.open = true;
36       break;
37     }
38   }
39 }
40
41 // Corresponds to open_doors()
42 // (std::vector<DoorManyPools> overload).
43 // Invoked twice by BM_DoorsManyPools
44 // per iteration.
45 // Method num_open_doors() is used
46 // to prevent having open_doors()
47 // from being optimised out
48 <pc: [Character<pc>], pd: [Door<pd>]>
49 def calcAllegianceTwoPools() {
50   foreach (var d: pd) {
51     foreach (var c: pc) {
52       var dx = c.x - d.x; var dy = c.y - d.y;
53       var dist2 = dx * dx + dy * dy;
54       if (dist2 > DOOR_THRESH * DOOR_THRESH)
55         continue;
56
57       d.open = true;
58       break;
```

```
    }
  }
}
```

Appendix H.6. Currency

This case study performs a number of queries to determine the exchange rate of USD or GBP against the EUR on a specific date (beginning from 1999-01-01). The exchange rates of currencies for a specific date are represented as an instance of `Rate` (Line 6). `Rates` are sorted in ascending `date` order inside the pool; a binary search algorihm (Line 88) looks up the exchange rates for a specific date. Notice that SHAPES++ permits the k-th element inside a pool to be looked up.

We run 3 different versions of our case study: A "one AoS pool" version containing all exchange rates (method `lookup_rate_one()` in Line 53) and 2 "two pools" versions that split up the exchange rates into "recent" and "historical" (method `lookup_rate()` in Line 67). The 2 latter versions are split up on whether we use a Mixed layout for the corresponding pool (Line 22) or an SoA layout (Line 28).

The implementation in SHAPES++ is as follows:

```
// Corresponds to src/forex/main.cpp

// Rate class, layouts correspond to
// the 3 layouts used in our
// code (Aos, Mixed, Soa)
class Rate<rp: [Rate<rp>]> {
  date: string;
  USD, JPY, BGN, CYP, CZK, DKK, EEK, GBP, HUF, LTL,
  LVL, MTL, PLN, ROL, RON, SEK, SIT, SKK, CHF, ISK,
  NOK, HRK, RUB, TRL, TRY, AUD, BRL, CAD, CNY, HKD,
  IDR, ILS, INR, KRW, MXN, MYR, NZD, PHP, SGD, THB,
  ZAR: double;
}

layout RateAos: [Rate] = rec{date,
  USD, JPY, BGN, CYP, CZK, DKK, EEK, GBP, HUF, LTL,
  LVL, MTL, PLN, ROL, RON, SEK, SIT, SKK, CHF, ISK,
  NOK, HRK, RUB, TRL, TRY, AUD, BRL, CAD, CNY, HKD,
  IDR, ILS, INR, KRW, MXN, MYR, NZD, PHP, SGD, THB,
  ZAR};

layout RateFreqInfreq: [Rate] = rec{date, USD, GBP}
  + rec{JPY, BGN, CYP, CZK, DKK, EEK, HUF, LTL,
        LVL, MTL, PLN, ROL, RON, SEK, SIT, SKK,
        CHF, ISK, NOK, HRK, RUB, TRL, TRY, AUD,
        BRL, CAD, CNY, HKD, IDR, ILS, INR, KRW,
        MXN, MYR, NZD, PHP, SGD, THB, ZAR};
layout RateSoa: [Rate] = rec{date},
  + rec{USD} + rec{GBP} + rec{JPY} + rec{BGN}
```

```
  + rec{CYP} + rec{CZK} + rec{DKK} + rec{EEK}
  + rec{HUF} + rec{LTL} + rec{LVL} + rec{MTL}
  + rec{PLN} + rec{ROL} + rec{RON} + rec{SEK}
  + rec{SIT} + rec{SKK} + rec{CHF} + rec{ISK}
  + rec{NOK} + rec{HRK} + rec{RUB} + rec{TRL}
  + rec{TRY} + rec{AUD} + rec{BRL} + rec{CAD}
  + rec{CNY} + rec{HKD} + rec{IDR} + rec{ILS}
  + rec{INR} + rec{KRW} + rec{MXN} + rec{MYR}
  + rec{NZD} + rec{PHP} + rec{SGD} + rec{THB}
  + rec{ZAR};

// Currency and query classes used
// in our C++ code
enum Currency : i8 { USD, GBP }
class Query {
  date: string;
  currency: Currency;
}

// Code for lookup given one pool of rates,
// corresponds to the code in the
// AosOnePool fixture
<rates: [Rate<rates>]>
def lookup_rate_one(Query query): double {
  var it = <rates>binary_search(q.date);
  if (it != null)
      return q.currency == Currency::USD
        ? it->USD
        : it->GBP;
  return -1;
}

// Code for lookup given one pool of rates,
// corresponds to the code in the
// MixedManyPools, MixedManyPoolsSoa
// fixtures
<recent: [Rate<recent>], historical: [Rate<historical>]>
def lookup_rate(Query query, string date_threshold): double {
  var rate: double;
  if (q.date > DATE_RECENT) {
    var recent_it = <recent>binary_search(q.date);
    if (recent_it != null)
        return q.currency == Currency::USD
          ? recent_it->USD
          : recent_it->GBP;
  } else {
   var historical_it = <historical>binary_search(q.date);
   if (historical_it != null)
     return q.currency == Currency::USD
       ? historical_it->USD
       : historical_it->GBP;
  }
  return -1;
}

// Implementation of binary search, we use
// std::lower_bound in our C++ code
<pl: [Rate<pl>]>
def binary_search(date: String): Rate<pl> {
  var lo = 0, hi = pl.size();
  while (lo <= hi) {
    var mid = low + (high - low) / 2;
    if (pl[mid].date < date)
      lo = mid + 1;
    else if (pl[mid.date] > date)
      hi = mid - 1;
    else
      return pl[mid];
```

```
    }
    return null;
}
```