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

Alexandros Tasos\textsuperscript{a,\textdagger,\textasteriskcentered}, Juliana Franco\textsuperscript{b}, Sophia Drossopoulou\textsuperscript{a,b}, Tobias Wrigstad\textsuperscript{c}, Susan Eisenbach\textsuperscript{a}

\textsuperscript{a}Imperial College London
\textsuperscript{b}Microsoft Research, United Kingdom
\textsuperscript{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 \textsc{Shapes}, which aims to offer developers more fine-grained control over the placement of data, without sacrificing memory safety or object abstraction. In \textsc{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 \textsc{Shapes}, present its type and memory safety

\textdagger 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)

1Supported 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.
Writing cache-friendly programs tends to be more straightforward in un-managed 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. 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

---

2Commonly 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 \S8, Related Work). As such, we propose a language extension-based solution to these issues with \textbf{SHAPES}.

\textbf{SHAPES} aims to support efficient cache use whilst enabling the programmer to write straightforward OO code and without having to abandon key OO concepts. \textbf{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 \textit{SHAPES}

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

\begin{figure}[h]
\begin{center}
\includegraphics[width=\textwidth]{stage1.png}
\end{center}
\caption{Stage 1: Language-based field clustering}
\end{figure}

\textit{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 \texttt{Student} object will allocate it inside the unique, implicit pool corresponding to class \texttt{Student}.

An optional \texttt{layout} declaration specifies how class instances are laid out inside that implicit pool. A \texttt{layout} declaration splits the class’ fields into \textit{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).
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 3, Student s3 is created “inside” the none pool, hence it will be placed on the heap.

\[\text{Student} = \text{rec} \{\text{name}, \text{age}\} + \text{rec} \{\text{supervisor}\}\]

\[\text{Student} = \text{rec} \{\text{name}\} + \text{rec} \{\text{age}\} + \text{rec} \{\text{supervisor}\}\]

\[\text{pStu1: StudentL1, pStu2: StudentL2}\]

\[s1 = \text{new Student}<\text{pStu1}>;\]

\[s2 = \text{new Student}<\text{pStu2}>;\]

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
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.

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*.

![Homogeneous Student pool](image1) ![Heterogeneous Student pool](image2)

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`:

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

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 $p_1$ has a bound of the form $[C<p_1, \ldots, p_{n_1}>]$, then all objects residing in $p_1$ must also have the type $C<p_1, \ldots, p_{n_1}>$. 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 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 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.
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) 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. 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 lists their specifications. We used CMake as our build system; all case studies were compiled as a Release
build (which implies the -O3 -DNDEBUG flags). We used the Google C++ Benchmark library for our measurements 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

```cpp
// SHAPES++ code

class Point<pPt> : [Point<pPt>]
  (x: double; y: double;
    pSeg: [Segment<pSeg, pPt>,
      pPt: [Point<pPt>]] {
      p1: Point<pPt>; p2: Point<pPt>;
      def len(): double {
        var dx = p2.x - p1.x, dy = p2.y - p1.y;
        return sqrt(dx * dx + dy * dy);
      }
    }
  }

layout PointL: Point = rec{x} + rec{y};
layout SegmentL: Segment = rec{p1} + rec{p2};

def main() {
  pools pPt1: PointL<pPt>,
    pSeg1: SegmentL<pSeg1, pPt1>;
  ... // Create objects in pPt1, pSeg1
  print(sum_lens_shapespp<pPt1, pSeg1>())
}
```

```cpp
// OP2 code

<ps: [Segment<ps, pp>], pp: [Point<pp>]> 

def sum_lens_shapespp(): double {
  var sum = 0;
  foreach (var e: ps)
    sum += e.len();
  return len_sum;
}
```

Figure 10: Example SHAPES++ code
class Point {
  double x, y;
};

class Segment {
  Point *p1, *p2;
};

void sum_lens(double* acc,
              double* x1, double* y1,
              double* x2, double* y2)
{
  double dx = *x2 - *x1, dy = *y2 - *y1;
  acc += sqrt(dx * dx + dy * dy);
}

... // Fill x_data, y_data, p1_data, p2_data

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

...
and *airfoil* in §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++ (§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].
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
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).

---

– 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 (voxel1) 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*: 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*, a high-level calculus that
Figure 15: Syntax of \texttt{SHAPES}^h where \( p \in \text{PoolVariableId, } x \in \text{LocalVariableId, } C \in \text{ClassId, } f \in \text{FieldId, } m \in \text{MethodId, and } L \in \text{LayoutId. Differences from standard OO languages in highlight.}

\[\begin{align*}
\text{prog} & \ ::= \ cd^* \ ld^* \quad \text{Program} \\
\text{cd} & \ ::= \ \text{class } C(p: [C(p^*))^+] \{ \ fd^* \ md^* \} \quad \text{ClassDecl} \\
\text{fd} & \ ::= \ f : t ; \quad \text{FieldDecl} \\
\text{md} & \ ::= \ \text{def } m(x: t) : t \{ \text{localPools ; localVars ; stmts} \} \quad \text{MethodDecl} \\
\text{localPools} & \ ::= \ \text{pools \ (p: L(ps))^*} ; \quad \text{LocalPools} \\
\text{localVars} & \ ::= \ \text{vars \ (x: t)^*} ; \quad \text{LocalVars} \\
\text{stmts} & \ ::= \ e \ | \ e ; \text{ stmts} \quad \text{Statements} \\
\text{e} & \ ::= \ \text{null | x | this | new } t | x.m(x) | x.f | x.f = x | x = e \quad \text{Expression} \\
\text{t} & \ ::= \ C(ps) \quad \text{ObjectType} \\
\text{id} & \ ::= \ \text{layout } L: [C] = (\text{rec } \{f^* \})^* \quad \text{LayoutDecl} \\
\text{np} & \ ::= \ p | \text{ none} \quad \text{PoolVariableOrNone} \\
\text{ps} & \ ::= \ np | np \cdot ps \quad \text{PoolVariables}
\end{align*}\]

provides the appropriate constructs (§4), and \texttt{SHAPES}^f, a low-level language designed with efficiency in mind for implementing these constructs (§5). §6 presents the translation of \texttt{SHAPES}^h code into \texttt{SHAPES}^f.

\texttt{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 \texttt{SHAPES}^h types are parameterised with pool variables as parameters (§4.3); it is thanks to these pool parameters that the \texttt{SHAPES}^h type system can statically enforce pool uniformity and homogeneity (§2 Stages 5, 6).

Representation of entities in \texttt{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 \texttt{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 \texttt{SHAPES}^h as a formalism and not as a runtime target, such inefficiencies are of no concern; in §5 we will demonstrate how \texttt{SHAPES}^f 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
$X \in \text{Heap} = \text{Address} \rightarrow (\text{Object} \cup \text{Pool})$

$\text{Address} = \text{ObjectAddress} \uplus \text{PoolAddress}$

$\text{Object} = \text{ClassId} \times \text{PoolArg}^* \times \text{Record}$

$\text{Pool} = \text{LayoutId} \times \text{PoolArg}^*$

$\pi \in \text{PoolArg} = \text{PoolAddress} \uplus \{\text{none}\}$

$\rho \in \text{Record} = \text{FieldId} \rightarrow \text{Value}$

$\beta \in \text{Value} = \text{ObjectAddress} \uplus \{\text{null}\}$

$\Phi \in \text{SFrame} = \text{VariableId} \rightarrow (\text{Value} \cup \text{PoolArg}) \cup (\{\text{none}\} \rightarrow \{\text{none}\})$

$\Sigma \in \text{Stack} = \text{SFrame}^*$

Figure 16: Dynamic Entities of SHAPE$h$ where $\omega \in \text{ObjectAddress}$.

SHAPE$h$.

4.1. The SHAPE$h$ language

Figure 15 presents the syntax of SHAPE$h$; highlighted entities represent the syntactic entities that are novel with respect to other object-oriented languages. Classes in SHAPE$h$ 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 $L : C = \text{rec } \{f_1\} ; \ldots ; \text{rec } \{f_n\}$, 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 $f_1$; the second cluster will contain fields $f_2$, 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 $C(p \cdot ps)$ 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 $C(\text{none} \cdot ps)$ 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 $\text{none} \cdot ps$. 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(p · ps)`, contains objects of type \( C(p · ps) \), 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(p · ps)`.

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 \( · \) for list concatenation: \( p · ps \) 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 \( F(x_1 · .. · x_n) \) as a shorthand for \( F(x_1) · .. · F(x_n) \).

4.2. Execution of \( \text{SHAPES}^h \) Programs

The execution of \( \text{SHAPES}^h \) corresponds to the execution of a typical OO language when not taking the highlighted parts into account. The \( \text{SHAPES}^h \) heap adds pool entities to standard OO; these pool entities consist of the layout \( L \) they adhere to. To express the correspondence between \( \text{SHAPES}^h \) and \( \text{SHAPES}^l \), we enrich the \( \text{SHAPES}^h \) 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(p_1 .. p_n)` will also contain the addresses of the pools \( p_1 \) to \( p_n \)).

The \( \text{SHAPES}^h \) operational semantics is given in terms of large steps semantics, and has the form \( \langle \mathcal{X}, \Sigma, stmts \rangle \xrightarrow{\text{large steps}} \langle \mathcal{X}', \Sigma', \beta \rangle \). That is, a heap \( \mathcal{X} \), a stack of frames \( \Sigma \), and a sequence of statements \( stmts \) are reduced to a new heap \( \mathcal{X}' \), a new stack \( \Sigma' \), and a value \( \beta \).
Figure 17: SHAPES example used in §4 and §5

```java
layout StudentL: Student = rec
  ■ name,
  ■ age
+ rec ■ supervisor;
...
pools pStu: StudentL<pStu, none>;
at = new Student<pStu, none>;
ec = new Student<pStu, none>;
jf = new Student<pStu, none>;
sd = new Professor<none>;
jf.supervisor = sd;
```

Figure 18: SHAPES runtime configurations consist of a stack (Σ) of frames (Φ) mapping identifiers to values, and heaps (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 π1, ω1, ω2, ω3, ω4. For convenience, if Σ = Φ · Σ' we use Σ(x) and Σ[x → β] as shorthands for Φ(x) and Φ[x → β] · Σ', 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 ω4 in Figure 18); an object stored in a pool π has a pool address π as its first parameter (e.g., Students at addresses ω1, ω2, ω3 belong to pool with address π1. The fields' values are stored as a record (ρ), which maps the
object’s fields to values.

Pools consist of a layout identifier $L$, and a sequence of pool arguments. The layout identifier determines how the objects inside the pool are laid out and Pools can only store instances of the class corresponding to layout identifier. As mentioned, pools in $\text{SHAPES}^h$ 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 $\text{null}$. Because $\text{SHAPES}^h$ allows pools to be referenced by variables, stack frames ($\Phi$) map variables to either values, pool addresses or $\text{none}$. $\text{SHAPES}^h$ uses sequences of stack frames ($\Sigma$). We require for convenience that any frame maps $\text{none}$ to $\text{none}$. The operational semantics of $\text{SHAPES}^h$ (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 $\text{new } C\langle ps \rangle$ expression) into objects. Secondly, $\text{MethodCall}$ 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).

$$\begin{align*}
\text{[Value]} & \quad \mathcal{X}, \Sigma, \text{null} \rightsquigarrow \mathcal{X}, \Sigma, \text{null} \\
\text{[Assignment]} & \quad \mathcal{X}, \Sigma, e \rightsquigarrow \mathcal{X}', \Sigma', \beta \\
\text{[Statement sequence]} & \quad \mathcal{X}, \Sigma, e ; \; \text{stmts} \rightsquigarrow \mathcal{X}'', \Sigma'', \beta \\
\text{[New Object]} & \quad \Sigma(ps) = \pi s \quad \omega \notin \mathcal{X} \quad fs = F_s(C) \\
& \quad \mathcal{X}' = \mathcal{X}[\omega \mapsto (C, \pi s, [fs \mapsto \text{null}^{[fs]}])] \\
\text{[Object Read]} & \quad \Sigma(x) = \omega \quad \mathcal{X}(\omega) = (C, \pi s, \rho) \\
& \quad \mathcal{X}, \Sigma, x.f \rightsquigarrow \mathcal{X}, \Sigma, \rho(f) \\
\text{[Object Write]} & \quad \Sigma(x) = \omega \quad \Sigma(x') = \omega' \quad \mathcal{X}(\omega) = (C, \pi s, \rho) \\
& \quad \mathcal{X}' = \mathcal{X}[\omega' \mapsto (C, \pi s, \rho[f \mapsto \omega'])] \\
& \quad \mathcal{X}, \Sigma, x.f = x' \rightsquigarrow \mathcal{X}', \Sigma, \omega' \\
\end{align*}$$

Figure 19: Operational semantics for pool-agnostic operations.
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.

<table>
<thead>
<tr>
<th>Fun</th>
<th>Used to Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{F}$</td>
<td>The type of a field in a given class.</td>
</tr>
<tr>
<td>$\mathcal{M}$</td>
<td>The definition of a method in a given class.</td>
</tr>
<tr>
<td>$\mathcal{B}$</td>
<td>The bound type of the given class parameter in a given class.</td>
</tr>
<tr>
<td>$\mathcal{P}$</td>
<td>All parameters of a given class.</td>
</tr>
<tr>
<td>$\mathcal{C}$</td>
<td>The class corresponding to a given layout.</td>
</tr>
</tbody>
</table>

Typing takes place in the context of an environment $\Gamma$, 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).**

$$
\Gamma \in \text{TypingContext} ::= x : t, \Gamma \mid p : u, \Gamma \mid \epsilon
$$

$$
u \in \text{PoolTypeOrPoolBound} ::= pt \mid pb
$$

$$
pt \in \text{PoolType} ::= L(ps)
$$

$$
pb \in \text{PoolBound} ::= [C(ps)] \mid \text{None}
$$

$$
T ::= t \mid u
$$

We distinguish three kinds of types:

**Object Types** ($C(ps)$), where $C$ is a class and $ps$ are pool arguments, some of which may be \textbf{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** ($L(ps)$) describe pools which store objects of type $C$ and are organised according to layout $L$. The arguments $ps$ specify which pools contain the objects pointed by the fields of the objects stored in this pool. Pool types characterise pool values, \textit{i.e.}, pools allocated dynamically through execution of a method's preamble.
Expression and Statement Types. Typing has the standard form \( \Gamma \vdash e : t \) and 
\( \Gamma \vdash \text{stmts} : t \). 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. \textbf{null} can have any well-formed 
object type (\textbf{Value}). The type of a variable \( x \) is looked-up in \( \Gamma \) (\textbf{Variable}). Assignment to a local variable is valid if both the variable and the right-hand-side expression have the same type (\textbf{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 (\textbf{Statements}). Creation of a new object is valid and has type 
\( C\langle ps \rangle \) if \( C\langle ps \rangle \) is a valid type (\textbf{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
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 \( F(C,f) \) 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 \( [Ps(C)/ps] \). For example, the term \( s1.supervisor \) in Line 19 of Figure 8 has type \( \text{Professor}<\text{pProf1}> \).

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 \( s1.supervisor = \text{new Professor}<\text{pProf1}> \) would be legal, while, given an \( \text{otherStudent} \) of type \( \text{Student}<\text{none}, \text{none}> \), the assignment \( \text{otherStudent.supervisor} = \text{new Professor}<\text{pProf1}> \) would be illegal.

Well-formed Types. Figure 22 describes well-formedness of types, which is the way we statically enforce homogeneity in \( \text{SHAPES}^h \). 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., \( B(C,Ps(C)[i]) \)) 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

\[
\frac{\forall i. \Gamma \vdash ps[i] :: B(C,Ps(C)[i])/[Ps(C)/ps]}{\Gamma \vdash C\langle ps \rangle} \quad \frac{\Gamma \vdash C\langle ps \rangle}{\Gamma \vdash [C\langle ps \rangle]} \]

\[
\frac{\Gamma \vdash C\langle ps \rangle \quad \mathcal{A}(L) = C}{\Gamma \vdash L\langle ps \rangle} \quad \frac{\Gamma \vdash p :: \mathcal{A}(L) = C}{\Gamma \vdash p :: [C\langle ps \rangle]} \]

\[
\frac{\Gamma \vdash \text{none} :: \text{None}}{\Gamma \vdash [C\langle ps \rangle]} \quad \frac{\Gamma \vdash \text{none} :: [C\langle ps \rangle]}{\Gamma \vdash [C\langle ps \rangle]} \]

Figure 22: Well-formed Types
given in $\Gamma$, and by $\text{PoolVar}$, a pool variable which adheres to $L(ps)$ also adheres to $[C(ps)]$, where $C$ is the class of the layout $L$.

Thus, in Line 19 of Figure 8 the type $\text{Student}<p\text{Stu1}, p\text{Prof1}>$ is well-formed (as $p\text{Stu1}$ adheres to the bound $[\text{Student}<p\text{Stu1}, p\text{Prof1}>]$, and $p\text{Prof1}$ adheres to the bound $[\text{Professor}<p\text{Prof1}>]$). However, in Line 21 the type $\text{Student}<p\text{Stu1}, p\text{Prof2}>$ is badly formed; for it to be well-formed, we would need for $p\text{Stu1}$ to adhere to the bound $[\text{Student}<p\text{Stu1}, p\text{Prof2}>]$, but this cannot be since the bound of $p\text{Stu1}$ is $[\text{Student}<p\text{Stu1}, p\text{Prof1}>]$. 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 $\pi$, and any sequence of field reads $f_1, \cdots, f_n$, if $o_1.f_1 \cdots f_n$ and $o_2.f_1 \cdots f_n$ are defined and not $\text{null}$, then they must reside in the same pool $\pi'$. Through an inductive argument, we can convince ourselves that homogeneity is equivalent to local homogeneity, where the latter requires that for any pool $\pi$ any objects $o_1$ and $o_2$ allocated in $\pi$, and any field $f$, if $o_1.f$ and $o_2.f$ are defined and not $\text{null}$, then they must reside in the same pool $\pi'$.

It remains to think how to achieve local homogeneity:

Remember that the static type of an object $(C(p \cdot ps))$ determines the pool that object resides in, as well as the pools the object’s fields reside in. In
particular, if $\Gamma \vdash x : C(p \cdot ps)$, then $\Gamma \vdash x : C'(p' \cdot ps)$, where $C'$ is some class, and $p' = p, p' = \text{none}$ 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 $C(p \cdot ps)$. 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 $C(p \cdot ps)$ and $C'(p' \cdot ps')$ 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).**

$\vdash \Gamma$ iff $\forall (\cdot : T) \in \Gamma, \Gamma \vdash T \land \forall p. [\Gamma \vdash p :: [C(ps)] \rightarrow ps[0] = p ]$

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 $\vdash \Gamma$, $\Gamma \vdash C(ps)$ and $\Gamma_C \vdash C'(ps')$ (where $\Gamma_C$ is the environment used to type-check the definition of class $C$, see §Appendix B), then $\Gamma \vdash C'(ps'[Ps(C)/ps])$.

**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).**

$\vdash \Gamma \land \Gamma \vdash C(p \cdot ps) \land \Gamma \vdash C'(p \cdot qs) \rightarrow ps=qs \land C=C'$. 32
Proof. If $\Gamma \vdash C(p \cdot ps)$ and $\Gamma \vdash C'(p \cdot qs)$, then $\Gamma \vdash p :: C(p \cdot ps)$ and $\Gamma \vdash p :: C'(p \cdot ps')$ from ObjTypeWF and the fact that $B(C; Ps(C)[0]) = Ps(C)$ (Definition 8). However, because $\Gamma$ 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 $\Gamma \vdash o_1 : C(p \cdot ps)$ and $\Gamma \vdash o_2 : C(p \cdot ps')$, 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 $\Gamma \vdash o_1.f : C'(p_1 \cdot \ldots)$ and $\Gamma \vdash o_2.f : C'(p_2 \cdot \ldots)$ 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 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 $C(p_1 \cdot p_2)$ in the caller’s environment and the type $C(p_3 \cdot p_4)$ 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 $\pi$ or none. That is, the pool parameters of static types are
variables (e.g., $C(p_1 \cdot p_2)$), whereas the pool parameters of runtime types are pool addresses (e.g., $C(\pi_1 \cdot \pi_2)$). 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., $C(\pi_1 \cdot \pi_2)$) throughout execution.

**Definition 3 (Runtime types).** A runtime type $\tau$ is defined as follows:

$$\tau \in \text{RunType} ::= \text{RunClassType} \cup \text{RunPoolType} \cup \text{RunBound}$$

$$\text{RunClassType} ::= C(\pi_1 .. \pi_n)$$

$$\text{RunPoolType} ::= L(\pi_1 .. \pi_n)$$

$$\text{RunBound} ::= [C(\pi_1 .. \pi_n)]$$

In the context of a well-formed configuration, we can expect that an object with runtime type $C(\pi \cdot \pi s)$ belongs to the pool with address $\pi$ and that the pool at address $\pi$ has a runtime type $L(\pi \cdot \pi s)$ such that $C(l(L)) = C$. This implies uniformity. Moreover, for two objects $o_1, o_2$ with runtime type $C(\pi \cdot \pi s)$, we require that $o_1.f, o_2.f$ point to objects that belong to the same pool $\pi'$. $\pi'$ 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 $\Phi$, an object, pool, or class parameter adheres to the static type $C(ps), L(ps)$, or, $[C(ps)]$ 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:
  - $X \models \omega \cdot C(\pi s)$ iff $X(\omega) = (C, \pi s, \rho) \land X \models \pi s[0] \cdot [C(\pi s)] \land \forall f. X \models \rho(f) : C(f)\{P(\pi s(C))/\pi s\}$
  - $X \models \pi \cdot L(\pi s)$ iff $X \models \pi s[0] : L(\pi s) \land C(l(L)) = C \land \forall i. X \models \pi s[i] : B(C, P(\pi s(C))[i])\{P(\pi s(C))/\pi s\}$

- Weak agreement for objects, pools, and bounds:
\[ X \vdash \omega : C(\pi_s) \quad \text{iff} \quad X(\omega) = (C, \pi_s, \_)
\]
\[ X \vdash \text{null} : C(\_)
\]
\[ X \vdash \pi : L(\pi_s) \quad \text{iff} \quad X(\pi) = (L, \pi_s) \land \pi = \pi_s[0]
\]
\[ X \vdash \pi : [C(\pi_s)] \quad \text{iff} \quad X(\pi) = (L, \pi_s) \land \pi = \pi_s[0] \land Cl(L) = C
\]
\[ X \vdash \text{none} : [C(\_)]
\]
\[ X \vdash \text{null} : \text{None}
\]

Well-formed heap:
\[ \models X \iff \forall \omega \in \text{dom}(X). \exists \tau. X \vdash \omega \triangleleft \tau \]
\[ \forall \pi \in \text{dom}(X). \exists \tau. X \vdash \pi \triangleleft \tau
\]

Well-formed stack frame and heap against an environment:
\[ \Gamma \vdash X, \Phi \quad \text{iff} \quad \models X \land \forall x \in \text{dom}(\Phi). \exists C, ps. [\Gamma(x) = C(\psi) \land X \vdash \Phi(x) : C(\Phi(\psi))] \land
\[ \forall p \in \text{dom}(\Phi). \exists L, C, ps. [\Gamma(p) = L(\psi) \land X \vdash \Phi(p) : L(\Phi(\psi))] \lor
\[ \Gamma(p) = C(\psi) \land X \vdash \Phi(p) : C(\Phi(\psi))]
\]

Well-formed sequence of stack frames and heap against a sequence of environments:
\[ \epsilon \vdash X, \epsilon \]
\[ \Gamma \cdot \Gamma_s \vdash X, \Phi \cdot \Sigma \quad \text{iff} \quad \Gamma \vdash X, \Phi \land \Gamma_s \vdash X, \Sigma
\]

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 \( X \), stack frame sequence \( \Sigma \), corresponding typing environment sequence \( \Gamma_s \), and sequence of statements \( \text{stmts} \): \[ \text{If } \Gamma_s \vdash X, \Sigma \land \Gamma_s[0] \vdash \text{stmts} : C(\psi) \land X, \Sigma, \text{stmts} \rightarrow X', \Sigma', \beta \]

**then** \( \Gamma_s \vdash X', \Sigma' \land X' \vdash \beta : C(\Sigma'(\psi)) \)

**Proof.** By structural induction over the derivation \( X, \Sigma, \text{stmts} \rightarrow X', \Sigma', \beta \).

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 f. 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 f**: Low-Level SHAPES

We now present SHAPES f, 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 f are implicitly represented on the heap (similar to how objects in
most language runtimes are implicitly represented as contiguous chunks of contiguous memory). and objects allocated into \( \text{SHAPES}^\ell \) pools are allocated consecutively. Moreover, \( \text{SHAPES}^\ell \) instructions can be easily translated to existing low-level intermediate representations such as LLVM \cite{llvm} and/or invocations to a standard memory allocation library (e.g., \texttt{malloc()}).

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

5.1. Runtime Configuration

Like \( \text{SHAPES}^h \), the \( \text{SHAPES}^\ell \) runtime configuration consists of a heap \( (\chi) \) and a stack \( (\sigma) \) of frames \( (\phi) \). 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 \( (\gamma) \). Values can be addresses \( (\alpha) \), natural numbers, or \texttt{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 \( \text{SHAPES}^\ell \) configuration that could occur after executing the example of Figure 17. This configuration shows the stack values corresponding to the pool \texttt{psstu} (Line 5) and the objects \texttt{at}, \texttt{ec}, \texttt{jf}, and \texttt{sd} (Lines 5–8), as well as how these entities are represented on the \( \text{SHAPES}^\ell \) heap. Figure 23b lists the components of a \( \text{SHAPES}^\ell \) pool and presents the nomenclature used for \( \text{SHAPES}^\ell \) pools throughout this section.

A standalone object (e.g., the \texttt{Professor} corresponding to \texttt{sd}, Line 10) is modelled as a contiguous chunk of allocated memory. A reference to a standalone object consists of the address \( \alpha \) to this chunk (e.g., variable \texttt{sd} on the
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 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.
χ ∈ Heap^ℓ = Address^ℓ → (Size^ℓ ∪ Capacity^ℓ ∪ Value^ℓ)
φ ∈ SFrame^ℓ = Variable^ℓ → Value^ℓ
σ ∈ Stack^ℓ = (SFrame^ℓ)^∗
γ ∈ Value^ℓ = Address^ℓ ∪ Index^ℓ ∪ {null}

\[ \alpha ∈ Address^ℓ = \mathbb{N} \]
\[ j ∈ Size^ℓ = \mathbb{N} \]
\[ M ∈ Capacity^ℓ = \mathbb{N} \]
\[ k ∈ Index^ℓ = \mathbb{N} \]

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 \( \text{SHAPES}^\ell \) 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 \( \text{SHAPES}^h \): If \( Γ ⊢ o : C⟨ p · ps ⟩ \), 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; \( \text{SHAPES}^\ell \) suffers from no such constraints. In §7 we discuss how \( \text{SHAPES} \) can be extended so that developers can reduce the footprint of references even further.

5.2. Syntax of \( \text{SHAPES}^\ell \)

Figure 25 presents the syntax of \( \text{SHAPES}^\ell \). A program consists of functions (fun^\ell), each with parameters, local variables, and a body. Unlike \( \text{SHAPES}^h \),
\[ \text{prog}^\ell ::= (\text{fun}^\ell)^+ \quad \text{Program} \]

\[ \text{fun}^\ell ::= \text{fun} \ (\text{this}, p^+, x) \ {\text{vars}^\ell; \text{stmts}^\ell} \quad \text{Function} \]

\[ \text{stmts}^\ell ::= \text{rhs}^\ell | \text{stmts}^\ell \quad \text{Statements} \]

\[ \text{rhs}^\ell ::= \text{null} | x = \text{rhs}^\ell | x | \text{fn}(x^+) \quad \text{Instruction} \]

\[ | \text{alloc}(N) | \text{read}(x, i) | \text{write}(x, x', i) \]

\[ | \text{plalloc}(p, N^*) | \text{plread}(p, x, i, N, j) | \text{plwrite}(p, x, x', i, N, j) \]

\[ \text{vars}^\ell ::= \text{locals} (p = \text{plcreate}(N^*)^x) \quad \text{LocalsDecl} \]

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

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 (\text{alloc}, \text{read}, \text{write}) and pool-aware variants (\text{plalloc}, \text{plread}, \text{plwrite}, \text{plcreate}).

5.3. Operational Semantics

SHAPES\( ^\ell \) execution has the format \( \chi, \sigma, \text{stmts}^\ell \leadsto \chi', \sigma', \gamma \). Thus, a SHAPES\( ^\ell \) configuration (heap \( \chi \) and stack \( \sigma \)) is reduced to a new configuration and return value \( \gamma \).

Pool-agnostic operations. Pool-agnostic operations in SHAPES\( ^\ell \) are similar to what we would expect from a typical intermediate representation:

- \text{alloc} constructs a new standalone object in memory (of size \( N \)). Construction of a new standalone Professor object is performed with the instruction \text{alloc} (2) (since Professors have 2 fields).

- \text{read} and \text{write} access an object’s field \( f \) given its address and the offset \( i \) of \( f \) inside the object. Instruction \text{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:

- \text{plread} and \text{plwrite}: Suppose that the pooled object \( o \) belongs to pool \( p \) with layout \( L \) and at \( \alpha \) 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 \( \alpha'' = \alpha' + k \times N \).

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

\( \text{plalloc} \) constructs a new pooled object in pool \( p \). Alongside \( p \), it takes a sequence \( N_0 \ldots N_{m-1} \) of parameters that specify the size of a record split in each of the \( m \) clusters. These parameters will allow \( \text{plalloc} \) to initialise the fields of the newly created object to \text{null}. As an example, \( \text{plalloc}(pStu, 2, 1) \) will construct a new \text{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.

\( \text{plcreate} \) 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 \( \text{plcreate}(2, 1) \) to create a pool that adheres to layout \( \text{StudentL} \). 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 \texttt{pccreate}) at the beginning of the method’s body.

The rules for pool-agnostic, pool-aware operations, and \texttt{Method Call} are given in Figure \ref{fig:rule_pa} Figure \ref{fig:rule_paw} and Figure \ref{fig:rule_mc}, 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 \texttt{plalloc} statement, the garbage collector is run so as to grow the pool in question. Rule \texttt{Garbage Collection} (Figure \ref{fig:garbage_collection}) dictates how a garbage collector designed or retrofitted to accommodate \texttt{SHAPES} must operate.

Rule \texttt{Garbage Collection} states that the GC can only run inbetween \texttt{SHAPES} 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 \(\chi, \sigma\) into a new configuration \(\chi', \sigma'\) such that the two are equivalent (\(\chi, \sigma \simeq_{\sigma} \chi', \sigma'\)). That is, all objects and pools reachable in \(\chi, \sigma\) through must have an isomorphic counterpart in \(\chi', \sigma'\) and, additionally, pooled objects cannot be moved into another pool. We present the definition of \(\simeq_{\sigma}\) in §Appendix E.2.

Note that pool growth does not necessarily imply a partial or full GC invocation: The semantics of Rule \texttt{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 (\textit{e.g.}, \texttt{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 $\text{SHAPES}^\ell$ to rely on any particular garbage
collection technique or algorithm, hence we expect adding support for $\text{SHAPES}^\ell$
to existing GCs to not be conceptually excruciating (i.e., modulo the amount
of engineering effort required). Considering our objective to have $\text{SHAPES}$ sup-
port 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 translating $\text{SHAPES}^h$ into $\text{SHAPES}^\ell$. The
most significant difference between $\text{SHAPES}^h$ and $\text{SHAPES}^\ell$ that we need to
take into account during translation is that $\text{SHAPES}^h$ classes are polymorphic
with respect to the layouts of pools (hence their member methods are also
polymorphic), whereas $\text{SHAPES}^\ell$ does not provide any functionality to imple-
ment any such polymorphism. This implies that when translating $\text{SHAPES}^h$ to
$\text{SHAPES}^\ell$:

- We need to know whether an object is standalone or pool allocated so as to
  emit the appropriate variant of an instruction (e.g., $\text{read}$ vs $\text{pread}$).
- 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 pa-
  rameters 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 $\text{clone}()$ for class $\text{Student}$
(Figure 8):
def clone(): Student<pStu, pProf> {   obj.supervisor = this.supervisor;
var obj = new Student<pStu, pProf>;   obj
obj.name = this.name;
obj.age = this.age;
}

The SHAPES\textsuperscript{ℓ} 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\textsuperscript{ℓ} 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\textsuperscript{ℓ}. We use specialised environments (Δ), where pool bounds are eliminated and pool variables have layout types or None:

**Definition 5.** \( \Delta ::= x : t, \Delta \mid p : L(np^+), \Delta \mid p : \text{None}, \Delta \mid \text{none} : \text{None} \)**

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

\[
\Gamma \vdash \Delta \text{ iff } \begin{align*}
\text{dom}(\Gamma) &= \text{dom}(\Delta) \land \\
\forall x. [\Gamma(x) = C(\_)] &\rightarrow \Delta(x) = \Gamma(x) \land \\
\forall p. [\Gamma(p) = L(\_)] &\rightarrow \Delta(p) = \Gamma(p) \land \\
\forall p. [\Gamma(p) = [C(ps)]] &\rightarrow \Delta(p) = \text{None} \lor \exists L. [C(L) = C \land \Delta(p) = L(ps)]
\end{align*}
\]

For two sequences of environments \( \Gamma_s, \Delta_s \), we state that \( \Delta_s \) specialises \( \Gamma_s \) (\( \Gamma_s \vdash \Delta_s \)) as follows:

\[
\Gamma_s \vdash \Delta_s \text{ iff } \forall i. \Gamma_s[i] \vdash \Delta_s[i]
\]

In our examples, we shall be using two specialized environments, \( \Delta_1 \) and \( \Delta_2 \), such that:

\[
\begin{align*}
\Delta_1(s) &= \text{Student}<pStu1, pProf1> & \Delta_2(s) &= \text{Student}<pStu1, pProf1> \\
\Delta_1(pStu1) &= \text{StudentL}<pStu1, pProf1> & \Delta_2(pStu1) &= \text{None} \\
\Delta_1(pProf1) &= \text{ProfL}<pProf1> & \Delta_2(pProf1) &= \text{ProfL}<pProf1>
\end{align*}
\]

We will be using the following definition of StudentL:

\[
\text{layout StudentL: Student} = \text{rec(name, age)} + \text{rec(supervisor)};
\]

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


<table>
<thead>
<tr>
<th>Expression</th>
<th>( \Delta_1 )</th>
<th>( \Delta_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new Student(pStu1, pProf1)</code></td>
<td><code>plalloc(p, 2, 1)</code></td>
<td><code>alloc(3)</code></td>
</tr>
<tr>
<td><code>s.age</code></td>
<td><code>pRead(pStu1, s, 0, 1, 1)</code></td>
<td><code>read(s, 1)</code></td>
</tr>
<tr>
<td><code>s.getAge()</code></td>
<td><code>#Student_getAge_StudentL_ProfL(s, pStu1, pProf1)</code></td>
<td><code>#Student_getAge_None_ProfL(s, null, pProf1)</code></td>
</tr>
</tbody>
</table>

Figure 26: Example translations

<table>
<thead>
<tr>
<th>Fun</th>
<th>Used to Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{F} )</td>
<td>All fields declared in a given class.</td>
</tr>
<tr>
<td>( \mathcal{O} )</td>
<td>Offset of a field identifier in a class; i.e., the cluster containing the field, and the field’s position within that cluster.</td>
</tr>
<tr>
<td>( \mathcal{R} )</td>
<td>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.</td>
</tr>
</tbody>
</table>

Translating Expressions and statements. Figure 27 defines the translation of SHAPES\(^b\) expressions and statements in terms of rules of the form \([ e ]_{\Delta} \) and \([ stmts ]_{\Delta} \), where \( e \) and \( stmts \) are SHAPES\(^b\) expressions or statement sequences, and \( \Delta \) 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 unmodified 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 \( C(np \cdot \_) \) where \( \Delta(np) = \text{None} \), we emit the instruction `alloc(N)` where \( N \) is the number of fields in class \( C \). For a pooled object, i.e., an object of type \( C(p \cdot \_) \) where \( \Delta(p) = L(\_) \) we emit the instruction `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 `read(x, k)` where \( k \) is the offset of \( f \) in the class of \( x \). In the second case, we emit `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 \( j \)
\[ [x]_\Delta \triangleq x \]
\[ [x = rhs]_\Delta \triangleq [\text{rhs}]_\Delta \]
\[ [e; \text{stmts}]_\Delta \triangleq [e]_\Delta \setminus [\text{stmts}]_\Delta \]
\[ \text{new } C\langle np \cdot \_\rangle_\Delta \triangleq \begin{cases} \text{alloc}([F_s(C)]) & \text{if } \Delta(np) = \text{None} \\ \text{plalloc}(p, |f_{s0}| \ldots |f_{sn}|) & \text{if } np = p \land \Delta(p) = L\langle \_ \rangle \\ \land f_{s0} \ldots f_{sn} = R_s(L) \end{cases} \]
\[ [x.f]_\Delta \triangleq \begin{cases} \text{read}(x, O(C, f)) & \text{if } \Delta(x) = C\langle np, \_ \rangle \land \Delta(np) = \text{None} \\ \text{pread}(p, x, i, N, j) & \text{if } \Delta(x) = C\langle p, \_ \rangle \land \Delta(p) = L\langle \_ \rangle \\ \land O(L, f) = (i, j) \land N = |R_s(L)[i]| \end{cases} \]
\[ [x.f = x']_\Delta \triangleq \begin{cases} \text{write}(x, x', O(C, f)) & \text{if } \Delta(x) = C\langle np, \_ \rangle \land \Delta(np) = \text{None} \\ \text{plwrite}(p, x, x', i, N, j) & \text{if } \Delta(x) = C\langle p, \_ \rangle \land \Delta(p) = L\langle \_ \rangle \\ \land O(L, f) = (i, j) \land N = |R_s(L)[i]| \end{cases} \]
\[ [x.m(x')]_\Delta \triangleq \text{Name} \Delta'(m)(x, np_1' \ldots np_k', x') \]
\[ \quad \text{if } \Delta(x) = C\langle np_1 \ldots np_k \rangle \\ \land \Delta' = \text{this} : C\langle np_1 \ldots np_k \rangle, p_1 : \Delta(np_1), \ldots, p_k : \Delta(np_k) \]
\[ \land \forall i \in 1..k, np_i' = \begin{cases} \text{null} & \text{if } \Delta(np_i) = \text{None} \\ np_i & \text{otherwise} \end{cases} \]
\[ \text{pools } p_1 : \ldots p_n : \ldots \text{locals } x_1 : \ldots x_m : \ldots \text{stmts}]_\Delta \]
\[ \text{locals } p_1 = \text{plcreate}(N_s) : \ldots p_n = \text{plcreate}(N_s) \ x_1 \ldots x_m ; \text{stmts}]_\Delta \]

\[ \forall i, [R_s(\Delta(p_i))] = f_{s0} \ldots f_{sn} \rightarrow N_s = |f_{s0}| \ldots |f_{sn}| ] \]

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\(^f\) under environments \( \Delta_1 \) and \( \Delta_2 \), 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 $\Delta$ being used.

$$\text{Name}_\Delta(m) \equiv \#C.m.G_1 \ldots G_n$$

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

Figure 26 shows the translation of the method call $s$.getAge() under $\Delta_1$ and $\Delta_2$. Notice that in the case of $\Delta_2$, 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 $\text{SHAPES}^k$ functions is performed by enumerating all possible specialised environments. We obtain all such environments through the $\text{SpecialiseClass}$ function, which substitutes the types of formal pool parameters with layout types or None.

$$\text{SpecialiseClass}(C) \equiv \{ \Delta \mid \text{dom}(\Delta) = \mathcal{P}s(C) \land \forall p \in \mathcal{P}s(C). [\mathcal{B}(C, p) = C'(ps)] \rightarrow \Delta(p) = \text{None} \lor \exists L.\mathcal{C}(L) = C' \land \Delta(p) = L\langle ps \rangle \}$$

Thus, we define translation of a method as:

$$\text{SpecialiseMethod}(C, m) \equiv \{ \text{Name}_\Delta(m)(\text{this}, p_1, \ldots, p_n, x')\{[\text{localPools}; \text{localVars}; \text{stmts}]_{\Delta'} \} \mid \Delta \in \text{SpecialiseClass}(C) \land \Delta' = \Delta, \text{this} : C\langle p_1 \ldots p_n \rangle, x' : t', p'_1 : L_1\langle ps'_1 \rangle, \ldots, p'_{k} : L_k\langle ps'_k \rangle, \ldots, x'_m : C'_m\langle ps'_m \rangle \},$$

where $\mathcal{M}(C, m) = (\_, x' : t', \text{localPools}; \text{localVars}, \text{stmts})$, and $\text{localPools} = \text{pools} p'_1 : L_1\langle ps'_1 \rangle \ldots, p'_{k} : L_k\langle ps'_k \rangle$; and $\text{localVars} = \text{vars} x_1 : C'_1\langle ps''_1 \rangle \ldots, x'_m : C'_m\langle ps''_m \rangle$.

Specialisation will always terminate. This is because a specialisation replaces the pool bound $[C\langle ps \rangle]$ of each formal pool parameter with a layout type $L\langle ps \rangle$ (such that $\mathcal{C}(L) = C$) 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 $\text{SHAPES}^h$ code in a high level configuration gives equivalent results as executing the translation of that $\text{SHAPES}^h$ 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 $(X,\Sigma \simeq_{\Gamma_s,\Delta_s,\mathcal{I},\text{stmts}} \chi,\sigma)$ to ensure that:

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

We define $X,\Sigma \simeq_{\Gamma_s,\Delta_s,\mathcal{I},\text{stmts}} \chi,\sigma$ as follows:

$$X,\Sigma \simeq_{\Gamma_s,\Delta_s,\mathcal{I},\text{stmts}} \chi,\sigma \iff \Gamma_s \vdash \Delta_s \quad \wedge \quad \Delta_s \models X,\Sigma$$

$$\quad \wedge \quad \llbracket \text{stmts} \rrbracket_{\Delta_s[0]} = \text{stmts}^\ell \quad \wedge \quad X,\Sigma \simeq_{\Delta_s,\mathcal{I}} \chi,\sigma$$

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

The theorems also use the relation $\beta \simeq_{\mathcal{I},\text{ps},\sigma} \gamma$ to express object equivalence between the high and low-level configurations. That is, the object with address $\beta$ in the high level corresponds to the standalone object with address $\gamma$ or the pooled object with index $\gamma$ in the low level. The pool the object belongs to in the low-level (if any) is derived from the stack $\sigma$ and the pool parameters $\text{ps}$. Full definition of $\beta \simeq_{\mathcal{I},\text{ps},\sigma} \gamma$ in §Appendix E.1

**Theorem 5 (Sound Translation).** For two well-formed and equivalent $\text{SHAPES}^h$ and $\text{SHAPES}^\ell$ configurations, a sequence of well-typed $\text{SHAPES}^h$ statements will
yield \( \text{SHAPES}^h \) configurations and return values equivalent to the \( \text{SHAPES}^l \) configurations and return values (respectively) yielded by the execution of a specialisation of the \( \text{SHAPES}^h \) statements into \( \text{SHAPES}^l \).

\[
\forall \mathcal{X}, \Sigma, \chi, \sigma, \Gamma_s, \Delta_s, \mathcal{I}, \text{stmts}, \text{stmts}^l, C, ps, \chi', \sigma'.
\]

If \( \mathcal{X}, \Sigma \simeq_{\Gamma_s, \Delta_s, \mathcal{I}, \text{stmts}} \chi, \sigma \land \Gamma_s[0] \vdash \text{stmts} : C(ps) \land \chi, \sigma, \text{stmts}^l \rightsquigarrow \chi', \sigma', \gamma \)
Then \( \exists \mathcal{I}', \chi', \Sigma', \beta. \)

\( \mathcal{X}, \Sigma, \text{stmts} \rightsquigarrow \mathcal{X}', \Sigma', \beta \land \mathcal{X}', \Sigma' \simeq_{\Delta_s, \mathcal{I}'} \chi', \sigma' \land \beta \simeq_{I', ps, \sigma} \gamma \)

\textbf{Proof.} By structural induction over the derivation \( \chi, \sigma, \text{stmts}^l \rightsquigarrow \chi', \sigma', \gamma. \)
See §\text{Appendix F}.

\textbf{Theorem 6} (Translation is complete). For two well-formed and equivalent \( \text{SHAPES}^h \) and \( \text{SHAPES}^l \) configurations, the specialised translation of a sequence of well-typed \( \text{SHAPES}^h \) statements will yield \( \text{SHAPES}^l \) configurations and return values equivalent to the \( \text{SHAPES}^h \) configurations and return values (respectively) yielded by the execution of the \( \text{SHAPES}^h \) statements.

\[
\forall \mathcal{X}, \Sigma, \chi, \sigma, \Gamma_s, \Delta_s, \mathcal{I}, \text{stmts}, \text{stmts}^l, C, ps, \mathcal{X}', \Sigma'.
\]

If \( \mathcal{X}, \Sigma \simeq_{\Gamma_s, \Delta_s, \mathcal{I}, \text{stmts}} \chi, \sigma \land \Gamma_s[0] \vdash \text{stmts} : C(ps) \land \mathcal{X}, \Sigma, \text{stmts} \rightsquigarrow \mathcal{X}', \Sigma', \beta \)
Then \( \exists \mathcal{I}', \chi', \sigma', \gamma. \)

\( \chi, \sigma, \text{stmts}^l \rightsquigarrow \chi', \sigma', \gamma \land \mathcal{X}', \Sigma' \simeq_{\Delta_s, \mathcal{I}'} \chi', \sigma' \land \beta \simeq_{I', ps, \sigma} \gamma \)

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

\textbf{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\cite{17}), alignment, and placement of auxiliary fields (e.g., mark word and klass pointer on the HotSpot VM\cite{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 \cite{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.

```java
class Professor {
  name: String;
  ssn: String;
}
class Student<pProf: [Professor<pProf>> {
  name: String;
  age: int;
  supervisor: Professor<pProf>;
}
layout ProfL: Professor = …;
layout StudentL: Student = …;
```

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 syntax 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, 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 §). Moreover, thanks to specialisation, a JIT is not necessary.

Clustering. As we showed in §, OP2 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 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 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, 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. attempt to reduce cache misses by automatically partitioning objects into pools of popular and unpopular objects. Later work 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<> 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


er.com/sloc

guide for assembly programmers and compiler makers, Copenhagen

c.duke.free.fr/coding/md5-specs-en.html, Wayback Machine:

[9] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight java: a minimal
core calculus for java and gj, ACM Transactions on Programming Lan-
503502.503505.

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

2018.

Conference on Object-Oriented Programming Languages, Applications,

analysis & transformation, in: Proceedings of the international symposium
on Code generation and optimization: feedback-directed and runtime opti-

57

URL http://drops.dagstuhl.de/opus/volltexte/2019/10809


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


[46] K. Nagel, M. Schreckenberg, A cellular automaton model for freeway traffic, 
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.

\[ C(C) \triangleq (pds fds mds) \text{ iff } \text{class } C(pds)\{fds mds\} \in \text{prog}[0] \]

\[ Ps(C) \triangleq p_1 \ldots p_n \text{ iff } C[C][0] = (p_1: \ldots, p_n: \ldots) \]

\[ Pb(C) \triangleq pbd_1 \ldots pbd_n \text{ iff } C[C][0] = (\ldots: pbd_1 \ldots, \ldots: pbd_n) \]

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

\[ M(C, m) \triangleq (t, x : t', \text{localPools}; \text{localVars}; \text{stmts}) \text{ iff } \]

\[ \{\text{def } m(x : t') : t \{\text{localPools}; \text{localVars}; \text{stmts}\}\} \in C[C][2] \]

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

\[ Fs(C) \triangleq f_1 \ldots f_n \text{ iff } C[C][1] = (f_1: \ldots, f_n: \ldots) \]

\[ L(L) \triangleq (C, fs_1 \ldots fs_n) \text{ iff } \]

\[ \text{(layout } L : C = \text{rec}\{fs_1\}; \ldots \text{rec}\{fs_n\}\} \in \text{prog}[1] \]

\[ O(L, f) \triangleq (i, j) \text{ iff } L(L) = (C, fs) \land fs[i, j] = f \]

\[ O(C, f) \triangleq i \text{ iff } Fs(C)[i] = f \]

\[ \mathcal{L}(L) \triangleq L(L)[0] \]

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

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.

\[ \vdash \text{prog} \text{ iff } (\forall \text{cd } \in \text{prog}[0]. \text{prog } \vdash \text{cd}) \land (\forall \text{ld } \in \text{prog}[1]. \text{prog } \vdash \text{ld}) \]

**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 $P(C) = p_1 \ldots p_n$, then $B(C, p_1) = [C(p_1, \ldots, p_n)]$.

The parameter list of all pool types must only contain parameters from the class’ pool parameter list (i.e. $P(C)$). 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 $\Gamma$ where the class’ formal pool parameters have their corresponding bounds as types. Moreover, $\Gamma$ is well-formed.

All the methods have a parameter and return type that is well-formed against the context $\Gamma$. Moreover, for each method, the corresponding method body is typeable against a context $\Gamma'$ which is an augmentation of $\Gamma$ and contains the types of `this` variable, local pool, and object variables of the method. Moreover $\Gamma'$ 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.

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

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

- $\Gamma \vdash e; \ \text{stmts } \iff \Gamma \vdash e \land \Gamma \vdash \text{stmts}$
- $\Gamma \vdash e \iff (e = \text{new } t) \rightarrow \Gamma \vdash t$

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 $C$.

$$\text{prog} \vdash \text{layout } L : [C] = \text{rec } \{f_{s_1}\} .. \text{rec } \{f_{s_n}\} \text{ iff}$$

$$\{\mathcal{F}_{s}(C)\} = \bigcup_{i \in 1 .. n} \{f_{s_i}\}$$

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\textsuperscript{\ell} definitions and operational semantics

Appendix C.1. Method lookup

SHAPES\textsuperscript{\ell} method lookup is defined as follows:

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

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 $\sigma$ addresses only the variable
on the top-most stack frame $\phi$. 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:

$$\text{path} \in \text{Path} ::= x | \text{path}.f$$

Given a specialised typing context $\Delta$, we would like to require that our paths
are well-formed with respect to $\Delta$. That is, $\exists \ell. \Delta \vdash \text{path} : t$ for a given path
path. Because for simplicity reasons the syntax of SHAPES as defined in §4 does not permit complex paths, i.e., \( x.f.g \), we define the following typing rules for paths:

\[
\begin{align*}
\Delta \vdash path : C(p) \\
\Delta \vdash x : \Delta(x) \\
\Delta \vdash path.f : F(C, f)[Ps(C)/ps]
\end{align*}
\]

**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...
to an object address. This variant is of the form \( \Delta, \chi, \Phi, path \leadsto \beta \).

Variable Path (HL) 

Null Path (HL)

\[
\begin{align*}
\Delta, \chi, \Phi, x \leadsto \Phi(x) & \quad \Delta, \chi, \Phi, path \leadsto null \\
\Delta, \chi, \Phi, path.f \leadsto null
\end{align*}
\]

Object Path (HL)

\[
\begin{align*}
\Delta \vdash path : C(\omega) & \quad \Delta, \chi, \Phi, path \leadsto \omega \quad \chi(\omega) = (C, \omega, \rho) \\
\Delta, \chi, \Phi, path.f \leadsto \rho(f)
\end{align*}
\]

Appendix D.2. Low-Level Paths

To evaluate paths in \( \text{SHAPES}^\ell \), 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 \( \Delta \)). This variant is, similarly, of the form \( \Delta, \chi, \phi, path \leadsto \gamma \).
Variable Path (LL) | Null Path (LL)
---|---
\[ \Delta, \chi, \phi, x \leadsto \sigma(x) \] | \[ \Delta, \chi, \phi, path \leadsto null \]

Heap Object Path (LL)

\[ \Delta \vdash path : C\langle np \cdot \_ \rangle \quad \Delta \vdash np : \text{None} \]
\[ \Delta, \chi, \phi, path \leadsto \alpha \quad \mathcal{O}(C, f) = i \]
\[ \Delta, \chi, \phi, path.f \leadsto \chi(\alpha + i) \]

Pool Object Path (LL)

\[ \Delta \vdash path : C\langle np \cdot \_ \rangle \quad \Delta \vdash np : L\langle \_ \rangle \]
\[ \Delta, \chi, \phi, path \leadsto k \quad \mathcal{O}(L, f) = (i, j) \quad N = |\mathcal{C}(L)[i]| \]
\[ \Delta, \chi, \phi, path.f \leadsto \chi(\phi(np) + i + 2) + N * k + j \]

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 \( I : Address \rightarrow Address^{\ell} \cup (Address^{\ell} \times Index^{\ell}) \)

For a given sequence of typing contexts \( \Delta s \), equivalence between a high level and a low level configuration via an injection \( I \) is represented using the notation \( \chi, \Sigma \simeq \Delta s, I \chi, \sigma \) 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:

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

\[ [\forall i. \forall p \in \text{dom}(\Delta s[i]).|\sigma[i](p) = \sigma'[i](p) = \text{null} \lor \mathcal{J}(\sigma[i]) = \sigma'[i] | \land \]

\[ \forall i, \text{path}.|\Delta s[i] \vdash \text{path} : C\langle p \cdot \_ \rangle \land \]

\[ \Delta s[i], \chi, \sigma[i], \text{path} \leadsto \gamma \land \]

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

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

---

68
\[\alpha, \gamma \simeq_{J} \alpha', \gamma' \iff \\begin{align*}
&[\gamma = \gamma' = \text{null}] \vee [\alpha = \alpha' = \text{null} \land J(\gamma) = \gamma'] \vee \\
&[\alpha \neq \text{null} \land \alpha' \neq \text{null} \land I(\alpha) = \alpha' \land I(\alpha, \gamma) = (\alpha', \gamma')]
\end{align*}\]

Appendix E.2. Correctness of Compilation Theorems

We will now prove that translation is meaning preserving.

**Definition 11.** Equivalence between high-level and low-level addresses under an injection \(I : \text{Address} \rightarrow \text{Address}^\ell \cup (\text{Address}^\ell \times \text{Index}^\ell)\) is defined as:

\[\beta \simeq_{I, ps, \sigma} \gamma \iff \begin{align*}
&[\beta = \gamma = \text{null}] \vee [\sigma(ps[0]) = \text{null} \land I(\beta) = \gamma] \vee [\sigma(ps[0]) = \alpha \neq \text{null} \land I(\beta) = (\alpha, \gamma)]
\end{align*}\]

Equivalence between a high-level and a low-level configuration is defined as:

\[\chi, \Sigma \simeq_{\Delta_s, I} \chi', \Sigma' \iff \begin{align*}
&\forall p, i. (\Sigma[i](p) = \text{none} \land \sigma[i](p) = \text{null}) \lor I(\Sigma[i](p)) = \sigma[i](p) \land \\
&\forall np, i, \text{path}, C, \beta, \gamma. [\Delta s[i] \vdash \text{path} : C(np, \ldots) \land \\
&\Delta s[i], \chi, \Sigma[i], \text{path} \rightsquigarrow \beta \land \Delta s[i], \chi', \Sigma[i], \text{path} \rightsquigarrow \gamma \rightarrow \beta \simeq_{I, np, \sigma} \gamma]
\end{align*}\]

Appendix E.2.1. Garbage collection

We use the notation \(\chi, \sigma \simeq_{\Delta_s, J} \chi', \sigma'\) to indicate that two low-level configurations \(\chi, \sigma\) and \(\chi', \sigma'\) are equivalent under the injection \(J\) for a given sequence of specialised contexts. The definition of equivalence under \(J\) is presented in Definition 10 (found in §Appendix E.1).

The definition ensures that if a path \(\text{path}\) yields two standalone or pool-allocated object addresses in the two configurations, then a mapping between them must exist and if both are allocated in a pool, then a mapping between the corresponding pool addresses must also exist. Furthermore, we require that all variables in a stack frame corresponding to pools must have a mapping between them (assuming they point to pools).

We now define the following theorem, which states that evaluating the same statement sequence under two equivalent low-level configurations reaches two low-level configurations that are also equivalent. This theorem allows us to reason that performing a GC on \(\text{SHAPES}^\ell\) 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 $\chi_1, \sigma_1$ and $\chi_2, \sigma_2$ be two low level configurations, let $\Delta_s$ be their corresponding typing contexts and let $J$ be an injection such that $\chi_1, \sigma_1 \simeq_{\Delta_s, J} \chi_2, \sigma_2$. Then, for a sequence of statements $\text{stmts}$ such that $\Delta_s[0] \vdash \text{stmts} : C(p_1 .. p_k)$, if $\chi_1', \sigma_1', \gamma_1$ and $\chi_2', \sigma_2', \gamma_2$ exist such that:

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

And $\Delta' = \text{this}: C\langle np_1 .. np_k \rangle, np_1 : \Delta s[0](np_1), .., np_n : \Delta s[0](np_n)$, then there exists an injection $J'$ such that $\chi_1', \phi_1' \cdot \sigma_1' \simeq_{\Delta, J'} \chi_2', \phi_2' \cdot \sigma_2'$. where

$$\phi_1' = [\text{this} \mapsto \gamma_1, np_1 .. np_k \mapsto \sigma_1'(np_1 .. np_k)]$$

$$\phi_2' = [\text{this} \mapsto \gamma_2, np_1 .. np_k \mapsto \sigma_2'(np_1 .. np_k)]$$

Appendix F. Proof sketches

Proof (Proof of 1). For this proof, we will make use of the utility predicate $\text{pdb}_1 \simeq \text{pdb}_2$ that is defined as follows:

$$\frac{\text{pdb}_1 \simeq \text{pdb}_2}{\text{pdb}_2 \simeq \text{pdb}_1} \quad \frac{\text{pdb}_1 \simeq \text{pdb}_2 \quad \text{pdb}_1 \simeq \text{pdb}_2}{\epsilon \simeq \epsilon}$$

$$\frac{\text{pdb}_1 \simeq \text{pdb}_2}{\epsilon \simeq \epsilon}$$

Thus we can redefine $\forall i. \Gamma \vdash \text{ps}[i] : \text{pdb}_i$ as $\Gamma(\text{ps}) \simeq \text{pdb}$. Let:

$$qs = P_s(C) \quad qbs = P_b(C)$$

$$rs = P_s(C') \quad rbs = P_b(C')$$

From Rule OBUYTYPEWF and from the above definition of $\simeq$, in order to show that $\Gamma \vdash C'(\text{ps}'[qs/ps])$, we need to show that:

$$\Gamma(\text{ps}'[qs/ps]) \simeq rbs[rs/\text{ps}'[qs/ps]] \quad (F.1)$$
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 \( \Gamma_C \) is the environment used to compile class \( C \), then:
\[
\Gamma_C(ps') \simeq rbs[rs/ps'] \tag{F.3}
\]

We now define the following:
\[
Qs = qs \cdot \text{none} \quad QBs = qbs \cdot \text{None} \\
Rs = rs \cdot \text{none} \quad RBs = rbs \cdot \text{None} \\
Ps = ps \cdot \text{none} \\
Ps' = ps' \cdot \text{none}
\]

For convenience, let \( \Gamma(\text{none}) = \text{None} \). Because \( \text{none} \) is the last pool parameter in \( Ps, Ps', Qs, Rs \) (hence the substitution e.g., \( [Qs/Ps] \) will replace \( \text{none} \) with \( \text{none} \)), it will also hold from \( \text{F.2}, \text{F.3} \) 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 \( \text{F.1} \) holds.

Let \( \text{Im} f \) be the image of function \( f \).

Well-formedness of \( prog \) implies that each of the parameters in \( ps' \) will be either \( \text{none} \) 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 \( \text{prog} \) is well-formed, each pool parameter from \( Ps' \) will also agree to their pool bound defined in \( C \), hence:

\[
\Gamma(Ps') \simeq QBs[\text{Im } \sigma]
\]

Thus:

\[
RBs[Rs/Ps'][\text{dom}(\sigma)] \simeq QBs[\text{Im } \sigma] \quad (F.7)
\]

Therefore:

\[
\Gamma(Ps'[Qs/Ps]) = \Gamma(Ps[\text{Im } \sigma]) \\
\simeq QBs[Qs/Ps][\text{Im } \sigma] \quad \text{From (F.4)} \\
\simeq RBs[Rs/Ps'][\text{dom}(\sigma)][Qs/Ps] \quad \text{From (F.7)} \\
= RBs[Rs/Ps'][Qs/Ps][\text{dom}(\sigma)] \quad \text{By structure} \\
= RBs[Rs/Ps'[Qs/Ps]] \quad \text{Since } |\text{dom}(\sigma)| = |Ps'| = |Rs|
\]

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** (\( \text{this} \)), **NEW** (\( \text{new } t \)): From the definition of well-formed program (§Appendix B).
- **FIELD READ** (\( x.f \)): Let \( \Gamma \vdash x : C(ps) \). Then we apply Lemma 2 given that \( \vdash \Gamma, \Gamma \vdash C(ps) \), and \( \Gamma_C \vdash F(C, f) \), where \( \Gamma_C \) 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 \(\mathcal{X} \vdash \Sigma(x) : C(\Sigma(ps))\) 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 \(\beta\) such that \(\mathcal{X}' \vdash \beta : C(\Sigma'(ps))\), therefore assigning \(\beta\) to \(x\) will not break well-formedness of the configuration or break the weak agreement requirement for \(\beta\).

- **Rule New Object**: We augment the heap \(\mathcal{X}\) with a new object \(\omega\); we only need to show that \(\mathcal{X}' \vdash \omega \prec C(\Sigma(ps))\) 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 \(f_s = \mathcal{F}_s(C)\) to **null**, we only need to show that \(\mathcal{X} \vdash \Sigma(ps[0]) : [C(\Sigma(ps))]\).

Because \(\Gamma s[0] \vdash C(\Sigma(ps))\), it holds from **ObjTypeWF** that \(\Gamma s[0] \vdash ps[0] :: [C(ps)]\), hence from the definition of well-formed configuration, it will hold that \(\mathcal{X} \vdash \Sigma(ps[0]) : [C(\Sigma(ps))]\), regardless of whether \(ps[0]\) is **none**, corresponds to a pool with layout \(L\) such that \(\mathcal{C}(L) = C\) or a formal pool parameter from \(Ps(C)\). And because \(\mathcal{X}'\) is an augmentation of \(\mathcal{X}\), it will also hold that \(\mathcal{X}' \vdash \Sigma(ps[0]) : [C(\Sigma(ps))]\).

- **Rule Object Read**: Configuration is not mutated, hence well-formedness of the configuration is preserved.

Suppose that \(\Gamma \vdash x : C'(ps')\), hence \(\Gamma \vdash x.f : \mathcal{F}(C', f)[Ps(C')/ps']\) and suppose that \(C(ps) = \mathcal{F}(C', f)[Ps(C')/ps']\). If \(\mathcal{X}(\omega) = (C, \pi s, \rho)\) and \(\rho(f) = \beta\), then we want to show that \(\mathcal{X}' \vdash \beta : C(\Sigma(ps))\).
From the definition of well-formed configuration, it holds that $\mathcal{X} \models \Sigma(x) \land C'(\Sigma(ps'))$, hence $\mathcal{X} \models \beta : F(C', f)[P s(C')/\Sigma(ps')]$.

Because $C'(ps) = F(C', f)[P s(C')/ps']$, each pool parameter $ps[i]$ is either \textbf{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 $F$ can be \textbf{none} or come from $P s(C')$ and the pool parameters $P s(C')$ are substituted by $ps'$.
Therefore $C(\Sigma(ps)) = F(C', f)[P s(C')/\Sigma(ps')]$, hence $\mathcal{X}' \models \beta : C(\Sigma(ps))$.

- \textbf{Rule \texttt{Object Write}:} Similar to \textbf{Rule \texttt{Object Read}}, it can be shown that if $\Gamma \vdash x.f : C(ps)$, $\mathcal{X}(\omega) = (C, \pi s, \rho)$, then $\mathcal{X} \models \rho(f) : C(\Sigma(ps))$. From the definition of well-formed configuration, it holds that $\mathcal{X}' \models \Sigma(x') : C(\Sigma(ps))$.

Because $\rho(f)$ and $\Sigma(x')$ have the same runtime type $C(\Sigma(ps))$, the assignment will maintain the well-formedness of the new configuration $\mathcal{X}', \Sigma$ and it will hold that $\mathcal{X}' \models \Sigma(x') : C(\Sigma(ps))$.

- \textbf{Rule \texttt{Statement Sequence}:} By structural induction over the derivation of $\texttt{stmts}$ and, subsequently, the derivation of $e$.

- \textbf{Rule \texttt{Method Call}:} When invoking a method, we construct a new stack $\Sigma'$ 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 $\Gamma'$ 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 $\Sigma'$ weakly agree to their “appropriate” runtime types and that the new pools in $\mathcal{X}'$ strongly agree to their “appropriate” runtime types.

Let $\Phi$ and $\Phi'$ be the top stack frame of $\Sigma$ and $\Sigma'$, respectively, and that $\Gamma'$ corresponds to the environment of $\Phi$. Then, with respect to the local variables of $\Phi'$:

- For the \texttt{this parameter}, it holds that $\mathcal{X}' \models \Phi'(\texttt{this}) : C(\Phi'(P s(C)))$, given that it holds that $\Gamma' \vdash \texttt{this} : C(P s(C))$. Suppose that the variable $x$ corresponding to \texttt{this} in $\Phi'$ has the type $C(ps)$ under $\Gamma$. Then, since $\mathcal{X}'$ augments $\mathcal{X}$, it holds that $\mathcal{X}' \models \Phi(x) : C(\Phi(ps))$. 
From the operational semantics of Method Call, we have that \( \Phi(x) = \Phi'(\text{this}) \) and \( \Phi(ps) = \Phi'(\text{Ps}(C)) \), hence we conclude that \( \mathcal{X}' \vdash \Phi'(\text{this}) : C(\Phi'(\text{Ps}(C))) \).

- 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 \( \Gamma \vdash x : C(ps), \Gamma' \vdash \text{this} : C(\text{Ps}(C)) \), and \( \Gamma' \vdash x'' : C'(ps') \). Then, it holds that \( \Gamma \vdash x' : C'(ps')[C(C)/ps] \).

Since our program is well-formed, each pool parameter \( ps'[i] \) can be either \text{none} or originate from \( \text{Ps}(C) \) (i.e., there exists \( j \) such that \( ps'[i] = \text{Ps}(C)[j] \)), each of the pool parameters of the type of \( x'' \) is either \text{none} or originates from \( ps \), therefore it holds that \( \mathcal{X}' \vdash \Phi(x') : C'(ps')[C(C)/\Phi(ps)] \). Thus, it holds that \( \mathcal{X}' \vdash \Phi'(x'') : C'(ps')[C(C)/\Phi'(\text{Ps}(C))] \), given that \( \Phi(x') = \Phi'(x'') \), \( \Phi(ps) = \Phi'(\text{Ps}(C)) \). And given the aforementioned limitation on \( ps' \), we reach the conclusion that it holds that \( \mathcal{X}' \vdash \Phi'(x'') : C'(\Phi'(ps')) \).

- Weak agreement on pool bounds is shown in a manner similar to that of the method argument.

- Object local variables (declared via \text{locals}) are initialised to \text{null}, hence weak agreement holds.

- If a Pool local variable \( p \) (declared via \text{pools}) has type \( L(ps) \) under \( \Gamma' \), then \( \mathcal{X}' \vdash \Phi'(p) : C(\Phi'(ps)) \) 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 \( \Gamma' \vdash p : L(ps) \) and let \( C = A(L) \). We need to show that for every \( i \) it holds that \( \mathcal{X}' \vdash \Phi'(ps[i]) : B(C, \text{Ps}(C)[i])[\text{Ps}(C)/\Phi'(ps)] \) in order to show that \( \mathcal{X}' \vdash \Phi'(p) : L(\Phi'(ps)) \).

If \( \Phi'(ps[i]) = \text{none} \), then weak agreement easily holds. Otherwise, since \( \Gamma' \vdash L(ps) \), it holds that \( \Gamma' \vdash ps[i] : B(C, \text{Ps}(C)[i])[\text{Ps}(C)/ps] \). Because \( \Phi'(ps[i]) \neq \text{none} \), \( ps[i] \neq \text{none} \), hence from the definition of \( \Gamma' \), \( ps[i] \) can only adhere to the bound \( B(C, \text{Ps}(C)[i])[\text{Ps}(C)/ps] \).
Since we have shown that all variables in $\Phi'$ weakly agree to their “appropriate” runtime type, it holds that $\lambda' \vdash \Phi'(ps[i]) : \mathcal{B}(C, \mathcal{P}s(C)[i])(\mathcal{P}s(C)/\Phi')$.

Therefore $\Gamma' \cdot \Gamma s \vdash \lambda', \Sigma'$. By structural induction over the statement sequence $stmts$, we deduce that $\Gamma' \cdot \Gamma s \vdash \lambda''', \Sigma'''$, where $\lambda''', \Sigma'''$ is the configuration resulting from the evaluation $stmts$.

Let $\beta$ be the value yielded by evaluating $stmts$. Then if $C'(ps')$ is the return type of method $m$, then by structural induction over $stmts$ it will also hold that $\lambda''' \vdash \beta : C'((\Sigma'''(ps'))).$ We want to show that if $\Gamma \vdash x.m(x'') : C'(ps'')$, then $\lambda''' \vdash \beta : C'((\Sigma'''[1](ps''))$.

Indeed, each pool parameter in $ps'$ can be either $\text{none}$ or originate from $\mathcal{P}s(C)$. As such, each pool parameter in $ps''$ can be either $\text{none}$ 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 $\lambda''' \vdash \beta : C'((\Sigma'''[1](ps''))$.

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 $\lambda', \Sigma \simeq_{\Delta, I} \chi, \sigma$ and it holds that $\chi, \sigma \simeq_{\Delta, J} \chi', \sigma'$, then $\lambda', \Sigma \simeq_{\Delta, I'} \chi, \sigma$, where $I' = J \circ I$.

**Proof.** By structural induction over the paths that are well-typed under each of the specialised contexts $\Delta$.

**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 $I' = I$.
- **Rule Variable:** High-level configuration need not be altered, hence $I' = I$.
- **Rule Assignment:** We obtain a new high-level configuration and injection $I''$ by structural induction over the derivation of $e$. $I''$ will also map the high-level value yielded ($\beta$) to its low-level counterpart ($\gamma$), thus assignment will not violate equivalence, hence $I' = I''$. 

76
• 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., \( p = \text{none} \) or \( \Delta(p) = \text{None} \)): \( I' \) extends \( I \) by mapping the address of the new high-level object to the address of the low-level one.

- **New Pooled Object** (i.e., \( \Delta(p) = L(\cdot) \)): \( I' \) extends \( I \) 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 \( I' = I \). 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 \( I' = I \).

• 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 \( I' \) 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; \( I' \) will extend \( I \) to map them to their low-level counterparts. Moreover, to obtain the new high-level configuration, we have to create the new stack frame \( \Phi \) and populate it with the method’s parameters and initialise the local variables. The values of these variables are already equivalent under \( I \), hence \( I' \) as well. Then, by structural induction over the body of the method, we obtain a new injection \( I'' \). \( I'' \) will also map the high-level value yielded (\( \beta \)) to its low-level counterpart (\( \gamma \)).
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 $I'$, 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 $I'$ 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 involves invoking a specialisation $m'$ of $m$. Suppose that $\Delta s[0] \vdash x : C(ps)$. Then, the types pool parameters $ps$ will be specialised (i.e., they will be $L(\_)$ or $\text{None}$). Method $m'$ is compiled under an environment $\Delta$ such that for all $i$, if $\Delta s[0](ps[i]) = L(\_)$ or $\Delta s[0](ps[i]) = \text{None}$, then

78
\[ \Delta(Ps(C)[i]) = L(\_) \text{ or } \Delta(Ps(C)[i]) = \textbf{None}, \text{ respectively. Therefore, if } \Phi \text{ is the stack frame corresponding to } m, \text{ then } \Delta \cdot \Delta s \models \mathcal{X}, \Phi \cdot \Sigma. \] 

The proof continues by structural induction on \( stmts \), in a similar manner to that of Theorem \( \text{5} \) for Rule \text{Method call}.

**Proof (Proof of Theorem \( \text{7} \)).** We prove this theorem on a case-by-case basis:

- **Rules Value, Rule Variable, Object Read:** It holds that \( J' = J \), since they do not modify the configuration and the references returned are either \text{null} or they both correspond to reachable objects.

- **Rule Assignment:** We obtain two new low-level configurations that are equivalent over an injection \( J \) by structural induction over the derivation of \( e \). \( J \) 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 \( J \) to map the address of the object in the first configuration to the one in the second configuration.
  - **New Pooled Object:** We extend \( J \) 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 \( J' = J \). This is because \( J \) maps the objects referenced by \( x, x' \) in first configuration to their counterparts in the second configuration, thus field assignment will not break the isomorphic property of the configurations.

- **Rule Statement Sequence:** Garbage collection is performed on two initial configurations, yielding two new configurations that are each equivalent to the initial ones. Thus, the two new configurations are transitively equivalent. The proof is then completed by structural induction over the sequence of statements.
- Rule 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.
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 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.

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)

---

6Hosted in [https://github.com/prg-titech/dynasoar/blob/9dab3900c142aa8ee41966647bd97ef3d035768c/example/traffic/baseline_pos/traffic.cu](https://github.com/prg-titech/dynasoar/blob/9dab3900c142aa8ee41966647bd97ef3d035768c/example/traffic/baseline_pos/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.
Appendix G.4. Doors

Consider a list of characters placed in a 2D space; each character belongs to one of two teams (Red team vs. Blue team). Now, consider a list of automatic doors in the same space; doors also have an allegiance and they can only open when a character of the same allegiance is in their proximity. 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 performance of checking 100 randomly generated doors and characters. We run our case study with 50%, 70%, and 90% of characters and doors belonging to the Red team. We assume an AoS layout in both cases. Figure G.35 presents the relevant results. We observe the following speedups of medians for each category:

---

Inspired from material from the following URL: https://web.archive.org/web/20190517194356/https://deplinenoise.files.wordpress.com/2015/03/gdc2015_alfredrikssoon_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, 161-165, 227-228). 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, 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 \textsc{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). \textsc{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 \textsc{Shapes++} implementation amounts to 238 SLoC. These numbers were calculated using SLOCCount \cite{SLOCCount}.

The \textsc{Shapes++} code is presented below:

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

// Constants used for computation, left as-is
const gam: double = 1.4f;
const gm1: double = gam - 1.0f;
const cfl: double = 0.9f;
const eps: double = 0.05f;
const mach: double = 0.4f;
const alpha: double = 3.0f * atan(1.0f) / 45.0f;
const p: double = 1.0f;
const r: double = 1.0f;
const u: double = sqrt(gam * p / r) * mach;
const e: double = p / (r * gm1) + 0.5f * u * u;
const qinf: double[4] = { r, r * u, 0.0f, r * e };

// Data structures present in all reimplementation files, each of the three layouts corresponds to the AoS, Mixed, and SoA layouts used in the actual C++ code
class Edge<pe: Edge<pe, pc, pn>>,
  pc: [Cell<pc, pn>], pn: [Node<pn>] {  
  pedge: Node<pn>[2];  
  pecell: Cell<pc, pn>[2];
```
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]);
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.
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;

        ri = 1.0f / q[0];
        u = ri * q1;
        v = ri * q2;
        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.
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 x30 = edges[i].pedge[2].p_x[0];
        var x31 = edges[i].pedge[2].p_x[1];
        var x40 = edges[i].pedge[3].p_x[0];
        var x41 = edges[i].pedge[3].p_x[1];
    }
```cpp
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 nu = 0.5f * (adt1 + adt2) * eps;
var f: double;
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];
    // 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.
```

```cpp
}  // std::vector<Color> as an argument.
```
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 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;
}

return rms;

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>], 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());
    }
    rms = sqrt(rms / (double) pc.size());
}
The SHAPES++ code is presented below:

```cpp
const double gam = 1.4;
const double gm1 = gam - 1.0;
const double gm1i = 1.0 / gm1;
const double* wtg1 = ...;
const double* xi1 = ...;
const double* Ng1 = ...;
const double* Ng1_xi = ...;
const double* wtg2 = ...;
const double* Ng2 = ...;
const double* Ng2_xi = ...;
const double minf = 0.1;
const double m2 = minf * minf;
const double freq = 1;
const double kappa = 1;
const double nmode = 0;
const double mfan = 1.0;

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.
```javascript
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_i = double = 0;
    var N_x: double[8];
    for (var m = 0; m < 4; m++)
        N_x[m] = det_x_i * 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];
    b = 0;
    b += Ng2_xi[4 * i + 0] * e.pcell[0].x[1];
    b += Ng2_xi[4 * i + 1] * e.pcell[1].x[1];
    b += Ng2_xi[4 * i + 2] * e.pcell[2].x[1];
    b += Ng2_xi[4 * i + 3] * e.pcell[3].x[1];
    for (var m = 0; m < 4; m++)
        N_x[4 + m] -= b * Ng2_xi[4 * i + m];
}
```

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, gm1);

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]);


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]);
    }
}

// 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.p_resm;
        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 sp MV() { 
  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[2].p_V[0] += e.K[2] * e.pcell[0].p_P[0]; 
    e.pcell[1].p_V[0] += e.K[3] * e.pcell[3].p_P[0]; 
    e.pcell[3].p_V[0] += e.K[3] * e.pcell[1].p_P[0]; 
    e.pcell[2].p_V[0] += e.K[4 + 1] * e.pcell[1].p_P[0]; 
    e.pcell[2].p_V[0] += e.K[4 + 2] * e.pcell[0].p_P[0]; 
    e.pcell[2].p_V[0] += e.K[8 + 2] * e.pcell[1].p_P[0]; 
    e.pcell[1].p_V[0] += e.K[8 + 3] * e.pcell[3].p_P[0]; 
  } 
} 

// Corresponds to the dot PV() method used in all implementations. 
// Corresponds to the dot PV() method used in all implementations. 
// Corresponds to the update UR() method used in all implementations. 
// Corresponds to the update UR() method used in all implementations. 
// Corresponds to the update R() method used in all implementations. 
// Corresponds to the update R() method used in all implementations. 
// Corresponds to the update P() method used in all implementations. 
// Corresponds to the update P() method used in all implementations. 
// Corresponds to the update() method used in all implementations. 
// Corresponds to the update() method used in all implementations.
```java
var rms: double = 0;
for (var e: pn) {
  e.p_phib = e.p_U;
  e.p_resm = 0.0;
  rms += e.p_U * e.p_U;
}
return rms;
```

```java
// 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>updateR(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 126).
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 weights 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:
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++];
  }
97 base_pos.y = frame_components[idx++];
98 if (flags & ANIM_ZPOS)
99 base_pos.z = frame_components[idx++];
100 if (flags & ANIM_XQUAT)
101 base_orient.x = frame_components[idx++];
102 if (flags & ANIM_YQUAT)
103 base_orient.y = frame_components[idx++];
104 if (flags & ANIM_ZQUAT)
105 base_orient.z = frame_components[idx++];
106 base_orient.gen_w();
107
108 var parent = it.parent;
109 if (parent == null) {
110 it.pos = base_pos;
111 it.orient = base_orient;
112 continue;
113 }
114
115 var rotated_pos = parent.orient.rotate(base_pos);
116 it.pos = rotated_pos + parent.pos;
117 it.orient = parent.orient * base_orient;
118 it.orient.normalize();
119 }
120 }
121
122 // Corresponds to animate_weights
123 // and Joint::animate_my_weights in
124 // stickmen/src/anim/anim.cpp.
125 def animate_weights() {
126 for (var j = this; j != null; j = j.next)
127 for (var it = j.begin; it != j.end; it++)
128 it.pos = j.orient.rotate(it.initialpos) + j.pos;
129 }
130 }
131 // Corresponds to JointInfo in
132 // stickmen/include/anim/anim.h
133 class JointInfo {
134 base_pos, base_quat: vec3;
135 flags: i8;
136 }
137 // Corresponds to WeightAos, WeightMixed,
138 // and WeightSoa in
139 // stickmen/include/anim/anim.h
140 class Weight<wp: [Weight<wp>]> {
141 next: Weight<wp>;
142 pos, initialpos: vec3;
143 bias: float;
144 }
145
146 layout JointAos: Joint =
147 rec(orient, glob_orient, pos, parent, next, first);
148
149 layout WeightAos: Weight = rec(next, pos, initialpos, bias);  
150 layout WeightMixed: Weight = rec(next) + rec(pos)
151 + rec(initialpos) + rec(bias);  
152 layout WeightSoa: Weight = rec(next)
153 + rec(pos.x) + rec(pos.y) + rec(pos.z)
154 + rec(initialpos.x) + rec(initialpos.y) + rec(initialpos.z)
155 + 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].

```cpp
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 TARGET_RATIO: float = 0.002f;
10 const CAR_ALLOCATION_RATIO: float = 0.02f;
11 const SLOW_DOWN_PROBABILITY: float = 0.2f;
12 13 var TRAFFIC_LIGHT_PHASE: int = 5;
14 15 // Enum used for cell flags (instead of explicit constants)
16 enum CellFlags: i8 {
17  Producer = 1 << 0,
18  HasCar = 1 << 1,
19  IsTarget = 1 << 2,
20  ShouldOccupy = 1 << 3,
21 22 );
23 24 // Corresponds to Cell and CellPool classes (former uses AoS, latter uses the CellMixed layout)
25 class Cell<cp: [Cell<cp>]> {
26  rand_state: Rng;
27  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;
}

TrafficLight<Tp, cp> {
cells: Cell<cp>[MAX_DEGREE];
um_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(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 =

100
```python
layout TrafficLightMixed: TrafficLight = rec {cells, num_cells, timer, phase_time, phase};

// Corresponds to create_cars() (both overloads)
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)
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)
void prepare_paths()
{
    var speedup_path_dist: uniform_dist<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 & 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 & CellFlags::HasCar) != 0)
        break;

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

e.car_velocity = e.num_car_path;

if (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 & CellFlags::HasCar) != 0) {
        distance--;
        e.car_velocity = distance;
        break;
    } else {
        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
                e.car_velocity = distance;
                break;
        } else {
            e.car_velocity = distance;
            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;
            } else 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:
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 algorithm (Line 58) 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:

```cpp
// 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, 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, 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, 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 RateSoa: [Rate] = rec{
  date, USD, GBP
  + rec{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};
```

105
// 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:
  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 = lo + (hi - lo) / 2;
    if (pl[mid].date < date)
      lo = mid + 1;
    else if (pl[mid].date > date)
      hi = mid - 1;
    else
      return pl[mid];
  }
}


```java
  98 }
  99    return null;
 100 }
```