Safely Abstracting Memory Layouts

Juliana Franco[1] Alexandros Tasos[2] Tobias Wrigstad[3] Sophia Drossopoulou[2] Susan Eisenbach[2]

[1] Microsoft Research Cambridge [2] Imperial College London [3] Uppsala University
juliana.franco@microsoft.com

Abstract

Modern architectures require applications to make effective use of caches to achieve high performance and hide memory latency. This in turn requires careful consideration of placement of data in memory to exploit spatial locality, leverage hardware prefetching and conserve memory bandwidth. In unmanaged languages like C++, memory optimisations are common, but at the cost of losing object abstraction and memory safety. In managed languages like Java and C#, the abstract view of memory and proliferation of moving compacting garbage collection does not provide enough control over placement and layout.

We have proposed SHAPES, a type-driven abstract placement specification that can be integrated with object-oriented languages to enable memory optimisations. SHAPES preserves both memory and object abstraction. In this paper, we formally specify the SHAPES semantics and describe its memory safety model.

1 Introduction

Modern computers use hierarchies of memory caches to hide memory latency [20]. Accessing data from the bottom of the hierarchy — from main memory — is often an order of magnitude slower than reading from the top — from the fastest memory cache. Whenever the CPU needs to read from a particular memory location, it first tries to read from the top-most cache. If it succeeds, then the data is delivered at very little cost. Otherwise, a *cache miss* has occurred, and the CPU tries to obtain the data from the next cache level. In the worst case scenario, data must be fetched from main memory. This commonly causes the data — and adjacent data — to be cached. How much adjacent data is cached and how much data can be cached at what level varies across different hardware. Caches typically have sizes in kilobytes or a few megabytes, meaning that bringing only relevant data into cache is important for proper cache utilisation.

In programs whose performance is memory-bound, reorganising data in memory to reduce the number of cache misses can have enormous performance impact [2, 20]. Writing cache-friendly code typically amounts to allocating contiguously data that is accessed together and in patterns recognisable to the *hardware prefetcher*, allowing it to anticipate a program's data needs ahead-of-time. As a concrete example, in a language like C, one can allocate an array of structures (not pointers to structures), and keep all the objects of the same data structure in that array, in the order in which the objects are most frequently accessed. Thus, when an object is fetched to cache, the next one to be read, is potentially fetched as well. To improve cache utilisation, programmers often *split objects* across multiple arrays, in order to keep the *hot fields* — those most often used — of different objects together. This is called transforming an array of structures into a structure of arrays. Both layouts are depicted in Figure 1. As a consequence of this representational change, a complex datum is no longer possible to reference by a single pointer.

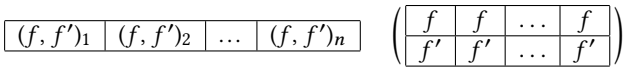

Figure 1. Array of structs (left) vs. struct of arrays (right). (f, f') denotes a struct (object) with fields f and f'. Left, each *cell* is an object. Right, each *column* holds the data for one object.

Optimisations like the one sketched above are common in low-level languages such as C or C++, where programmers have control over where and how memory is allocated. Unfortunately, manipulating data layouts in memory is complex, error-prone, and pollutes the logic of the program with layout concerns.

In managed languages like Java and C#, the abstract level at which memory is handled, as well as the presence of moving garbage collectors, make many memory optimisations impossible. To this end, we have proposed a type-driven approach to abstractly express layout concerns, called SHAPES [6], whose aim is to allow the application of layout optimisations in high-level, managed, object-oriented languages. SHAPES protects the object abstraction, allowing an object to be referenced by — and manipulated via – a single pointer, regardless of its physical representation.

Contributions In previous work, we have introduced the SHAPES idea, and recap the key moving parts in § 2. In this paper, we make the following contributions:

- We formalise SHAPES (§ 3) through a high-level object-oriented calculus where classes are parameterised by layout specifications. The formalism is an important component of proving soundness of *uniform access* — allowing programmers to write `x.f` to access the field `f` of the object *o* pointed to by `x`, regardless of how *o* is laid out in memory. We specify the dynamic semantics (§ 3.1) as well as the static semantics (§ 3.2).
- We define the main invariants of SHAPES in terms of type safety and memory safety (§ 4).

§ 5 discusses related work and § 6 concludes.

2 Getting into SHAPES

The SHAPES vision gives the programmer control of how data structures are laid out in memory in object-oriented, managed programming languages. Although programmers have control over how their data structures are allocated, memory is still abstract, and all operations are type and memory safe.

The SHAPES approach makes classes parametric with *layouts* — abstract regions (*pools*) which collect their objects together in physical memory, optionally using some form of *splitting strategy* to keep hot fields together in memory. The design delays the choice of the physical representation of a data structure to instantiation-time, rather than declaration-time. Thus, it is possible to reuse the same

Formal Techniques for Java-like Programs, 2018, (FTfJP)
2018.

data structure declaration with multiple layouts. Reflecting layout choices in types thus serves the dual purpose of separating layout concerns from business logic and guiding compilers' generation of efficient code for allocating and accessing data. It is also key to a uniform access model (*e.g.* x.f), regardless of how an object may be laid out in memory. In a typical statically typed OO language, x.f is translated by a compiler into loading a value at the address pointed to by x plus the offset of f in the type of x. Because the same x.f may manipulate x's that point to objects with different layout, we must take care to propagate the layout information to ensure the correct code is emitted, to ensure memory safety.

2.1 Using SHAPES for Improved Cache Utilisation

As a concrete example, Listing 1 is a partial program with a list of Students, with 1 000's of elements at run-time. (For simplicity, each student holds a pointer to the next student.)

```
class Student {
  age: int
  supervisor: Professor
  next: Student
  // ...
  def getAge(): int {
    return age
  }
}

var list = new Student
```

Listing 1. A linked list.

```
1  class Student<p> {
2    age: int
3    supervisor: Professor
4    next: Student<p>
5    // ...
6    def getAge(): int {
7      return age
8    }
9  }
10 layout L : [Student] = rec { age, next }, *
11 pool P : L
12 var list = new Student<P>
```

Listing 2. Listing 1 using SHAPES.

Iterating over a list of students to calculate the average age in a language like Java involves dereferencing many Student-pointers to get the age and next fields. However, because of how data is loaded into the cache (outlined in § 1), in addition to these relevant fields, the supervisor field and all adjacent data mapped to the same cache line will be loaded as well. If that data is not another student in the list, it is irrelevant to performing this calculation.

For improved cache utilisation, we would like to:

a) only load the age and next fields into the cache; and
b) store Student objects from the same list adjacent in memory, with no interleaving from unrelated objects.

Listing 2 shows Listing 1 using SHAPES. We accomplish b) by introducing a *pool*—a contiguous region of storage—for holding Student objects and use this pool to hold all the objects of our linked list (and nothing else). This effectively stores our students like the left of Figure 1. Listing 2 shows how the Student class is parametrised by the pool parameter (Lines 1 & 4). The pool is created on Line 10 and connected to the list on Line 11.

To accomplish a), we alter the layout of objects in the pool to use a structure of array storage. The layout is declared on Line 9 and used in the pool declaration on Line 10. Our students are now stored like the right of Figure 1. Note how the layout is orthogonal from the Student class.

Finally, if the order of the students in the pool (mostly) matches the order of the list, iterating over the list will result in (mostly) regular load patterns with even strides that will be detected by a prefetcher to bring data into memory ahead-of-time. Unless this order-alignment happens by construction, a pool-aware moving compacting garbage collector can be used to create it. Such a collector will compact respecting pool boundaries, and it is possible to influence moving semantics on a per-pool basis.

2.2 SHAPES in a Nutshell

SHAPES adds pools, layout declarations and parameterises classes and types with pools to a Java-like language. The number of pools is not fixed and pools are created at run-time. Objects may be allocated in pools or in a "traditional heap." Layout specifications specify how objects of a certain class will be laid out in a pool of a specific layout by grouping fields together. Allocation takes place in a pool using the layout specification of that pool. As we do not yet model deallocation or garbage collection, pools can be thought of as growing monotonically in this paper.

The first parameter in a class declaration indicates the pool the object will be allocated into. The remaining parameters can be used in the type declarations of fields, parameter types and return types of methods and as local variables inside them. Similarly, types are annotated with pool arguments. In that respect, SHAPES is similar to parametric polymorphism in C++ [10] and Java [7] and to Ownership Types [3].

The type system for shapes follows our desire for pools to be *homogeneous*. A pool P of objects of a class C is homogeneous iff for all objects $o_1, o_2 \in P$ and for all non-primitive fields $f \in C$, $o_1.f$ and $o_2.f$ always point to objects in the same pool (or null).

To prevent the occurrence of heterogeneous pools, we require that the types of objects in the same pool share the same set of parameters. This falls out of the (upcoming) rules for well-formed types. We favour homogeneous pools for the following reasons:

- *Smaller memory footprint.* Pointers to pools consist of a pool address and an index. By using homogeneous pools, the pool address becomes redundant, as all objects of a pool will point to objects allocated inside a specific pool for a specific field f.
- *Better cache utilisation.* A direct result of the smaller footprint.
- *No need for run-time support.* Heterogeneous pools allow objects allocated inside them to point to objects allocated in different pools. These pools may have different layouts, forcing a run-time check on each field access at run-time to obtain the correct address to load. This issue is eliminated for homogeneous pools.

SHAPES and Performance In a nutshell, the SHAPES design currently targets programs that perform iteration over subsets of pooled objects, roughly in the order of access (pool allocation creating prefetcher-friendly access patterns). If objects in data structures are spread over multiple pools, programmers must manually align the objects in the pools to be cache-friendly (although we have ideas on how to automate this in future work). When objects are split into clusters, the cost of one-off accesses to objects increase because objects are spread over multiple locations that must be loaded separately. However, iterations become more efficient by loading only relevant data into cache. Note that performance means both execution time and power-efficiency stemming from improved cache-utilisation.

3 Formalising SHAPES

The syntax of SHAPES is shown in Figure 2. To simplify the formalism, we do not support programmer conveniences including ones used in Listing 2: the $*$ notation used to group remaining fields (Line 9) and defaulting to store professors in the none pool (Line 3).

Notation and Implicit requirements. We append s to names to indicate sequences; for instance, xs is a sequence of x-s, while xss is a sequence of sequences of x-s.

We assume that layout and class identifiers are unique within the same program, and field and method identifiers are unique within the same class.

Lookup Functions SHAPES rely on the following set of lookup functions. Their exact definitions are in Appendix A.

Fun.	Used to Lookup
$\mathcal{C}$	The class declaration for a given *class* identifier.
$\mathcal{P}s$	All the class *pool parameters* of a given class.
$\mathcal{P}$	The *bound* of a given parameter in a given class.
$\mathcal{M}$	The return type, the parameter type, local variables, and expression of a given *method*.
$\mathcal{F}$	The type of a given *field* in a given class.
$\mathcal{F}s$	*All the field identifiers* declared in a given class.
$\mathcal{L}$	The layout declaration of a given layout identifier.
$\mathcal{W}$	The offset(s) (*i.e.*, *where* it is located) of a given field, within an object, or within a pool.

3.1 Dynamic Semantics

SHAPES's run-time entities are defined in Figure 3.

A heap ($\mathcal{X}$) maps addresses to objects (ω) and pools (π). Objects consist of a class identifier (determining its type), a sequence of pool addresses and a record (ρ). A record is a sequence of values representing the values in the fields of an object.

Pools consist of a layout identifier, a sequence of pool addresses and a sequence of clusters (κ). The layout identifier determines how the objects inside the pool are laid out. Pools can only store instances of the class indicated by the layout identifier. Furthermore, the layout type determines the type of the pool and the type of the objects stored inside it. SHAPES also defines a global pool called none that permits objects of any type and no splitting is performed. This is similar to the default heap representation of *e.g.* the JVM.

$$
\begin{array}{rcl}
\mathit{prog} \in \mathit{Program} & ::= & \mathit{cd}^\star\ \mathit{ld}^\star \\
\mathit{cd} \in \mathit{ClassDecl} & ::= & \mathsf{class}\ C\langle \mathit{pd}^+ \rangle\ \{\ \mathit{fd}^\star\ \mathit{md}^\star\ \} \\
\mathit{fd} \in \mathit{FieldDecl} & ::= & f : t\ ; \\
\mathit{md} \in \mathit{MethodDecl} & ::= & \mathsf{def}\ m(x : t) : t\ \{\mathit{vd}\ ;\ e\} \\
\mathit{vd} \in \mathit{PoolsDecl} & ::= & \mathsf{pools}\ \mathit{lpd}^\star\ \mathsf{locals}\ \mathit{lvd}^\star \\
e \in \mathit{Expression} & ::= & \mathsf{null} \mid x \mid \mathsf{this} \mid \mathsf{new}\ t \mid x.m(x) \\
& \mid & x.f \mid x.f = x \mid x = e \\
y \in \mathit{PoolVariable} & ::= & \mathsf{none} \mid v \\
x \in \mathit{LocalVariable} & ::= & v \\
\mathit{ld} \in \mathit{LayoutDecl} & ::= & \mathsf{layout}\ L : [C] = \mathit{rd}^+ \\
\mathit{rd} \in \mathit{RecordDecl} & ::= & \mathsf{rec}\ \{f^+\}; \\
\mathit{pd} \in \mathit{PoolParameterDecl} & ::= & y : \mathit{pbd}\ \mathit{where}\ y \neq \mathsf{none} \\
t \in \mathit{ClassType} & ::= & C\langle y^+ \rangle \\
\mathit{pt} \in \mathit{PoolType} & ::= & L\langle y^+ \rangle \\
\mathit{pbd} \in \mathit{PoolBoundType} & ::= & [C\langle y^+ \rangle] \\
\mathit{lpd} \in \mathit{LocalPoolDecl} & ::= & y : \mathit{pt}\ \mathit{where}\ y \neq \mathsf{none} \\
\mathit{lvd} \in \mathit{LocalVariableDecl} & ::= & v : t
\end{array}
$$

Figure 2. Syntax of SHAPES where $v \in \mathit{VariableId}$, $C \in \mathit{ClassId}$, $f \in \mathit{FieldId}$, $m \in \mathit{MethodId}$, and $L \in \mathit{LayoutId}$.

$$
\begin{array}{rcl}
\mathcal{X} \in \mathit{Heap} & = & \mathit{Address} \rightharpoonup (\mathit{Object} \cup \mathit{Pool}) \\
\Phi \in \mathit{SFrame} & = & \mathit{VariableId} \rightharpoonup (\mathit{Value} \cup \mathit{PoolAddress}) \\
& & \cup\ \{\mathsf{none}\} \to \{\mathsf{none}\} \\
\mathit{Object} & = & \mathit{ClassId} \times \mathit{PoolAddress}^+ \times \mathit{Record} \\
\mathit{Pool} & = & \mathit{LayoutId} \times \mathit{PoolAddress}^+ \times \mathit{Cluster}^+ \\
\rho \in \mathit{Record} & = & \mathit{Value}^+ \\
\kappa \in \mathit{Cluster} & = & \mathit{Record}^+ \\
\beta \in \mathit{Value} & = & \mathit{ObjectAddress} \cup \mathit{Location} \cup \{\mathsf{null}\} \\
\mathit{Location} & = & \mathit{HeapPoolAddress} \times \mathit{Index} \\
n \in \mathit{Index} & = & \mathbb{N} \\
\alpha \in \mathit{Address} & = & \mathit{ObjectAddress} \uplus \mathit{HeapPoolAddress} \\
\omega \in \mathit{ObjectAddress} & & \\
\pi \in \mathit{PoolAddress} & = & \mathit{HeapPoolAddress} \cup \{\mathsf{none}\}
\end{array}
$$

Figure 3. Dynamic Entities of SHAPES.

The pool addresses of both objects and pools are ghost state intended to be used for proofs; we will demonstrate in future work that they are superfluous and serve no purpose at run-time.

The values (β) corresponding to the fields of objects that are allocated in pools are stored in clusters. A layout declaration designates splits; each split contains a subset of the class' fields. Each cluster, therefore, contains the values that correspond to the respective split's fields for all objects allocated in that pool. Addresses of objects inside pools (π, n) require a pool address π and an index n. The index indicates which record in each cluster contains the values that the fields of the instance correspond to.

A frame (Φ) maps variables to values. SHAPES designates three kinds of local variables:

Local object variables The this variable, and function parameters. These behave exactly like local variables in object-oriented languages.

Local pool variables These correspond to the pools that are constructed upon entering a method body and are initially empty. The reason local pool variables are defined altogether is because we allow reference cycles between objects that are allocated inside different pools.

Class pool parameters The class' pool parameters can be used inside method bodies as local variables for type declarations, object instantiations, etc.

For convenience in our definitions, we define $\Phi(\mathsf{none}) = \mathsf{none}$.

SHAPES semantics rules are of the form $\mathcal{X}, \Phi, e \rightsquigarrow \mathcal{X}', \Phi', \beta$. They take a heap, a stack frame and an expression and reduce to a new heap, a new stack frame, and a new value, in a big-step manner.

Operations on Pool-Agnostic Expressions The operational semantics for these rules are given in Figure 4. They are unsurprising, with the exception of New Object and Method Call. New Object deal with creation of unpooled objects (*i.e.*, stored in the none pool). It stores pool parameters in the objects' run-time types. As pool parameters are initialised in methods, and the implicit passing of the current object's pool parameters, Method Call is not pool-agnostic.

Pool-dependent operations The operational semantic is given in Figure 5. New Pooled Object allocates objects inside an existing pool π, by appending a new record of values (all initialised to null) for each cluster in the pool. The notation $|fs_i|$ denotes the number of fields in cluster i and $\mathsf{null}^{|fs_i|}$ a sequence of null values.

[Value] [Variable]

$$\frac{}{\mathcal{X}, \Phi, \mathsf{null} \rightsquigarrow \mathcal{X}, \Phi, \mathsf{null}} \qquad \frac{}{\mathcal{X}, \Phi, x \rightsquigarrow \mathcal{X}, \Phi, \Phi(x)}$$

[Assignment]

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

[New Object]

$$\frac{\Phi(ys) = \mathsf{none} \cdot \pi s \quad \omega \notin \mathcal{X} \quad |\mathcal{F}s(C)| = n}{\mathcal{X}, \Phi, \mathsf{new}\ C\langle ys\rangle \rightsquigarrow \mathcal{X}[\omega \mapsto (C, \mathsf{none} \cdot \pi s, \mathsf{null}^n)], \Phi, \omega}$$

[Object Read]

$$\frac{\Phi(x) = \omega \quad \mathcal{X}(\omega) = (C, \_, \rho) \quad \mathcal{W}(C, f) = i}{\mathcal{X}, \Phi, x.f \rightsquigarrow \mathcal{X}, \Phi, \rho[i]}$$

[Object Write]

$$\frac{\Phi(x) = \omega \quad \Phi(x') = \beta \quad \mathcal{X}(\omega) = (C, \pi s, \rho) \quad \mathcal{W}(C, f) = i}{\mathcal{X}, \Phi, x.f = x' \rightsquigarrow \mathcal{X}[\omega \mapsto (C, \pi s, \rho[i \mapsto \beta])], \Phi, \beta}$$

Figure 4. Operational semantics for pool-agnostic operations.

[New Pooled Object]

$$\frac{\begin{array}{c}\Phi(ys) = \pi \cdot \_ \quad \pi \neq \mathsf{none} \\ \mathcal{X}(\pi) = (L, \pi s, \kappa_0\ ..\ \kappa_n) \quad \mathcal{L}(L) = (C, fs_0\ ..\ fs_n) \\ \mathcal{X}' = \mathcal{X}[\pi \mapsto (L, \pi s, \kappa_0 \cdot \mathsf{null}^{|fs_0|}\ ..\ \kappa_n \cdot \mathsf{null}^{|fs_n|})]\end{array}}{\mathcal{X}, \Phi, \mathsf{new}\ C\langle ys\rangle \rightsquigarrow \mathcal{X}', \Phi, (\pi, |\kappa_0|)}$$

[Pooled Object Read]

$$\frac{\Phi(x) = (\pi, n) \quad \mathcal{X}(\pi) = (L, \pi s, \kappa s) \quad \mathcal{W}(L, f) = (i, j)}{\mathcal{X}, \Phi, x.f \rightsquigarrow \mathcal{X}, \Phi, (\pi, \kappa s[i, n, j])}$$

[Pooled Object Write]

$$\frac{\begin{array}{c}\Phi(x) = (\pi, n) \quad \mathcal{X}(\pi) = (L, \pi s, \kappa s) \quad \mathcal{W}(L, f) = (i, j) \\ \Phi(x') = \beta \quad \mathcal{X}' = \mathcal{X}[\pi \mapsto (L, \pi s, \kappa s[i, n, j \mapsto \beta])]\end{array}}{\mathcal{X}, \Phi, x.f = x' \rightsquigarrow \mathcal{X}', \Phi, \beta}$$

Figure 5. Operational semantics for pool dependent operations.

By Pooled Object Read, accessing a field f of an object (at location n) in a pool (π) requires looking up the j:th value of the i:th cluster of the n:th object, where i is the the cluster containing f, and j the offset into that cluster according to the layout L (by way of helper function $\mathcal{W}$). For brevity, we conflate the latter into a single 3-ary lookup: $\kappa s[i, n, j]$.

As shown by Pooled Object Write, modifying a field is isomorphic. We use a shorthand for updating, $\kappa s[i, n, j \mapsto \beta]$, similar to lookup.

Note that the syntax for constructing objects and accessing/mutating their members is the same regardless of layout and whether the object is allocated in a pool or not.

Method Call The operational semantics for method invocation are presented in Figure 6 as two separate rules for readability. The second rule, Variable/Pool Declaration is only used from inside the first, Method Call. By Method Call, a method call proceeds by constructing a new stack frame Φ for the method m, populating it with the current `this`, the method parameter x' and the `this`' pool parameters. In a big-step way, it then proceeds to evaluate the method's body in the context of the new stack frame using Variable/Pool Declaration and returning the resulting value β.

[Method Call]

$$\frac{\begin{array}{c}getThis(\mathcal{X}, \Phi(x)) = (C, \pi s, \beta) \quad \mathcal{M}(C, m) = (\_, x' : \_, vd, e) \\ \mathcal{X}, [\mathsf{this} \mapsto \beta, x' \mapsto \Phi(x''), \mathcal{P}s(C) \mapsto \pi s], vd; e \rightsquigarrow \mathcal{X}', \_, \beta'\end{array}}{\mathcal{X}, \Phi, x.m(x'') \rightsquigarrow \mathcal{X}', \Phi, \beta'}$$

[Variable/Pool Declaration]

$$\frac{\begin{array}{c}vd = \mathsf{pools}\ y_1{:}\ L_1\langle ys_1\rangle\ ..\ y_n{:}\ L_n\langle ys_n\rangle\ \mathsf{locals}\ x_1{:}\ \_\ ..\ x_m{:}\ \_ \\ \pi_1, \ .., \pi_n \notin \mathcal{X} \quad \forall i, j.\ \pi_i = \pi_j \rightarrow i = j \\ \mathcal{X}' = \mathcal{X}[\pi_1 \mapsto init(L_1\langle ys_1\rangle, \Phi'), \ .., \pi_n \mapsto init(L_n\langle ys_n\rangle, \Phi')] \\ \mathcal{X}', \Phi[x_1\ ..\ x_m \mapsto \mathsf{null}^m, y_1\ ..\ y_n \mapsto \pi_1\ ..\ \pi_n], e \rightsquigarrow \mathcal{X}'', \_, \beta'\end{array}}{\mathcal{X}, \Phi, vd; e \rightsquigarrow \mathcal{X}'', \Phi, \beta'}$$

Where:

$$\begin{array}{lll} getThis(\mathcal{X}, \omega) & \equiv (C, \pi s, \omega) & \textit{iff } \mathcal{X}(\omega) = (C, \pi s, \_) \\ getThis(\mathcal{X}, (\pi, n)) & \equiv (C, \pi s, (\pi, n)) & \textit{iff } \mathcal{X}(\pi) = (L, \pi s, \_) \ \wedge\ \mathcal{L}(L)[0] = C \\ init(L\langle ys\rangle, \Phi) & \equiv (L, \Phi(ys), \emptyset^n) & \textit{iff } n = |\mathcal{L}(L)\ [1]| \end{array}$$

Figure 6. Operational semantics for method call.

Variable/Pool declaration initialises the method's local variables. For simplicity, we require all local variables to be declared upfront and initiaise local object variables x to null. For pool variables y, 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 enable cycles among pools.

3.2 Type System

Our type system, in addition to ensuring well-typedness of run-time objects, ensures that objects are allocated with the correct layout in the correct pool. This is essential to ensure that there can be no accesses to undefined memory.

Expression Types The type rules are presented in Figure 7, and have the form $\Gamma \vdash e : t$. Γ maps variables to types:

$$\Gamma \in \mathit{TypingContext} ::= x : T, \Gamma \mid \epsilon$$
$$T \in \mathit{Type} ::= \mathit{ObjectType} \cup \mathit{PoolType} \cup \mathit{PoolBound}$$

We distinguish three kinds of types:

[Value] [Variable] [Assignment]

$$\frac{\Gamma \vdash C\langle ys\rangle}{\Gamma \vdash \mathsf{null} : C\langle ys\rangle} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash x, e : t}{\Gamma \vdash x = e : t}$$

[New Object] [Field Read]

$$\frac{\Gamma \vdash C\langle ys\rangle}{\Gamma \vdash \mathsf{new}\ C\langle ys\rangle : C\langle ys\rangle} \qquad \frac{\Gamma \vdash x : C\langle ys\rangle}{\Gamma \vdash x.f : \mathcal{F}(C, f)[\mathcal{P}s(C)/ys]}$$

[Field Write] [Method Call]

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

[Pool Variable] [None bound] [Pool bound]

$$\frac{}{\Gamma \vdash y : \Gamma(y)} \qquad \frac{\Gamma \vdash [C\langle ys\rangle]}{\Gamma \vdash \mathsf{none} : [C\langle ys\rangle]} \qquad \frac{\Gamma \vdash y : L\langle ys\rangle \quad \mathcal{L}(L)[0] = C}{\Gamma \vdash y : [C\langle ys\rangle]}$$

Figure 7. Expression type checking.

$$\frac{\mathcal{C}(C)[0] = y_1\colon [C_1\langle ys_1\rangle]\ ..\ y_n\colon [C_n\langle ys_n\rangle] \quad \forall i \in [1,n].\ \Gamma \vdash ys'[i]\colon [C_i\langle ys_i\rangle][y_1\ ..\ y_n/ys'] }{\Gamma \vdash [C\langle ys'\rangle]}\ [\textsc{Bound well-formedness}]$$

$$\frac{\Gamma \vdash [C\langle ys\rangle]}{\Gamma \vdash C\langle ys\rangle}\ [\textsc{Type well-formedness}] \qquad \frac{\Gamma \vdash [C\langle ys\rangle] \quad \mathcal{L}(L)[0] = C}{\Gamma \vdash L\langle ys\rangle}\ [\textsc{Pool well-formedness}]$$

Figure 8. Type bound well-formedness.

Object Types $C\langle ys\rangle$ where C is a class and ys are pool parameters which correspond to the class pool parameters of C.

Pool Types $L\langle ys\rangle$ where L is a layout. If L is a layout that stores objects of class C, then ys are pool parameters which correspond to the class pool parameters of C.

Pool Bounds $[C\langle ys\rangle]$ where C is a class and ys are pool parameters corresponding to the class pool parameters of C.

Pool types characterise pool values, *i.e.* pools that have been allocated dynamically (that is, when a method has been called) and are organised according to a layout. Pool bounds characterise class pool parameters, which have not yet been initialised and, therefore, do not have a layout. However, when a method is called, the class pool parameters will have pool values assigned to them.

The rationale for bounds is that a method may be invoked on an object that could be allocated on the none pool or a pool that adheres to a specific layout. Thanks to bounds, we can write code that is agnostic on the kind of pool the object is allocated.

By Value, null can have any well-formed object type. By Variable, the type of a variable x is looked-up from Γ. By Assignment, assignment to a local variable must match types. While we do not model inheritance or subtyping, these extensions are possible and straightforward. By New Object, we can create objects from any well-formed type. By Field Read, looking up a field f from a receiver x of type T requires that f is in T. Furthermore, we must translate the pool parameters names internal to T, used in its typing of f, to the arguments to which these parameters are bound where the field lookup takes place. We use the helper function $\mathcal{P}s(C)$ to extract the names of the formal pool parameters of the class C. Both Field Write and Method Call must apply similar substitutions to translate between the internal names of the formal parameters and the arguments used at the call-site. Otherwise, these rules are standard.

Roles for pool variables are straightforward. By Pool Variable, the type of a pool variable y is looked-up from Γ. By None Bound, any well-formed pool type is a bound on the none pool. By Pool Bound, the bound of a pool y is derived from its pool type.

Figure 8 describes well-formedness of types. They are similar to Featherweight Java [9]. By Bound Well-Formedness, a pool bound with object type T is well-formed if T has all its formal pool parameters bound to arguments of the correct type, modulo a substitution from the names of the formal parameters to the actual arguments of T. By Type Well-Formedness, and Pool Well-Formedness, well-formed object types and pool types can be derived from well-formed pool bounds.

3.3 Type Checking Examples

In this section we illustrate how our type rules reject programs that violate our designs and would therefore suffer performance penalties (and add complexity to our implementation).

Pool Monomorphism With the exception of the none pool, pools in SHAPES are monomorphic, *i.e.* only store objects of a single type.

```
1  class Student⟨ps: [Student⟨ps, pp⟩], pp: [Professor⟨pp, ps⟩] ⟩{
2      supervisor: Professor⟨pp, ps⟩;
3      def generate() {
4          pools stuPool: StudentSplit⟨stuPool, profPool⟩
5                profPool: ProfessorSplit⟨profPool, stuPool⟩
6          locals stu: Student⟨stuPool, profPool⟩
7                  prof: Professor⟨profPool, stuPool⟩ ;
8
9          stu = new Student⟨stuPool, profPool⟩; // OK
10         prof = new Professor⟨stuPool, profPool⟩; // Error!
11     }
12 }
13 ...
14 layout StudentSplit: [Student] = ...;
15 layout ProfessorSplit: [Professor] = ...;
```

The above program is rejected because of the attempt to construct a new Professor object inside a pool of students on Line 10.

Pool Homogeneity Pools in SHAPES are homogeneous. This means that two objects in a pool cannot have equi-named fields with different types. Consequently, all objects in a pool can share the same code for dereferencing a field.

```
1  class Student⟨ps: [Student⟨ps, pp⟩], pp: [Professor⟨pp⟩] ⟩{
2      supervisor: Professor⟨pp⟩;
3      def generate() {
4          pools stuPool: StudentSplit⟨stuPool, profPool1⟩
5                profPool1: ProfessorSplit⟨profPool1⟩
6                profPool2: ProfessorSplit⟨profPool2⟩
7          locals stu: Student⟨stuPool, profPool1⟩
8                  prof1: Professor⟨profPool1⟩
9                  prof2: Professor⟨profPool2⟩ ;
10
11         stu1 = new Student⟨stuPool, profPool1⟩;
12         stu2 = new Student⟨stuPool, profPool2⟩;
13
14         stu1.supervisor = new Professor⟨profPool1⟩; // OK
15         stu2.supervisor = new Professor⟨profPool2⟩; // Error!
16     }
17 }
18 ...
19 layout StudentSplit: [Student] = ...;
20 layout ProfessorSplit: [Professor] = ...;
```

The above program is rejected as it attempts to assign two students in the same pool to professors in different pools (Line 12 & 15). If this was legal, emitting code for student.supervisor.name, where student could refer to either stu1 or stu2, would be forced to branch on the layout of the supervisor at run-time.

4 Well-formedness and Type Safety

The main meta-theoretic result of this paper is the type and memory safety of field accesses, invariant of layout changes. Going back to Figure 1, a programmer cannot directly reference $(f, f')_i$ in the right-most representation. To access "$i.f$", we must find the ith f in the array. To extract the object $(f, f')_i$, we must take care not to accidentally return (f_i, f'_j), which would be an object created out of thin air. This is a consequence of the broken object abstraction.

Therefore, it is necessary for a type safety definition of SHAPES to show that given two aliasing references to a pooled object, the values yielded from accessing the respective fields must be always equal. This must also take into consideration that the types of

the variables/fields that store these references may have different conceptual types (for instance, during a method call, the pool parameters of a variable may be renamed).

In the earlier sections, we have treated the program as implicitly given. Here too, we assume its existence and we assume its well-formedness. A program is well-formed if all of its class and layout declarations are well-formed. The definition of the former is quite standard and the definition of the latter checks that the layout declaration for a given class considers all the fields of that class and that no field appears repeated in different clusters. Formal definitions can be found in Appendix B.

Because of the above aliasing requirement, we need to define well-formedness in such a way that the requirement holds. We also want to show that the well-formedness definition allows us to perform a few optimising transformations, while showing that the compiled code still preserves the layout requirements of all pools and returns appropriate values (we leave the definition of such a compilation for future work). Two of these optimisations are the removal of pool parameters from objects and pools and the simplification of pool addresses from a pool and an index (π, n) to simply an index (n).

Therefore, we define a pool-aware and layout-aware definition of well-formed configurations that is stronger than a typical definition of well-formed configurations in object-oriented languages. Well-formed configurations in SHAPES require, among other things, that all objects in a pool have the same class (the one required in the pool's layout type), and that all the fields of an object point to objects which have classes and are in pools as described in the object's type. We formally define well-formed configuration in § C.

We state type safety for SHAPES in Theorem 4.1, in the sense that if a well-formed configuration takes a reduction step, then the resulting configuration is well-formed too.

Theorem 4.1 (Type Safety)**.** *For a well-formed program with heap $\mathcal{X}$, stack frame Φ, corresponding typing context Γ, and expression e,*

$$\textbf{\textit{If}}\ \Gamma \vDash \mathcal{X}, \Phi\ \wedge\ \Gamma \vdash e : C\langle xs\rangle\ \wedge\ \mathcal{X}, \Phi, e \rightsquigarrow \mathcal{X}', \Phi', \beta$$
$$\textbf{\textit{then}}\ \Gamma \vDash \mathcal{X}', \Phi'\ \wedge\ \mathcal{X}' \vDash \beta : C\langle \Phi'(xs)\rangle$$

Proof sketch. By structural induction over the derivation $\mathcal{X}, \Phi, e \rightsquigarrow \mathcal{X}', \Phi', \beta$. □

5 Related Work

For an extended coverage of related work, see Franco et al. [6]

Memory layouts The idea of data placement to reduce cache misses was first introduced by Calder et al. [2], applying profiling techniques to find temporal relationships among objects.

This work was then followed up by Lattner et al. [12, 13] where rather than relying on profiling, static analysis of C and C++ programs finds what layout to use. Huang et al. [8] explore pool allocation in the context of Java.

Ureche et al. [18] present a Scala extension that allows automatic changes to the data layout. The developer defines transformations and the compiler applies the transformation during code generation.

Heap partitioning Deterministic Parallel Java also provides means to split data in the heap: Java code is annotated with regions information used to calculate the effects of reading and writing to data [1]. Loci [19] split the heap into per-thread subheaps plus a shared space. Note that these languages only divide the heap *conceptually*, and do not aim to affect representation in memory.

Jaber et al. [11] present a heap partitioning scheme that works by inferring ownership properties between objects.

In the context of NUMA systems, Franco and Drossopoulou use annotations to describe in which NUMA nodes the objects should be placed [5],with the aim to improve program performance by reducing memory accesses *to remote nodes*, ignoring any possible in-cache data accesses.

Formalisation The SHAPES type system is influenced by Ownership types [4] but uses pools rather than ownership contexts, and without enforcing a hierarchical decomposition of the object graph (that is, we allow and handle cycles between pools).

As mentioned above, the concept of bounds and well-formed types is drawn from Featherweight Java [9], with the exception that our formalism does not have any concepts of polymorphism.

Formalisms for automatic data transformations with regards to functional languages also exist. Leroy [14] presents a formalised transformation of ML programs that allows them to make use of unboxed representations. Thiemann [17] extends on this work and Shao [16] generalises it.

Petersen et al [15] describe a model that uses ordered type theory to define how constructs in high-level languages are laid out in memory. This allows the runtime to achieve optimisations such as the coalescing of multiple calls to the allocator.

6 Final Remarks

We have presented a formal model (operational semantics and type system) of SHAPES, an object-oriented programming language that provides first class support for object splitting and pooling. We have also provided the definition for a well-formed runtime configuration of SHAPES and justified our design decisions.

We include an extended discussion of future work in [6]. Our next step is showing that the well-formedness of a configuration is preserved during execution. We will also provide a translation to a low-level language that will be designed in such a way so as to achieve reasonable performance and preserve the operational semantics of SHAPES code.

7 Acknowledgements

We would like to thank Christabel Neo and the FTfJP reviewers for their feedback, and in particular, the very useful suggestions on how to make our explanations and notation better and easier to understand.

References

[1] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. L. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, pages 97–116, 2009. DOI: 10.1145/1640089.1640097

[2] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data Placement. In *ASPLOS VIII*, pages 139–149. ACM, 1998. DOI: 10.1145/291069.291036

[3] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership Types: A Survey. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, LNCS 7850, pages 15–58, 2013. DOI: 10.1007/978-3-642-36946-9_3

[4] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA '98*, pages 48–64. ACM, 1998. DOI: 10.1145/286942.286947

[5] J. Franco and S. Drossopoulou. Behavioural Types for Non-Uniform Memory Accesses. *PLACES 2015*, page 39, 2015. DOI: 10.4204/EPTCS.203.9

[6] J. Franco, M. Hagelin, T. Wrigstad, S. Drossopoulou, and S. Eisenbach. You Can Have it All: Abstraction and Good Cache Performance. In *Onward!* 2017, pages 148–167, New York, NY, USA, 2017. ACM. DOI: 10.1145/3133850.3133861

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

[8] X. Huang, S. M. Blackburn, K. S. Mckinley, J. Eliot, B. Moss, Z. Wang, and P. Cheng. The Garbage Collection Advantage: Improving Program Locality. In *OOPSLA*, 2004. DOI: 10.1145/1035292.1028983

[9] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS.*, 23(3):396–450, May 2001. DOI: 10.1145/503502.503505

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

[11] N. Jaber and M. Kulkarni. Data Structure-Aware Heap Partitioning. In *26th International Conference on Compiler Construction*, pages 109–119, 2017. ACM. DOI: 10.1145/3033019.3033030

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

[13] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI '05*, pages 129–142. 2005. DOI: 10.1145/1065010.1065027

[14] X. Leroy. Unboxed Objects and Polymorphic Typing. In *POPL '92*, pages 177–188, New York, NY, USA, 1992. ACM. DOI: 10.1145/143165.143205

[15] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A Type Theory for Memory Allocation and Data Layout. In *POPL '03*, pages 172–184, 2003. ACM. DOI: 10.1145/604131.604147

[16] Z. Shao. Flexible Representation Analysis. In *ICFP '97*, pages 85–98, New York, NY, USA, 1997. ACM. DOI: 10.1145/258948.258958

[17] P. J. Thiemann. Unboxed Values and Polymorphic Typing Revisited. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 24–35, 1995. ACM. DOI: 10.1145/224164.224175

[18] V. Ureche, A. Biboudis, Y. Smaragdakis, and M. Odersky. Automating Ad Hoc Data Representation Transformations. In *OOPSLA* 2015, pages 801–820, 2015. DOI: 10.1145/2814270.2814271

[19] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple Thread-Locality for Java. In *ECOOP 2009*, LNCS, pages 445–469. Springer, 2009. DOI: 10.1007/978-3-642-03013-0_21

[20] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995. DOI: 10.1145/216585.216588

A Lookup functions

$$
\begin{aligned}
\mathcal{C}(C) &\triangleq (pds\ fds\ mds)\ \textit{iff}\ (\text{class}\ C\langle pds\rangle\{fds\ mds\}) \in prog[0]\\
\mathcal{P}s(C) &\triangleq y_1\ ..\ y_n\ \textit{iff}\ \mathcal{C}(C)[0] = (y_1 : \_\ , ...,\ y_n : \_)\\
\mathcal{P}(C, y) &\triangleq pbd\ \textit{iff}\ (y : pbd) \in \mathcal{C}(C)[0]\\
\mathcal{M}(C, m) &\triangleq (t, x : t', vd, e)\ \textit{iff}\ (\text{def}\ m(x : t') : t\ \{vd; e\}) \in \mathcal{C}(C)[2]\\
\mathcal{F}(C, f) &\triangleq t\ \textit{iff}\ (f : t) \in \mathcal{C}(C)[1]\\
\mathcal{F}s(C) &\triangleq f_1\ ..\ f_n\ \textit{iff}\ \mathcal{C}(C)[1] = (f_1 : \_\ ..\ f_n : \_)\\
\mathcal{L}(L) &\triangleq (C, fs_1\ ..\ fs_n)\ \textit{iff}\\
&\qquad (\text{layout}\ L : [C] = \text{rec}\{fs_1\};\ ..\ \text{rec}\{fs_n\}) \in prog[1]\\
\mathcal{W}(L, f) &\triangleq (i, j)\ \textit{iff}\ \mathcal{L}(L) = (C, fss)\ \wedge\ fss[i, j] = f\\
\mathcal{W}(C, f) &\triangleq i\ \textit{iff}\ \mathcal{F}s(C)[i] = f
\end{aligned}
$$

B Well-formed Programs

Definition B.1 (Well-formed context)**.** We use the notation $\vdash \Gamma$ to declare that a context is well-formed. We define:

$$\vdash \Gamma\ \textit{iff}\ \forall(\_ : T) \in \Gamma.\ \Gamma \vdash T\ \wedge\ \forall y.\ y \vdash \Gamma : [C\langle ys\rangle] \rightarrow ys[0] = y$$

Given definitions B.3 and B.4, we define well-formed programs as follows:

Definition B.2 (Well-formed program)**.** A program is well-formed if all its layout and all its class declarations are well-formed.

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

Definition B.3 (Well-formed class declaration)**.** A class C is well-formed if:

- Their first pool parameter has to be annotated with a bound that is of the same class and its parameters are the same as in the class declaration (and in the same order). That is, if the class pool parameters of the class C are $\mathcal{P}s(C) = y_1\ ..\ y_n$, then $\mathcal{P}(C, y_1) = [C\langle y_1, \ldots, y_n\rangle]$.
- The parameter list of all pool types must only contain parameters from the class' pool parameter list (*i.e.* $\mathcal{P}s(C)$). This means that the none keyword is disallowed as a pool parameter name.
- The fields must have class types that are well-formed against the typing context Γ where the class' formal pool parameters have their corresponding bounds as types. Moreover, Γ is well-formed.
- All the methods have a parameter and return type that is well-formed against the context Γ. Moreover, for each method, the corresponding method body is typeable against a context Γ' which is an augmentation of Γ and contains the types of this variable, local pool, and object variables of the method. Moreover Γ' is well-formed.

$$
\begin{aligned}
prog \vdash\ &\text{class}\ C\langle y_1 : [C_1\langle ys_1\rangle]\ ..\ y_n : [C_n\langle ys_n\rangle]\rangle\ \{\ fds\ \ mds\ \}\ \textit{iff}\\
&\quad \vdash \Gamma\ \wedge\ C_1 = C\ \wedge\ ys_1 = y_1\ ..\ y_n\\
&\wedge\ \forall f : T \in fds.\ \Gamma \vdash T\\
&\wedge\ \forall \text{def}\ m(x : t) : t'\ \{vd\ ;\ e\} \in mds.[\\
&\qquad \Gamma \vdash t\ \wedge\ \Gamma \vdash t'\\
&\quad \wedge\ \vdash \Gamma'\ \wedge\ \Gamma' \vdash e : t'\ ]\\
&\qquad \textit{where}\ \Gamma' = \Gamma, \text{this} : C\langle y_1\ ..\ y_n\rangle, x : t,\\
&\qquad\quad y'_1 : L_1\langle ys'_1\rangle,\ ..\ , y'_k : L_k\langle ys'_k\rangle,\\
&\qquad\quad x_1 : C'_1\langle ys''_1\rangle,\ ..\ , x_m : C'_m\langle ys''_m\rangle\\
&\qquad\quad vd = \text{pools}\ y'_1 : L_1\langle ys'_1\rangle\ ..\ , y'_k : L_k\langle ys'_k\rangle\\
&\qquad\qquad \text{locals}\ x_1 : C'_1\langle ys''_1\rangle\ ..\ x_m : C'_m\langle ys''_m\rangle\\
&\textit{where}\ \Gamma = y_1 : [C_1\langle ys_1\rangle]\ ..\ y_n : [C_n\langle ys_n\rangle]
\end{aligned}
$$

We now define well-formedness of layout declarations:

Definition B.4 (Well-formed layout declaration)**.** A layout declaration for instances of a class C is well-formed iff the disjoint union of its clusters' fields is the set of the fields declared in C.

$$
\begin{aligned}
prog \vdash\ &\text{layout}\ L : [C] = rd_1 ... rd_n\ \textit{iff}\\
&\{\mathcal{F}s(C)\} = \biguplus_{i \in 1\ ..\ n} \{fs \mid rd_i = \text{rec}\ \{fs\}\}
\end{aligned}
$$

This definition excludes repeated or missing fields. For example, if a class *Video* has 3 fields with names *id*, *likes*, *views*, then both of these layout declarations are ill-formed:

```
// repeated field
layout Ill_formed_Layout1: [Video] = rec {id, likes} + rec {likes, views}
// missing field
layout Ill_formed_Layout2: [Video] = rec {id} + rec {views}
```

C Well-formed Configurations

As usual, an object or pool adheres to a type $C\langle ys\rangle$ or $L\langle ys\rangle$, respectively, if the class of the object or pool is C, the pool parameters are $\Phi(ys)$ and all of its fields or clusters adhere to their type. To avoid the circularities in the definition of agreement, we break it down into weak agreement, which only ensures that the run-time type of

the object is the one specified by its class and pool parameters, and strong agreement, which also considers the contents of the fields or clusters. In a nutshell, strong agreement checks the vertices of the object graph, while weak agreement checks the edges.

Although the type system uses local variables in its types, we cannot use them in the well-formedness definition, because local pool names may change across function calls. Thus, we use run-time types instead, where the pool parameters are substituted with pool addresses.

Definition C.1 (Run-Time types)**.** A run-time type τ is a defined as follows:

$$
\begin{aligned}
\tau \in \mathit{RuntimeType} ::= {} & \mathit{RuntimeClassType} \cup \mathit{RuntimeLayoutType} \\
& \cup \mathit{RuntimeBound} \\
\mathit{RuntimeClassType} ::= {} & C\langle \pi_1 \,..\, \pi_n \rangle \\
\mathit{RuntimePoolType} ::= {} & L\langle \pi_1 \,..\, \pi_n \rangle \\
\mathit{RuntimeBound} ::= {} & [C\langle \pi_1 \,..\, \pi_n \rangle]
\end{aligned}
$$

We now define the well-formedness of a run-time configuration:

Definition C.2 (Well-formed high-level configurations)**.** The definition of well-formedness of a heap is as follows:

$\vDash \mathcal{X}$ *iff* $\forall \alpha \in \mathrm{dom}(\mathcal{X}).\ \exists \tau.\ \mathcal{X} \vDash \alpha \triangleleft \tau$

An address can be either an address to heap-allocated object, to a pool, or to a pool-allocated object, therefore we split the definition of well-formed address as follows:

- $\mathcal{X} \vDash \omega \triangleleft C\langle \pi s \rangle$ *iff*
  $\mathcal{X}(\omega) = (C, \pi s, \rho) \ \wedge\ \pi s[0] = \mathsf{none}$
  $\wedge\ \forall i \in 1 \,..\, n.\ \mathcal{X} \vDash \pi s[i-1] : \mathcal{P}(C, y_i)[y_1 \,..\, y_n / \pi s]$
  $\wedge\ \forall f.\ \mathcal{X} \vDash \rho[\mathcal{W}(C, f)] : \mathcal{F}(C, f)[\mathcal{P}s(C)/\pi s]$
  where $y_1 \,..\, y_n = \mathcal{P}s(C)$
- $\mathcal{X} \vDash \pi \triangleleft L\langle \pi s \rangle$ *iff*
  $\mathcal{X}(\pi) = (L, \pi s, \kappa s) \ \wedge\ \pi s[0] = \pi$
  $\wedge\ \forall i \in 1 \,..\, n.\ \mathcal{X} \vDash \pi s[i-1] : \mathcal{P}(C, y_i)[y_1 \,..\, y_n / \pi s]$
  $\wedge\ |\kappa_1| = \ldots = |\kappa_n|$
  $\wedge\ \forall i \in 0 \,..\, |\kappa_1| - 1.\ \mathcal{X} \vDash (\pi, i) \triangleleft C\langle \pi s \rangle$
  where $y_1 \,..\, y_n = \mathcal{P}s(C), \mathcal{L}(L)[0] = C$
- $\mathcal{X} \vDash (\pi, n) \triangleleft C\langle \pi s \rangle$ *iff*
  $\mathcal{X}(\pi) = (L, \pi s, \kappa s) \ \wedge\ \pi s[0] = \pi$
  $\wedge\ \mathcal{L}(L)[0] = C$
  $\wedge\ \forall f.\ \mathcal{X} \vDash \kappa s[i, n, j] : \mathcal{F}(C, f)[\mathcal{P}s(C)/\pi s]$
  where $(i, j) = \mathcal{W}(L, f)$

The definition of weak agreement for objects is as follows:

- $\mathcal{X} \vDash \omega : C\langle \pi s \rangle$ *iff* $\mathcal{X}(\omega) = (C, \pi s, \_)$
  $\wedge\ \pi s[0] \neq \mathsf{none}$
- $\mathcal{X} \vDash (\pi, n) : C\langle \pi s \rangle$ *iff* $\mathcal{X}(\pi) = (L, \pi s, \_)$
  $\wedge\ \pi s[0] = \pi \ \wedge\ \mathcal{L}(L)[0] = C$
- $\mathcal{X} \vDash \mathsf{null} : C\langle \_ \rangle$

The definition of weak agreement for pools and bounds is as follows:

- $\mathcal{X} \vDash \pi : L\langle \pi s \rangle$ *iff* $\mathcal{X}(\pi s[0]) = (L, \pi s, \_)$
- $\mathcal{X} \vDash \mathsf{none} : [C\langle \_ \rangle]$
- $\mathcal{X} \vDash \pi : [C\langle \pi s \rangle]$ *iff* $\mathcal{X}(\pi) = (L, \pi s, \_)$
  $\wedge\ \mathcal{L}(L)[0] = C \ \wedge\ \pi = \pi s[0]$

Finally, the definition of well-formedness of a stack frame and a heap against a context is as follows:

$$
\begin{aligned}
\Gamma \vDash \mathcal{X}, \Phi \quad \mathit{iff} \quad & \vDash \mathcal{X} \\
\wedge \quad & \mathrm{dom}(\Phi) = \mathrm{dom}(\Gamma) \\
\wedge \quad & \forall x \in \mathrm{dom}(\Phi).\ [\\
& \quad [\Gamma(x) = C\langle ys \rangle \rightarrow \mathcal{X} \vDash \Phi(x) : C\langle \Phi(ys) \rangle] \wedge \\
& \quad [\Gamma(x) = L\langle ys \rangle \rightarrow \mathcal{X} \vDash \Phi(x) : L\langle \Phi(ys) \rangle] \wedge \\
& \quad [\Gamma(x) = [C\langle ys \rangle] \rightarrow \mathcal{X} \vDash \Phi(x) : [C\langle \Phi(ys) \rangle]] \\
&]
\end{aligned}
$$