The OHMM framework: Optimised Heaps with Memory Management

Juliana Franco[1], Martin Hagelin[2], Tobias Wrigstad[2], and Sophia Drossopoulou[1]

[1] Imperial College London
[2] Uppsala University

Abstract. On modern architectures, as a consequence of the gap between CPU speeds and memory bandwidth, an application's performance is often controlled by the speed at which data can be delivered to the CPU. Therefore, to a large extent, optimizing a program's performance requires organising memory so as to improve the speed of data delivery through optimal use of caches and prefetching.
Two proven techniques are pooling and splitting, which allow grouping and splitting of data in memory depending on whether they are accessed together or separately. However, these are low-level optimisations, and therefore outside of the control of the programmer in managed programming languages where memory is abstract.
In this paper, we propose OHMM, a framework for memory optimisations for high-level languages with managed run-times. OHMM is based on a variation of ownership types to express high-level constraints on objects' placement in memory. It compiles to a low-level language that features pooling and splitting. Objects' placement in OHMM does not affect their functional behaviour. This leads to a clean separation of concerns, and supports change of placement without need to rewrite the program logic. We introduce OHMM through a running example, give a formal description, and argue its correctness.

1 Introduction

Most modern programming languages are designed with the mindset that memory accesses are "for free". When the speed of a memory access rivalled that of the CPU, this abstraction was valid. However, today, CPU speeds greatly exceed memory bandwidths on most platforms, to the point where computation is almost for free, and the real cost of execution, both in terms of speed and power consumption, is in accessing memory (*c.f.* the memory wall [25]).

Cache memories, or hierarchies of cache memories, have been part of modern architectures to hide this latency for long time, exploiting the temporal and spatial locality inherent in most programs. In this type of architectures, accessing data from the different memory levels has different costs—accessing caches is much faster than accessing data in main memory. Therefore, a core interacts as much as possible with caches, reading and writing to main memory only upon *cache misses*, that is, when the required data in not in cache.

In order to reduce the number of cache misses, and improve the program's performance, the programmer needs to understand "what goes into cache" when data is loaded from memory, and be able to write programs to leverage cache effects – which is at odds with mainstream programming abstractions[3]. Ultimately, memory is an array of bytes, and unless the high-level data is carefully mapped to this array of bytes, there is no control over what will cause cache misses.

In languages with manual memory management, two techniques for efficient memory layout have been proposed. *Pooling*, whereby objects are created in separate memory pools depending on their type or time of allocation, thus objects that are frequently used together should be placed together for better cache utilisation [4,17]. *Object splitting* splits composite objects up into different parts that ideally are not used together often [12]. When performance is crucial, programmers explicitly program pooling and splitting by manually transforming an array of *structs* into a *struct* of arrays in order to obtain the same behaviour of pooling and splitting. However, this manual approach adds complexity to the code, and makes code error-prone and not suitable for object-oriented programming, as the reasoning about an object, or struct, as a single unit is not longer valid.

Pooling and splitting are low-level techniques, which do not support separation of concerns. The functional, "business logic" part of the code is polluted by low-level considerations: every object allocation, and every field access needs to be written so as to adhere to the data layout strategy. Also, any changes to the layout strategy affect all object allocations and accesses. Moreover, in languages with managed runtimes (*e.g.*, Java), the implementation of data layout is virtually impossible.

To alleviate these problems, we propose a lightweight extension to high-level languages, which is translated to code that supports the particular data layout. In more detail, types in high-level language programs should be annotated with layout information, and the appropriate allocation, and object access code should be generated by the compiler. Thus programmers can separate the functional concerns from data-layout concerns, and can treat the data layout declaratively, and leave the generation of allocation and access code to the compiler. We call this approach OHMM (Architecture in Fig. 1.)

In this paper we give a formal model for OHMM. We define OHMM^{hl} – a high-level language with layout information, and give its semantics, which is data-layout agnostic (Section 3). We then define OHMM^{ll}, the low-level target language which is data-layout aware, and define its semantics (Section 4). We define a translation from OHMM^{hl} to OHMM^{ll} (Section 5), and prove that it is meaning preserving (Section 6). We discuss the underlying framework implementation which supports the data-layout concerns (Section 7) and outline several directions for future work.

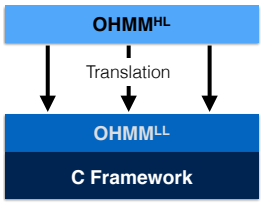


Fig. 1: OHMM overview.

[3] In Java, for instance, the size of objects is generally not even known by programmers, and there is no direct control over how data is organised in memory.

2 The Importance of Data Layout

In this section we give an overview on how processors interact with memories, and explain how layout of data in memory impacts performance due to caching and prefetching. Fig. 2 shows a simple architecture composed of one CPU core, a hierarchy of caches, and the main memory. When a value is loaded from main memory (due to a *cache miss*), the value, and some surrounding values (a *cache line*), are *fetched*/copied to the cache[4] making subsequent accesses to any of these values significantly faster (*cache hits*). If the hardware detects a pattern in the addresses loaded (*e.g.*, $10, 26, 42, 58\ldots$), it will speculatively load – *prefetch* – data into cache in-time for subsequent access. Such patterns arise easily when iterating over arrays, but not necessarily with linked lists, unless links are adjacent.

Table 1 shows access times to different memory levels for a concrete machine.

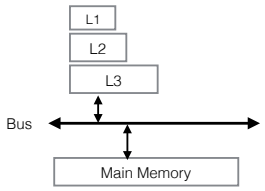


Fig. 2: Memory configuration.

Sub system	Latency	Slowdown
L1 Cache	4 cycles	x 1
L2 Cache	12 cycles	x 3
L3 Cache	21 cycles	x 5
DRAM	250 cycles	x 62

Table 1: Latency of the different memory levels, in an Intel i7-4600U CPU. Numbers taken from [13].

For the machine in Table 1, accessing the top most cache is almost for "free", while accessing the main memory is 62 times more expensive. Clearly, for memory-bound workloads, whether memory accesses mostly go to cache or to DRAM significantly impacts performance. Thus, writing programs with good locality often improves programs performance. Caches leverage programs' *temporal locality*, the fact that an accessed value is likely to be accessed again soon. The copying of entire cache lines into cache, leverage programs' *spatial locality*, the fact that values close to a recently accessed value have a higher probability of being accessed. The rationale for OHMM is increased cache-utilisation by facilitating the organising of data for increased spatial locality and prefetcher-friendly patterns.

We now demonstrate why programs may suffer from poor data locality. Consider the VideoList in Fig. 3, a typical list of nodes linked by a next field. Each object of type Node points to a further object of type Video, which contains three fields of type int: an identifier (id), the number of times the video has been played (views) and the number of likes the video has gotten.

The method popularVideos measures popularity by iterating over the whole list and checking for each video, if the number of views is greater than a given pivot. If it is, then it prints its id, views and likes. Note that the colour of the coloured squares next to fields corresponds to the colours of the values of the fields, in all the diagrams of the paper.

[4] How values are fetched, and how many values are fetched to cache, is machine-specific.

```
class Video                    class VideoList
  id: int                        head: Node
  views: int                     def popularVideos(pivot: int): void
  likes: int                       let cur = this.head in
                                   while (cur != null) {
class Node                           if cur.video.views > pivot then
  video: Video                         print(cur.video.id, cur.video.views, cur.video.likes);
  next: Node                       cur = cur.next }
```

Fig. 3: Running example: Video List. Data colours are used in subsequent figures.

The way that programmers "visualise" this list in memory may differ from how the objects are actually allocated. Fig. 4 shows how data structures are typically taught and depicted in text books, which does not at all reflect the reality in memory. For example, it is possible that nodes and videos are scattered over memory with little or no ordering, depending on how memory is allocated, or on how a garbage collector might move or compact objects, as shown in Fig. 5.

A data layout as "chaotic" as the one in Fig. 5 is problematic for two reasons mainly: no patterns will arise in the addresses loaded (meaning no prefetching); surrounding objects in the same cache line of a may not be part of the accessed data (*e.g.*, a video that is not popular, or worse – an object that is not part of the data structure) and therefore not used (no spatial locality). In the worst case, each video or node access will result in an separate, expensive cache miss.

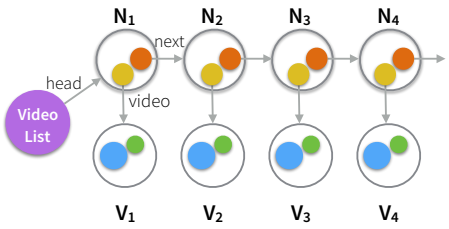


Fig. 4: VideoList representation in the programmer's mind.

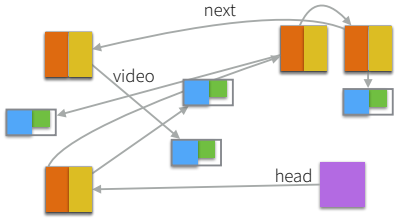


Fig. 5: Actual VideoList representation in memory.

3 OHMM^{hl}: Abstract Placement Control

One way to improve data locality in our example is to allocate all the Video objects of each list in consecutive memory, so that, when a video is read from memory, a few more videos (depending on the cache line size) will be loaded as well. This optimisation can be refined by splitting objects so that only the "useful" part of the object is loaded into cache, thereby increasing cache utilisation.

In this section we introduce OHMM^{hl}, an object oriented programming language with annotations that describe where objects should be allocated and how

they should be split. We use the video list as a running example. The syntax and notation of OHMM^{hl} is as below:

$$
\begin{array}{rl}
P \in \mathit{Program} & ::= cd^\star \\
cd \in \mathit{ClassDecl} & ::= \mathsf{class}\ C\langle\ o^\star\ \rangle\ cl^\star\ md^\star \\
cl \in \mathit{ClusterDecl} & ::= \mathsf{cluster}\ \{fd^\star\} \\
fd \in \mathit{FieldDecl} & ::= f : T \\
md \in \mathit{MethodDecl} & ::= \mathsf{def}\ m(x : T) : T\ \{e\} \\
T \in \mathit{Type} & ::= C\langle\ l^\star\ \rangle
\end{array}
\qquad
\begin{array}{rl}
l \in \mathit{Location} & ::= o \mid p \mid \mathsf{none} \\
e \in \mathit{Expression} & ::= \mathsf{null} \mid x \\
& \mid \mathsf{new}\ C\langle\ l^\star\ \rangle \mid e.m(e) \\
& \mid e.f \mid e.f = e \\
& \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{pool}\ p\ \mathsf{of}\ C\ \mathsf{in}\ e \\
x \in \mathit{Variable} & \cup \{\mathsf{this}\}
\end{array}
$$

$o \in \mathit{OwnerParam} \qquad p \in \mathit{PoolId} \qquad C \in \mathit{ClassId} \qquad f \in \mathit{FieldId} \qquad m \in \mathit{MethodId}$

It is very similar to a standard object-oriented language, except for the highlighted parts, which denote unusual constructs. Our annotations are inspired from ownership types [6], where classes declarations take ownership/pool parameters. These class parameters denote the pools to which objects belong. In addition, note that OHMM^{hl} groups field declarations inside *clusters*.

OHMM^{hl} expressions are quite standard; we only introduce a new expression construct for pool creation. OHMM^{hl} types are also annotated with locations which indicate whether an object is allocated in a pool (o or p) or floating in memory (none). We will now explain with more detail the role of our annotations.

Ownership Annotations and Pool Allocation. In this section we informally explain the use of our location annotations to place objects. Our class declarations can be extended as follows:

```
class VideoList⟨o1, o2, o3⟩
  head: Node⟨o2, o3⟩
  /** etc **/
```

```
class Node⟨o1, o2⟩
  video: Video⟨o2⟩
  next: Node⟨o1, o2⟩
  /** etc **/
```

```
class Video⟨o⟩
  id: int
  views: int
  likes: int
  /** etc **/
```

This means that an instance of VideoList will be located in o1, and its field points to objects located in o2, similarly to other ownership type systems [7]. The class declaration Node takes two location parameters and has two fields. The object referred by next is an instance of Node in o1, i.e., at the same location as as the this node. The object referred by video is an instance of Video in some other location o2. The Video type is parametrised over a single pool parameter denoting its containing location; all its three fields are of primitive type, and have value semantics—they do not take additional parameters. With this OHMM code, all the nodes of this list are allocated in the same contiguous space, the same pool, as well as, all the videos are allocated all together in some other pool.

OHMM^{hl} types can be used to allow different instances of the same class to be allocated in different places: objects can be *floating somewhere* in memory, or allocated in pools, depending on their type. We consider, for example, three different allocating schemes, using the current VideoList example:

Scheme 1. All objects are allocated anywhere in memory (cf. Fig. 5).

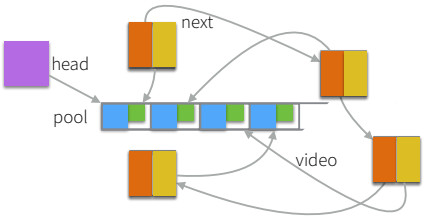


Fig. 6: Scheme 2: VideoList pointing to nodes allocated somewhere in memory, which videos allocated in one pool.

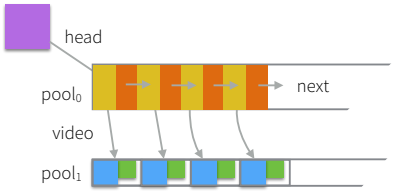


Fig. 7: Scheme 3: VideoList pointing to objects allocated in two different pools.

Scheme 2. Videos are allocated in one pool while all the other objects are floating in memory (cf. Fig. 6).

Scheme 3. Nodes and videos are allocated in pools; and the VideoList object is somewhere in memory, (cf. Fig. 7).

In order to achieve these different layouts we allow the programmer to use in her types the keyword none to identify objects that are not allocated in pools, or to create pools which can be referred in the types as well. The code to create a VideoList with these three schemes is below:

```
/** Scheme 1 **/
new List⟨none, none, none⟩
/** Scheme 2 **/
Pool pool of Video in
  new List⟨none, none, pool⟩
```

```
/** Scheme 3 **/
Pool pool0 of Node
     pool1 of Video
in
  new List⟨none, pool0, pool1⟩
```

In the third layout, depicted in Fig. 6, the head field of the list points to the first position of $pool_0$[5], and all the next fields point to nodes in their next positions. As a consequence, an access to a node will either result in a cache hit, or in loading more nodes into cache for subsequent cache hits. The videos are also allocated contiguously. Note that the order in which the objects are allocated inside the pool is very important. In the diagram of Fig. 6, the next field always points to the next node in the pool. If instead, it pointed to another Node in some random location, then we would potentially not get any advantage in using pooled allocation, as fetching of a node would not bring other useful nodes into cache. Currently, the order of objects within pools is determined by the order in which they were allocated. We intend to explore other kinds of ordering in future work and also take advantage of the garbage collector, as we describe later.

Cluster Annotations and Object Splitting. We now explain how cluster annotations split objects in a pool into different clusters. In our example, each loop iteration reads the video and the next fields of a Node (a common pattern when iterating over linked-lists). Therefore, keeping these two fields together allows

[5] we can think of a pool as an array where each element contains an object (not a pointer as it is common in other languages).

fetching them together, potentially avoiding one cache miss per iteration. Indeed, in the code below, video and next fields are in the same cluster.

```
class Node⟨o1, o2⟩
  cluster {
    video: Video⟨o2⟩
    next: Node⟨o1, o2⟩
  }
```

```
class Video⟨o⟩
  cluster {
    id: int
    likes: int }
  cluster {
    views: int }
```

```
1  let cur = this.head in
2  while (cur != null) {
3    let v = cur.video in
4    if v.views > pivot then
5      print(v.id, v.views, v.likes);
6    cur = cur.next}
```

On the other hand, while in the popularVideos method each iteration must read the views of a Video, only under certain conditions will the fields id and likes be read (lines 4 and 5 below). Allocating these three fields together will *probably* bring useless data, id and likes fields, into cache. The less likely it is that these two fields are read (because of a high pivot) the more we hurt program performance by reading the entire video. Therefore, we annotate the Video in a way that the views field is allocated in a cluster, and the id and likes fields are allocated in another cluster. This class declaration splits pool_1 of Fig. 6 in cluster_1 and cluster_2. The result is in Fig. 8. From now on we only refer to this Video class declaration when using the example.

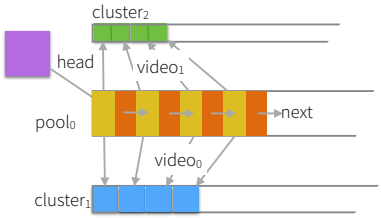


Fig. 8: Splitting within a pool.

OHMM^{hl} Dynamic Semantics. The semantics of OHMM^{hl} is meant to represent the programmer's high-level view of execution, not its actual execution. OHMM's programs are meant to be translated to executable code.

Thus even though OHMM^{ll} allows objects to be split and allocated in pools, this information is ghost at the high-level. This semantics does not differ much from other object oriented calculi [6]. The heap is a map from addresses to objects, $\chi \in \mathit{Heap} = (\mathit{Address} \to \mathit{Object}) \times \mathit{PoolLoc}^\star$. Objects contain a sequence of locations (where the first location corresponds to the object's location) in addition to the usual class identifier and map from fields to values, $\omega \in \mathit{Object} = \mathit{ClassId} \times \mathit{HeapLoc}^\star \times (\mathit{FieldId} \to \mathit{Value})$.

Objects are not actually split in memory, and their locations are ghost information; OHMM^{hl} semantics consider object graphs as in Fig. 5. This means that reading and writing to fields totally ignore object splitting and pool allocation. The only expression that differs from other calculi is pool creation; it just adds a new pool location to the heap and stack frame, so that it can be used in the continuation. Because of its similarity with other systems, we leave the semantics of OHMM^{hl} in Appendix A.

Static Semantics. OHMM's typing rules have the form $\Gamma \vdash e \triangleright T$, meaning that the expression e has type T when type-checked against Γ. These rules are presented in Appendix A.3. Type checking is mostly standard. Note that we require that

that objects allocated in the same pool have the same type, as different types might differ in size, and in their structures. While heterogeneity is possible, *e.g.*, by treating all objects in a pool as the size of the largest type, we exclude it from this treatise. Therefore, in addition to the mapping between variables and types, the typing context also keeps a mapping from locations to types:

$$\Gamma \in \mathit{TypingContext} = ((\mathit{Variable} \cup \{\mathsf{this}\}) \to \mathit{Type}) \cup (\mathit{Location} \to \mathit{ClassId})$$

Note the rule POOL which assigns a class identifier to the newly created pool, and the rule NEW2 which used a definition of well-formed type to ensure that an instance of class C is only allocated in a pool of class C. Type well-formedness is defined as:

$$\begin{aligned}\Gamma \vdash C\langle l_1, ..., l_n\rangle \Leftrightarrow |\mathcal{O}(C)| = n \ \wedge \ \{l_1, ..., l_n\} \subseteq (\mathrm{dom}(\Gamma) \cup \{\mathsf{none}\}) \\ (l_1 \neq \mathsf{none} \implies \Gamma(l_1) = C)\end{aligned}$$

It requires that the number of locations passed is the same as the number of class parameters, that all the locations used are in scope or are none, and if the first location used, the object's location, is a pool, then this pool contains objects of this type only. The following class shows examples of two ill-formed types and one well-formed type.

```
class C⟨o1, o2⟩
  f1: F⟨o1⟩ ✗ // o1 has instances of C, not of F
  f2: C⟨o1⟩ ✗ // wrong number of pool parameters to C
  f3: C⟨o1, o2⟩ ✓
```

Safe Data Layouts. OHMM^{hl} is type safe, in the sense that if a well-typed expression is reduced in the context of a well-formed configuration, the type of the expression and the well-formedness of the configuration are preserved.

Even though OHMM^{hl}'s semantics does not reflect the actual program execution, because of its type soundness, we can guarantee that programs *correctly* compiled from OHMM^{hl}, when executed, result in well-formed configurations, and thus well-formed layouts. And well-formed layouts exclude, for instance, objects of different sizes in the same pool. Compiler correctness is discussed later. We state type soundness as below.

Theorem 1 (Type Soundness). *For any heap χ, stack frame φ, typing context Γ, and expression e, if 1) the high-level configuration composed by χ and φ is well-formed under Γ, and 2) the expression has type T under the same context, and 3) it reduces with χ and φ to a new heap χ' and a value β, then χ' is also well-formed under the same context Γ and β has the same type T.*

$$\begin{aligned}&\forall \chi, \varphi, \Gamma, e.\\ &\quad \textbf{If } \Gamma \vDash \chi, \varphi \ \wedge \ \Gamma \vdash e \triangleright T \ \wedge \ \chi, \varphi, e \rightsquigarrow \chi', \beta \textbf{ then } \Gamma \vDash \chi', \varphi \ \wedge \ \chi' \vDash \beta \triangleleft T\end{aligned}$$

Proof. By structural induction over the derivation $\chi, \varphi, e \rightsquigarrow \chi', \beta$. *c.f.* Appendix G.

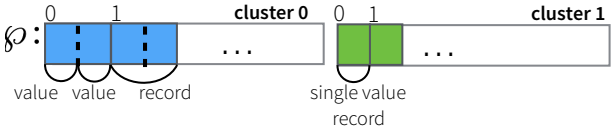
4 OHMM^{ll}: Abstract Placement Control

While OHMM^{hl} semantics is the abstract semantics offered to the programmer, OHMM^{ll} programs are supposed to be executed so that objects may be pooled and split. To give semantics to this, we define a low level language, OHMM^{ll} and its runtime. Therefore in OHMM^{ll}, contrarily to OHMM^{hl}, pooling information represents the actual location of objects[6]. In addition, if an object is allocated in a pool it may be that it is split in different parts, called *records* from now on. Moreover, while objects floating in memory can be found in addresses, objects in pools cannot. OHMM^{ll}'s runtime entities, defined as follows, take all this into account.

$$
\begin{aligned}
h \in \mathit{Heap}^{ll} &= (\mathit{ObjAddr} \to \mathit{Object}^{ll}) \\
&\quad \cup (\mathit{PoolAddr} \to \mathit{Pool}) \\
ob \in \mathit{Object}^{ll} &= \mathit{ClassId} \times (\mathit{FieldId} \to \mathit{Value}^{ll}) \\
\mathit{pool} \in \mathit{Pool} &= \mathit{ClassId} \times \mathit{Cluster}^{\star} \\
\mathit{clt} \in \mathit{Cluster} &= \mathit{Record}^{\star} \\
\mathit{rec} \in \mathit{Record} &= \mathit{Value}^{ll\star}
\end{aligned}
\qquad
\begin{aligned}
v \in \mathit{Value}^{ll} &= \{\mathsf{null}\} \cup \mathit{ObjAddr} \\
&\quad \cup \mathit{Reference} \cup \mathit{PoolAddr} \\
\mathit{Reference} &= \mathit{PoolAddr} \times \mathbb{N} \\
\mathit{fr} \in \mathit{SFrame}^{ll} &= \mathit{Register} \to \mathit{Value}^{ll} \\
\ell &\in \mathit{ObjAddr} \\
\wp &\in \mathit{PoolAddr}
\end{aligned}
$$

Fig. 9: Low level runtime entities.

Note that in order to keep our model as simple as possible we assume: 1) every value, of any type, fits in one byte; and 2) pools are of infinite size. We explain how to deal with different value sizes and finite pools in Section 7. A heap h is a mapping from object addresses ℓ to objects (somewhere in memory) and from pool addresses $\wp$ to pools. An object is composed by a class identifier and a mapping from fields to values. Pooled objects have a more complex structure. In order to understand them, consider the following pool diagram.



The pool above has two instances of the class Video located at $(\wp, 0)$ and $(\wp, 1)$, and distributed over two clusters. A pool contains a sequence of clusters (named cluster 0 and cluster 1 above). Each cluster is a sequence of records, which are sequences of values. Values can be object addresses, references to pooled objects, pool addresses, or just null.

Thus, pools contain values without any information regarding which fields are pointing to these values. Because of this and because each object may be represented by several records (one per cluster), reading and writing to objects' fields becomes more complex. To address this, records must be aligned, that is,

[6] We use objects to refer to low-level data in memory, in order to use our running example and keep consistency with the previous sections. However OHMM^{ll} does not require the use of an object-oriented language.

records which are part of the same object can be found in different clusters, but at the same index. For instance, if we consider that $clts[0]$ is cluster 0 and $clts[1]$ is cluster 1 of the diagram above, then $clts[0][1]$ and $clts[1][1]$ together represent a full object. Accessing a specific field of an object must take into account its position within the record. For instance the field likes from the second video is stored at $clts[0][1][1]$.

This means that rather than using field identifiers to access this specific value, the programmer, or the compiler, should use two indices: the cluster position, and the value position (within in the record). This is valid for reading from objects inside pools; when reading from objects floating somewhere in memory, the use of a field identifier is enough. OHMM^{ll} therefore provides two different instructions for pool reading and for object reading: $\mathsf{pread}(r, i, j)$ and $\mathsf{read}(r, f)$, where r is a register which contains the location of the object being read, i, j are integers and f is a field identifier. Similarly, writing to objects, and creating objects also require two different instructions. The syntax of these instructions is in Fig. 10.

$$
\begin{array}{ll}
P^{ll} ::= fnd^{\star} & rhs ::= fn(rs) \mid \mathsf{null} \\
fnd ::= \mathsf{fun}\ fn(r_{this}, r_x, r^{\star}) \Rightarrow ss & \quad \mid \mathsf{pread}(r, j, k) \mid \mathsf{read}(r, f) \\
ss ::= a^{\star}; r & \quad \mid \mathsf{pwrite}(r, j, k, r') \mid \mathsf{write}(r, f, r') \\
a ::= r \coloneqq rhs & \quad \mid \mathsf{pcreate}(C) \mid \mathsf{palloc}(r) \mid \mathsf{alloc}(C) \\
r \in Register \quad fn \in FunctionId & \quad i, j, k, n \in \mathbb{N}
\end{array}
$$

Fig. 10: OHMM^{ll} syntax and notation.

The syntax of instructions (rhs) also provides for pool creation, and function calls. A low level program consists of a sequence of function declarations. Each function takes as parameters: a register r_{this}, as a ghost receiver of that function; a register r_x as the function parameter, and a sequence of registers which store pool addresses. The function body is a sequence of assignments followed by a register.

We now introduce the semantics of OHMM^{ll}. A heap, a stack frame, and a sequence of statements reduce to a new heap and a value, thus:

$$
\frac{h, fr, rhs \rightsquigarrow h', v \quad h', fr[r \mapsto v], ss \rightsquigarrow h'', v'}{h, fr, r \coloneqq rhs; ss \rightsquigarrow h'', v'}\ [\textsc{reduce-rhs}] \qquad \frac{}{h, fr, r \rightsquigarrow h, fr(r)}\ [\textsc{reg}]
$$

The rule REDUCE-RHS, for statements of the form $r \coloneqq rhs; ss$, evaluates rhs with heap h and stack frame fr, resulting in a new value which is then assigned to register r in the stack frame, so that it can be used in the following statements. Fig. 11 shows the semantic rules for right-hand sides.

Note in particular the function call instruction that also passes as arguments the registers that point to the pools used in the function's body. The object allocation instruction is also quite interesting. While the allocation of an object somewhere in memory is straightforward, allocating an object in a pool, through the instruction $\mathsf{palloc}(r)$ looks up the pool address $\wp$ stored in the register r

$$\frac{fr(r) = (\wp, n) \quad h(\wp) = (C, clts) \quad clts[j, n, k] = v}{h, fr, \mathsf{pread}(r, j, k) \rightsquigarrow h, v}\ [\textsc{pread}] \qquad \frac{fr(r) = \ell \quad h(\ell, f) = v}{h, fr, \mathsf{read}(r, f) \rightsquigarrow h, v}\ [\textsc{read}]$$

$$\frac{fr(r) = (\wp, n) \quad fr(r') = v \quad h(\wp) = (C, clts) \quad clts' = clts[(j, n, k) \mapsto v])}{h, fr, \mathsf{pwrite}(r, j, k, r') \rightsquigarrow h[\wp \mapsto (C, clts')], v}\ [\textsc{pwrite}]$$

$$\frac{fr(r) = \ell \quad fr(r') = v}{h, fr, \mathsf{write}(r, f, r') \rightsquigarrow h[(\ell, f) \mapsto v], v}\ [\textsc{write}]$$

$$\frac{fr(r) = \wp \quad h(\wp) = (C, clt_1, ..., clt_j) \quad \forall k \in 1..j.clt'_k = clt_k :: \text{initRec}(C, k)}{h, fr, \mathsf{palloc}(r) \rightsquigarrow h[\wp \mapsto (C, clt'_1, ..., clt'_j)], (\wp, |clt_1|)}\ [\textsc{palloc}]$$

$$\frac{\ell \text{ fresh in } h \quad \text{initObj}(C) = ob}{h, fr, \mathsf{alloc}(C) \rightsquigarrow h[\ell \mapsto ob], \ell}\ [\textsc{alloc}]$$

$$\frac{\wp \text{ fresh in } h \quad n = |\Sigma(C)\downarrow_2| \quad \forall i \in 1..n.clt_i = \emptyset}{h, fr, \mathsf{pcreate}(C) \rightsquigarrow h[\wp \mapsto (C, clt_1, ..., clt_n)], \wp}\ [\textsc{pcreate}]$$

$$\frac{}{h, fr, \mathsf{null} \rightsquigarrow h, \mathsf{null}}\ [\textsc{null}]$$

$$\frac{\mathcal{F}\mathsf{un}(fn) = (ss, rs) \quad h, (r_{this} \mapsto fr(r), r_x \mapsto fr(r'), rs \mapsto fr(r'_1...r'_n)), ss \rightsquigarrow h', v}{h, fr, fn(r, r', r'_1, ..., r'_n) \rightsquigarrow h', v}\ [\textsc{fun}]$$

Fig. 11: Operational semantics of OHMM^{ll}.

and adds an initialised record to each cluster of $\wp$. Records are initialised using the pool structure information which should be provided by the programmer, or generated from the high-level code. This information is kept in a table Σ, which is defined as follows:

$$\Sigma \in \mathit{Structure} = \mathit{ClassId} \to (\mathit{FieldId}^\star \times \mathit{PoolStruct})$$
$$ps \in \mathit{PoolStruct} = \mathit{ClusterStruct}^\star \qquad cs \in \mathit{ClusterStruct} = \mathit{ClassId}^\star$$

Σ maps class identifiers to sequences of fields (of objects somewhere in memory) and to pool structures. The structure of a pool is composed by the structure of all its clusters; the structure of a cluster is composed by the type of the values that are allocated in each of its records.

5 Translating OHMM^{hl} to OHMM^{ll}

Program translation consists of two steps: 1) generation of the structure information, Σ, out of the high-level class declarations; and 2) generation of low-level functions from the high-level methods. The definitions of these two steps can be found in Appendix B. The first step takes, for instance, the Video class declaration and adds to Σ that Video has fields id, likes, and views and that its pool allocated instances are split into two clusters structured (int, int) and (int). Σ for our example is:

$$\Sigma(\mathsf{Video}) = ((\mathsf{id}, \mathsf{likes}, \mathsf{views}), ((\mathsf{int}, \mathsf{int}), (\mathsf{int})))$$
$$\Sigma(\mathsf{Node}) = ((\mathsf{video}, \mathsf{next}), (\mathsf{Video}, \mathsf{Node})) \qquad \Sigma(\mathsf{VideoList}) = ((\mathsf{head}), (\mathsf{Node}))$$

In the second step, each OHMM^{hl} method declaration is compiled to one or more OHMM^{ll} function declarations. Given that class declarations may be instantiated with different location arguments, their methods may have different memory access behaviours. For example, consider typing contexts Γ and Γ', variable

x, and pool identifier p, such that $\Gamma(x) = \mathsf{Video}\langle\mathsf{none}\rangle$, $\Gamma'(x) = \mathsf{Video}\langle p\rangle$, and $\Gamma'(p) = \mathsf{Video}\langle\mathsf{none}\rangle$. The expression x.id will be translated under context Γ to read(r, id), and under context Γ' to pread(r, 0, 0), assuming that there is a Δ that maps x to r. This means that for each method declaration, and for each possible combination of location arguments, a different low-level function declaration is generated. For instance, the VideoList class declaration may take eight different combinations of location arguments, therefore the method popularVideos results in eight different function declarations. An example of one of these functions, corresponding to the case where a VideoList takes arguments $\langle\mathsf{none}, \mathsf{p}_1, \mathsf{p}_2\rangle$, is:

fun popularVideos_n_p1_p2($\mathsf{r}_{\mathsf{this}}$, r_{x}, r_1, r_2) ⇒//transl. of popularVideos method's body

Its parameters are 4 registers: the first two for the this and x variables, and the other two for the pools p_1 and p_2 that will be used in its body.

We now discuss the translation of OHMM^{hl} terms to OHMM^{ll} terms. Because this translation is based on type information, it takes place simultaneously with type checking, similarly to Java[7]. Therefore, these rules show the addition of translation work into the typing rules. We do not describe further OHMM's typing as this was explained in the previous section.

OHMM's translation rules require a typing context Γ and a register map

$$\Delta \in \mathit{RegisterMap} = ((\mathit{Variable} \cup \{\mathsf{this}\} \cup \mathit{PoolId}) \to \mathit{Register}) \times \mathit{Register}^\star$$

which maps high-level variables and pool identifiers to low-level registers. It also keeps a sequence of registers, in order to avoid repetitions of registers within the same sequence of statements. All auxiliary definitions are in appendix F; note in particular, the predicate $T@\mathsf{none}$ which indicates whether the type T describes an object in an none location or not.

OHMM's typing and translation rules are given in Fig. 12. These rules have the form $\Delta, \Gamma \vdash e \mapsto T, ss, \Delta'$, where the blue highlight shows the parts concerned with type checking. They mean that given a register map Δ, and a typing context Γ, the expression e has type T and is translated to ss, and the register map is updated to Δ' with the registers introduced by translating the expression.

The most interesting translation rules are the ones that depend on types. For instance, the translation of object creation of type $C\langle ls\rangle$ (rules NEW1 and NEW2) depends on the first location, $ls[0]$: if it is none, then it generates a pool allocation (palloc), otherwise it it generates a regular allocation (alloc). Translating a field read expression depends on the type of the receiver; if it is a type of a pooled object ($\neg T@\mathsf{none}$) then it generates a pool read (pread), otherwise it generates a regular read (read). The former takes two indices i, j which indicate the cluster, and the position of the value pointed by the field. These indices may be obtained statically (*c.f.* field lookup and $\mathcal{W}$ definitions in appendix F). Translating field write expressions follows a similar approach.

Special care must be taken when compiling a method call expression. Given that a low-level function call requires to have in scope the registers that point to all the pools its body will use, the compilation must find what are these registers.

[7] For typing rules on their own *c.f.* Appendix A.3

$$\frac{}{\Delta, \Gamma \vdash x \mapsto \Gamma(x), \Delta(x), \Delta}\ [\textsc{Var}] \qquad \frac{\Gamma \vdash T \quad r \notin \Delta}{\Delta, \Gamma \vdash \mathsf{null} \mapsto T, r = \mathsf{null}; r, \Delta \oplus r}\ [\textsc{Null}]$$

$$\frac{\Delta, \Gamma \vdash e \mapsto T, as; r, \Delta' \quad \Delta'[x \mapsto r], \Gamma[x \mapsto T] \vdash e' \mapsto T', ss', \Delta''}{\Delta, \Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \mapsto T', as; ss', \Delta''}\ [\textsc{Let}]$$

$$\frac{r \notin \Delta \quad r \notin \Delta' \quad a = (r \coloneqq \mathsf{pcreate}(C)) \quad \Delta[p \mapsto r], \Gamma[p \mapsto C] \vdash e \mapsto T, ss', \Delta'}{\Delta, \Gamma \vdash \mathsf{pool}\ p\ \mathsf{of}\ C\ \mathsf{in}\ e \mapsto T, a; ss', \Delta' \oplus r}\ [\textsc{Pool}]$$

$$\frac{r \notin \Delta \quad T = C\langle \mathsf{none}, ls \rangle \quad \Gamma \vdash T \quad a = (r \coloneqq \mathsf{alloc}(C))}{\Delta, \Gamma \vdash \mathsf{new}\ T \mapsto T, a; r, \Delta \oplus r}\ [\textsc{New1}]$$

$$\frac{r \notin \Delta \quad T = C\langle p, ls \rangle \quad \Gamma \vdash T \quad \Delta(p) = r' \quad a = (r \coloneqq \mathsf{palloc}(r'))}{\Delta, \Gamma \vdash \mathsf{new}\ T \mapsto T, a; r, \Delta \oplus r}\ [\textsc{New2}]$$

$$\frac{\Delta, \Gamma \vdash e \mapsto T, as; r, \Delta' \quad \mathcal{F}(T, f) = T' \quad T@\mathsf{none} \quad r' \notin \Delta' \quad a = (r' \coloneqq \mathsf{read}(r, f))}{\Delta, \Gamma \vdash e.f \mapsto T', as; a; r', \Delta' \oplus r'}\ [\textsc{FRead1}]$$

$$\frac{\Delta, \Gamma \vdash e \mapsto T, as; r, \Delta' \quad \mathcal{F}(T, f) = (T', i, j) \quad \neg(T@\mathsf{none}) \quad r' \notin \Delta' \quad a = (r' \coloneqq \mathsf{pread}(r, i, j))}{\Delta, \Gamma \vdash e.f \mapsto T', as; a; r', \Delta' \oplus r'}\ [\textsc{FRead2}]$$

$$\frac{\Delta, \Gamma \vdash e \mapsto T, as; r, \Delta' \quad T@\mathsf{none} \quad \Delta', \Gamma \vdash e' \mapsto T', as'; r', \Delta'' \quad \mathcal{F}(T, f) = T' \quad r'' \notin \Delta'' \quad a = (r'' \coloneqq \mathsf{write}(r, f, r'))}{\Delta, \Gamma \vdash e.f = e' \mapsto T', as; as'; a; r'', \Delta'' \oplus r''}\ [\textsc{FWrite1}]$$

$$\frac{\Delta, \Gamma \vdash e \mapsto T, as; r, \Delta' \quad \mathcal{F}(T, f) = (T', i, j) \quad \Delta', \Gamma \vdash e' \mapsto T', as'; r', \Delta'' \quad \neg(T@\mathsf{none}) \quad r'' \notin \Delta'' \quad a = (r'' \coloneqq \mathsf{pwrite}(r, i, j, r'))}{\Delta, \Gamma \vdash e.f = e' \mapsto T', as; as'; a; r'', \Delta'' \oplus r''}\ [\textsc{FWrite2}]$$

$$\frac{\Delta, \Gamma \vdash e \mapsto T, as; r, \Delta' \quad \Delta', \Gamma \vdash e' \mapsto T', as'; r', \Delta'' \quad \mathcal{M}_T(T, m) = (T'', T', \_, fn) \quad rs = regs_{\Delta''}(T) \quad r'' \notin \Delta'' \quad a = (r'' \coloneqq fn(r, r', rs))}{\Delta, \Gamma \vdash e.m(e') \mapsto T'', as; as'; a; r'', \Delta'' \oplus r''}\ [\textsc{Call}]$$

Abbreviation: $\Gamma \vdash e \triangleright T \Leftrightarrow \exists \Delta.\ \Delta, \Gamma \vdash e \mapsto T, \_, \_$

Fig. 12: Expression type checking and translation.

It does so by looking at the type (to its locations, in particular) of the receiver and by using the register map (*c.f.* $reg_{\Delta}(T)$ of Appendix F).

We now exemplify how an high-level expression can be translated to low-level statements. Table 2 shows the translation of the body of the method popularVideos into the function popularVideos_n_p1_p2. The translation of this function takes a register map Δ, and a typing context Γ, such that

$\Delta(\mathsf{this}) = r_{this} \ \wedge\ \Delta(\mathsf{above}) = r_x \ \wedge\ \Delta(p_1) = r_1 \ \wedge\ \Delta(p_2) = r_2$

$\Gamma(\mathsf{this}) = \mathsf{VideoList}\langle \mathsf{none}, p_1, p_2 \rangle$

In Table 2, the first column contains the high-level expression to be translated, the second column the low-level statement obtained, and the third column justifies the translation[8]. We show the low level code for more functions in Appendix E.

6 OHMM's Translation Correctness

In this section we state the correctness of our translation definition. For this we need to first define a correspondence between high and low level runtime configuration. This is defined in terms of a bijection that maps high-level addresses and locations to low-level addresses:

[8] Note that due to space limitations, rather than using the field lookup function as in the typing rules of Fig. 12, we use the $\mathcal{W}$ function (*c.f.* Appendix F)

OHMM^{hl}	OHMM^{ll}	
let cur = this.head in	r_0=read(r_{this}, head);	Δ(this) = r_{this} $\wedge$ Γ(this)@none
while (cur not null)	while ($r_0$not null)	
let v = cur.video in	r_1= pread(r_0,0,0)	$\neg\Gamma$(cur)@none $\wedge$ $\mathcal{W}$(Node, video) = (0, 0)
if v.views gt pivot then	r_2= pread(r_1, 1, 0); if $r_2$gt $r_x$then	$\neg\Gamma$(v)@none $\wedge$ $\mathcal{W}$(Video, views) = (1, 0)
print(v.id, v.views, v.likes);	 r_3= pread(r_1, 0, 0); r_4= pread(r_1, 1, 0); r_5= pread(r_1, 0, 1); r_6= print(r_3, r_4, r_5);	Δ(v) = r_1 $\wedge$ $\neg\Gamma$(this)@none $\wedge$ $\mathcal{W}$(Video, id) = (0, 0) $\mathcal{W}$(Video, views) = (1, 0) $\mathcal{W}$(Video, likes) = (0, 1)
cur = cur.next	r_0= pread(r_0,0,1)	$\neg\Gamma$(cur)@none $\wedge$ $\mathcal{W}$(Node, next) = (0, 1)

Table 2: Example of compilation of an high-level expression.

$B : (Address \to (ObjAddr \cup Reference)) \times (PoolLoc \to PoolAddr)$

Using such a bijection, we define configuration equivalence, $\chi, \varphi \simeq_{B\Delta} h, fr$, heap equivalence, $\chi \simeq_B h$, stack frame equivalence, $\varphi \simeq_{B\Delta} fr$, and value equivalence $\beta \simeq_B v$, in Appendix C. OHMM's correctness is expressed in terms of Theorems 2 and 3. They state that OHMM's translation preserve meaning, in the sense that the execution of a low-level code has an equivalent result to the execution of the high-level code from which it was generated, and vice-versa. Note that both theorems rely on the following definition

$$B \leq B' \Leftrightarrow \mathrm{dom}(B) \subseteq \mathrm{dom}(B') \ \wedge \ \forall \alpha \in \mathrm{dom}(B).B(\alpha) = B'(\alpha)$$

which is a natural definition of a bijection B' extending a bijection B.

Theorem 2 (Compilation is sound).

$\forall \chi, \varphi, h, fr, B, \Gamma, \Delta, e.$

If $\Gamma \vDash \chi, \varphi \ \wedge \ \chi, \varphi \simeq_{B\Delta} h, fr \ \wedge \ \Delta, \Gamma \vdash e \triangleright \_, ss \ \wedge \ h, fr, ss \rightsquigarrow h', v$

then $\exists B', \chi', \beta.\ (\chi, \varphi, e \rightsquigarrow \chi', \beta \ \wedge \ B \leq B' \ \wedge \ \chi' \simeq_{B'} h' \ \wedge \ \beta \simeq_{B'} v)$

Proof. By case analysis on e. *c.f.* Appendix G.

Theorem 3 (Compilation is complete).

$\forall \chi, \varphi, h, fr, B, \Gamma, \Delta, e.$

If $\Gamma \vDash \chi, \varphi \ \wedge \ \chi, \varphi \simeq_{B\Delta} h, fr \ \wedge \ \Delta, \Gamma \vdash e \triangleright \_, ss \ \wedge \ \chi, \varphi, e \rightsquigarrow \chi', \beta$

then $\exists B', h', v.\ (h, fr, ss \rightsquigarrow h', v \ \wedge \ B \leq B' \ \wedge \ \chi' \simeq_{B'} h' \ \wedge \ \beta \simeq_{B'} v)$

Proof. By structural induction over the derivation $\chi, \varphi, e \rightsquigarrow \chi', \beta$. *c.f.* Appendix G.

7 Low-level Prototype Implementation

We have implemented a C framework which corresponds to OHMM^{ll}. The code can be found in: https://github.com/jupvfranco/ohmm. More details about this implementation, as well as promising benchmark results can be found in [13].

We now describe how the implementation currently goes beyond OHMM^{ll}.

7.1 Pool Allocation

The system allows the allocation of objects in pools, as described in previous sections. Pools are constructed from sub-pools, each with space for 4096 objects. Whenever a sub-pool is completely full, another one is created allowing for more 4096 objects (creating a collection of finite sub-pools). In order to accommodate values of different sizes, we keep size information together with type information, and use it to calculate the offset of objects in pools.

7.2 Pointer Compression

The system optionally allows the compression of references to data inside the same pool. For instance, in the VideoList example, all the next fields point to Nodes within the same pool, therefore rather than using an entire 64 bit reference, it is possible to use a compressed reference—the pool offset where the referred object is allocated. And because each sub-pool contains 4096 objects only, this compressed reference fits in 12 bits. In the end, compressed pointers use 16 bits including 4 bits to handle references to objects in other sub-pools through one indirection. We leave addition of this feature to the formal model for future work.

7.3 Object Splitting

The system currently only supports *maximally* splitting objects within pools, that is, a 1–1 mapping between clusters and fields. In order to do so, it uses a data structure that contains all the fields of a type and their sizes. This data structure plays a similar role to the pool structure Σ, defined before.

7.4 Copying Garbage Collector

Garbage collection can be used to reduce memory fragmentation. Currently the framework implements a simple copying garbage collector, where each pool can be collected separately, and where collections are triggered manually.

7.5 Future Work: Extending the Framework

We now briefly describe some extensions to our framework.

Per-Object Pools In the current formalisation, pools are declared on the stack outside of the data structures that use them. This is in contrast to ownership types whose owning contexts are created and destroyed along with objects. Adding the ability for *e.g.*, a list to create a pool to hold its nodes is a sensible and straightforward addition to our work. The current cost of a pool is oversubscription of virtual memory, to allow pools to grow continuously up to a limit of 4 GB. To allow pools to be used more pervasively for small data structures, we will add more configuration parameters to pool creation, *e.g.*, with a configurable upper limit. Finally, we note that allowing pool parameters to be instantiated external to a class facilitates re-use across different scenarios in a way that is less accessible if pool details are encapsulated.

Subtyping Adding subtyping to OHMM will raise issues regarding the size of objects allocated in pools, as a class may contain fields that are not existent in its superclass. Consider we add to our example, the following class:

```
class MusicVideo extends Video
  artistId: int
```

An object of type MusicVideo also has type Video and therefore should be allowed to be allocated in a pool of Videos. However this object would require more space than an instance of the class Video, as it need to accommodate one more field (the artistId field in addition to the three fields inherited from the class Video). We intend to take advantage of object splitting in order to deal with this problem.

Optimising Layout with Garbage Collection For the time being, consider that we extend the popularVideos method in the following ways: 1) if the videos are not popular enough, then they are removed. 2) if the video is not popular enough, rather than removing it, it is swapped with the video in the head of the list.

Executing popularVideos with the first extension makes some of the videos to become unreachable; upon garbage collection their memory locations would be freed, thus creating empty gaps in the pool of Videos. These gaps would lead to fetching *void* data to cache, possibly provoking more cache misses. Using a *compacting* garbage collector it would be possible to avoid such situations.

Executing popularVideos with the second extension reorders the list. This means that the alignment between the pool of Nodes and the pool of Videos in Fig. 7 and Fig. 8 is no longer observed. With this layout, fetching a Video to cache might mean fetching other Videos that will not be used soon, or will not be used at all. This layout would also lead to more cache misses. A *moving* garbage collector would reorder pools to avoid these cache misses. However, pool reordering brings a different problem. Such an operation may be very expensive depending on the size of the pool; and if, for instance, the list is never used again, it is not worth it to change the pool. Moreover, moving objects in a system where aliases are allowed can be challenging, as well. In order to deal with the first problem we intend to apply techniques as static analysis to find the iteration patterns of the program. In order to tackle the second problem, we intend to explore well-known ownership types techniques to deal with aliasing [7].

8 Related Work

Pooling and splitting are two techniques often used to improve programs' performance through better data layout. The idea of data placement to reduce cache misses was first introduced by Calder et al. [4], where the authors apply profiling techniques to find temporal relationships among objects. This work was then followed up by Lattner et al. [17,18] where rather than relying on profiling, static analysis of C and C++ programs finds what layout to use. Huang et al. [16] explore pool allocation in the context of Java. Object Splitting was introduced by Franz and Kistler [12], where fields are classified as hot (accessed frequently) and cold (accessed less frequently); this classification is used to decide how to

split objects. Since then splitting has been combined with pooling [5,10,21,22,23]. Another interesting work is presented by Hirzel [15], and uses a copying garbage collector in order to implement several data layouts of object oriented programs and evaluate which layout presents the best performance.

Tofte and Talpin introduced the concept of region-based memory management [19,20]. They use region types, which divide memory in regions, in an ML language, where allocation and deallocation are inferred from type and effect analysis. This idea was then used in the Cyclone language [14], which is concerned with safety of C-like languages. Other language that also provides means to split data in the heap, is the Deterministic Parallel Java where code is annotated with regions information and it is possible to calculate the effects of reading and writing to data [3]. Note that these languages only divide the heap *conceptually*.

There are as well some programming languages that split the heap in several sub-heaps in order to simplify garbage collection or parallelism. Examples of these languages are Pony [8,9], Erlang [1,2] and Loci [24]. None of these languages share goals with OHMM, in the sense that they do not try to improve data locality. and particularly in Pony and Erlang, the programmer does not have any control on how to divide the heap.

OHMM is not the first time that ownership types are used to express object layouts. In the context of NUMA systems, Franco and Drossopoulou, use similar annotations to describe in which NUMA nodes the objects should be placed [11]. The final goal of this work was to improve program performance by reducing memory accesses *to remote nodes*, ignoring any possible in-cache data accesses.

9 Conclusion

This paper introduces the OHMM framework. OHMM is composed of a high-level object-oriented language, and a low-level language, which is the interface for a C implementation that provides pooling and object splitting. We have presented the high-level language using a running example, and we have formalised both languages, as well as the translation from the high-level to the low-level. We have argued that the high-level language is type sound and that the compilation to the low-level language is correct.

References

1. J. Armstrong. A History of Erlang. In *HOPL*, pages 6–1. ACM, 2007.
2. J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent Programming in Erlang. 1993.
3. 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.
4. B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data Placement. In *ASPLOS VIII*, pages 139–149. ACM, 1998.
5. T. M. Chilimbi and R. Shaham. Cache-Conscious Coallocation of Hot Data Streams. In *PLDI '06*, pages 252–262. ACM, 2006.

6. D. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. *SIGPLAN Not.*, 37(11):292–310, 2002.
7. D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA '98*, pages 48–64. ACM, 1998.
8. S. Clebsch, S. Blessing, J. Franco, and S. Drossopoulou. Ownership and Reference Counting Based Garbage Collection in the Actor World. In *ICOOOLPS'2015*.
9. S. Clebsch and S. Drossopoulou. Fully Concurrent Garbage Collection of Actors on Many-Core Machines. In *OOPSLA'2013*, pages 553–570. ACM, 2013.
10. S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, R. Silvera, and R. Archambault. Mpads: Memory-Pooling-Assisted Data Splitting. In *ISMM '08*, ACM, 2008.
11. J. Franco and S. Drossopoulou. Behavioural Types for Non-Uniform Memory Accesses. *PLACES 2015*, page 39, 2015.
12. M. Franz and T. Kistler. Splitting Data Objects to Increase Cache Utilization. Technical report, University of California, Irvine, 1998.
13. M. Hagelin. Optimizing memory management with object-local heaps. Master's thesis, Department of Information Technology, Uppsala University, 2015.
14. M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with Safe Manual Memory-Management in Cyclone. In *ISMM'04*, pages 73–84. ACM, 2004.
15. M. Hirzel. Data Layouts for Object-Oriented Programs. In *ICMMCS*, pages 265–276. ACM, 2007.
16. 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'04*.
17. C. Lattner and V. Adve. Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis. Technical report, U. of Illinois, 2003.
18. C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI '05*, ACM, 2005.
19. M. Tofte and J.-P. Talpin. Implementation of the Typed Call-by-Value λ-Calculus Using a Stack of Regions. In *POPL '94*, pages 188–201. ACM, 1994.
20. M. Tofte and J.-P. Talpin. Region-Based Memory Management. *Inf. Comput.*, 132(2):109–176, 1997.
21. H. L. A. van der Spek, C. W. M. Holm, and H. A. G. Wijshoff. Automatic Restructuring of Linked Data Structures. In *LCPC'09*, Springer, 2010.
22. Z. Wang, C. Wu, and P.-C. Yew. On Improving Heap Memory Layout by Dynamic Pool Allocation. In *CGO '10*, pages 92–100. ACM, 2010.
23. Z. Wang, C. Wu, P.-C. Yew, J. Li, and D. Xu. On-the-fly Structure Splitting for Heap Objects. *ACM TACO*, 8(4):26:1–26:20, 2012.
24. 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.
25. 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.

A OHMM^{hl} in More Detail

A.1 Runtime entities

The runtime entities of OHMM^{hl} are defined in Fig. 13.

$$\begin{aligned}
\chi \in \mathit{Heap} &= (\mathit{Address} \to \mathit{Object}) \times \mathit{PoolLoc}^\star \\
\varphi \in \mathit{SFrame} &= (\mathit{Variable} \to \mathit{Value}) \cup ((\mathit{PoolId} \cup \mathit{OwnerParam}) \to \mathit{PoolLoc}) \\
\omega \in \mathit{Object} &= \mathit{ClassId} \times \mathit{HeapLoc}^\star \times (\mathit{FieldId} \to \mathit{Value}) \\
\beta \in \mathit{Value} &= \{\mathsf{null}\} \cup \mathit{Address} \\
\lambda \in \mathit{HeapLoc} &= \{\mathsf{none}\} \cup \mathit{PoolLoc} \qquad \pi \in \mathit{PoolLoc} \qquad \alpha \in \mathit{Address}
\end{aligned}$$

As abbreviation we use: $\alpha \in \chi \Leftrightarrow \alpha \in \mathrm{dom}(\chi)\downarrow_1 \qquad \pi \in \chi \Leftrightarrow \alpha \in \chi\downarrow_2$

Fig. 13: Dynamic Entities of OHMM^{hl}.

A heap χ maps addresses (α) to objects (ω), and it contains a sequence of pool locations (π), as ghost information. An object is composed by a class identifier, a sequence of heap locations, and a mapping from fields to values. For instance, $(\mathsf{Video}, \mathsf{none}, (\mathsf{id} \mapsto 112, \mathsf{likes} \mapsto 245452, \mathsf{views} \mapsto 1234566))$ is an object. A heap location λ can be either a pool location π or none. The *first* heap location of an object indicates where the object is allocated: if in a pool π or if somewhere in memory (none). A value β is either null or an address.

A.2 Dynamic semantics

OHMM^{hl} semantic rules are defined in Fig. 14. They have the form $\chi, \varphi, e \rightsquigarrow \chi', \beta$ where each rule takes a heap, a stack frame and an expression and reduces to a new heap and a value.

$$\frac{}{\chi, \varphi, x \rightsquigarrow \chi, \varphi(x)}\textsc{Var} \qquad \frac{}{\chi, \varphi, \mathsf{null} \rightsquigarrow \chi, \mathsf{null}}\textsc{Val} \qquad \frac{\alpha \notin \chi \quad \varphi(ls) = \lambda s \quad \omega = \mathrm{init}(C\langle \lambda s \rangle)}{\chi, \varphi, \mathsf{new}\ C\langle ls \rangle \rightsquigarrow \chi[\alpha \mapsto \omega], \alpha}\textsc{New}$$

$$\frac{\pi \notin \chi \quad \chi \oplus \pi, \varphi[p \mapsto \pi], e \rightsquigarrow \chi', \beta}{\chi, \varphi, \mathsf{pool}\ p\ \mathsf{of}\ C\ \mathsf{in}\ e \rightsquigarrow \chi', \beta}\textsc{Pool} \qquad \frac{\chi, \varphi, e \rightsquigarrow \chi', \beta \quad \chi', \varphi[x \mapsto \beta], e' \rightsquigarrow \chi'', \beta''}{\chi, \varphi, \mathsf{let}\ x = e\ \mathsf{in}\ e' \rightsquigarrow \chi'', \beta''}\textsc{Let}$$

$$\frac{\chi, \varphi, e \rightsquigarrow \chi', \alpha \quad \chi'(\alpha, f) = \beta}{\chi, \varphi, e.f \rightsquigarrow \chi', \beta}\textsc{FRead} \qquad \frac{\chi, \varphi, e \rightsquigarrow \chi', \alpha \quad \chi', \varphi, e' \rightsquigarrow \chi'', \beta}{\chi, \varphi, e.f = e' \rightsquigarrow \chi''[(\alpha, f) \mapsto \beta], \beta}\textsc{FWrite}$$

$$\frac{\chi, \varphi, e \rightsquigarrow \chi', \alpha \quad \chi'(\alpha) = (C, \lambda s, \_) \quad \chi', \varphi, e' \rightsquigarrow \chi'', \beta \quad \chi'', (\mathsf{this} \mapsto \alpha, x \mapsto \beta, \mathcal{O}(C) \mapsto \lambda s), \mathcal{M}(C, m)\downarrow_3 \rightsquigarrow \chi''', \beta'}{\chi, \varphi, e.m(e') \rightsquigarrow \chi''', \beta'}\textsc{Call}$$

Fig. 14: Operational semantics of OHMM^{hl}.

The intuition behind these rules is quite standard and similar semantics can be found in the literature [6]. Note however the rule for pool creation, POOL, which stores a new pool location, π, in the heap and assigns it to the pool identifier p in the stack frame, so that it may be used by e.

A.3 Static semantics

OHMM^{hl}'s typing rules are as in Fig. 15. Note that these rules are a simplification of those presented in Fig. 12 where we dropped all the compilation steps.

$$\frac{}{\Gamma \vdash x \triangleright \Gamma(x)}\ [\textsc{Var}] \qquad \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{null} \triangleright T}\ [\textsc{Null}] \qquad \frac{\Gamma \vdash e \triangleright T \quad \Gamma[x \mapsto T] \vdash e' \triangleright T'}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \triangleright T'}\ [\textsc{Let}]$$

$$\frac{\Gamma[p \mapsto C] \vdash e \triangleright T}{\Gamma \vdash \mathsf{pool}\ p\ \mathsf{of}\ C\ \mathsf{in}\ e \triangleright T}\ [\textsc{Pool}] \qquad \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{new}\ T \triangleright T}\ [\textsc{New}]$$

$$\frac{\Gamma \vdash e \triangleright T \quad \mathcal{F}(T, f) = T'}{\Gamma \vdash e.f \triangleright T'}\ [\textsc{FRead}] \qquad \frac{\Gamma \vdash e \triangleright T \quad \Gamma \vdash e' \triangleright T' \quad \mathcal{F}(T, f) = T'}{\Gamma \vdash e.f = e' \triangleright T'}\ [\textsc{FWrite}]$$

$$\frac{\Gamma \vdash e \triangleright T \quad \Gamma \vdash e' \triangleright T' \quad \mathcal{M}_T(T, m) = (T'', T', \_)}{\Gamma \vdash e.m(e') \triangleright T''}\ [\textsc{Call}]$$

Fig. 15: Expression type checking.

A.4 Well-formedness

Definition 1 (Well-formed high-level programs).

$$\begin{aligned}
&\textbf{prog.}\ \vdash P \ \Leftrightarrow\ P = cd_1, ..., cd_n \ \wedge\ \forall i \in \{1..n\}.\exists \Gamma_i.P, \Gamma_i \vdash cd \\
&\textbf{class.}\ P, \Gamma \vdash \mathsf{class}\ C\langle os\rangle\{_\} \ \Leftrightarrow\ \Gamma \vdash C\langle os\rangle \ \wedge\ (\forall f.\ \mathcal{F}(C, f) = T \implies \Gamma \vdash T) \wedge \\
&\qquad (\forall m.\ \mathcal{M}(C, m) = (T, T', e) \implies \Gamma \vdash T \ \wedge\ \Gamma \vdash T' \wedge \\
&\qquad\qquad \Gamma[\mathsf{this} \mapsto C\langle os\rangle, x \mapsto T'] \vdash e \triangleright T)
\end{aligned}$$

Definition 2 (Well-formed high-level configurations).

$$\begin{aligned}
&\textbf{wfConfig.}\ \Gamma \vDash \chi, \varphi \Leftrightarrow \vDash \chi \ \wedge\ \Gamma, \chi \vDash \varphi \\
&\textbf{wfHeap.}\ \vDash \chi \Leftrightarrow\ \forall \alpha \in \mathrm{dom}(\chi).\ \chi(\alpha) = (C, \lambda s, _) \implies \chi \vDash \alpha \triangleleft C\langle \lambda s\rangle \\
&\textbf{wfFrame.}\ \Gamma, \chi \vDash \varphi \Leftrightarrow\ \forall x \in \mathrm{dom}(\varphi).\chi \vDash \varphi(x) \triangleleft \Gamma(x) \\
&\textbf{wfValue.}\ \chi \vDash \beta \triangleleft C\langle ls\rangle \Leftrightarrow\ \chi \vDash \beta : C \\
&\qquad\qquad \wedge\ (\beta = \alpha \implies \forall f.(\mathcal{F}o(C, f, ls) = T' \implies \chi \vDash \chi(\alpha, f) : T')) \\
&\qquad \chi \vDash \beta : C \Leftrightarrow \beta = \mathsf{null} \ \vee\ (\beta = \alpha \ \wedge\ \chi(\alpha)\downarrow_1 = C)
\end{aligned}$$

B Program translation

In order to translate high-level programs to low-level programs we use the following definitions:

$$\mathcal{C} : \mathbb{N} \to \mathit{Boolean}^\star$$
$$\mathcal{C}(n) \equiv \{bs \mid \#bs = n \ \wedge \ \forall i \in 1..n.(bs[i-1] \in \{\mathsf{false}, \mathsf{true}\})\}$$

$$\mathcal{L} : \mathit{Boolean}^\star \to \mathit{Location}^\star$$
$$\mathcal{L}(b_1, ..., b_n) \equiv (l_1, ..., l_n) \Leftrightarrow \forall i \in 1..n.(b_i \ \wedge \ \exists p.l_i = p) \ \vee \ (\neg b_i \ \wedge \ l_i = \mathsf{none})$$

$$\mathcal{T} : \mathit{ClassId} \times \mathit{MethodId} \times \mathit{Boolean}^\star \to \mathit{FunctionId}$$
$$\mathcal{T}(C, m, b_1, ..., b_n) \equiv Cm\_l_1...l_n \Leftrightarrow (l_1, ..., l_n) = \mathcal{L}(b_1, ..., b_n)$$

Definition 3 (Translation definition).

$$\begin{aligned}[\![P]\!] =_\Sigma P^{ll} \Leftrightarrow \Sigma = (\!|P|\!) \\ \wedge \ P = cd_1, ..., cd_n \\ \wedge \ \forall i \in 1..n.\exists \Gamma.\ \Gamma_i \vdash cd_i \ \wedge \ P^{ll} = [\![cd_1 \mid \Gamma_1]\!], ..., [\![cd_n \mid \Gamma_n]\!]\end{aligned}$$

1st step:

$$\begin{aligned}&[\![\mathsf{class}\ C\langle os\rangle\ \_\ mds \mid \Gamma]\!] \equiv fnds \Leftrightarrow \\ &\quad \forall bs \in \mathcal{C}(|os|).\forall md \in mds.\exists fnd \in fnds. \\ &\qquad \mathcal{L}(bs) = ls \ \wedge \ |bs| = |os| \ \wedge \ \mathcal{T}(C, m, ls) = fn \ \wedge \ ls - \mathsf{none} = (p_1, ..., p_k) \ \wedge \\ &\qquad md[ls/os] = \mathsf{def}\ m(x\colon T)\colon T'\{e\} \ \wedge \ fnd = \mathsf{fun}fn(r_{this}, r_x, r_1, ..., r_k) \Rightarrow ss \ \wedge \\ &\qquad \Delta = (\mathsf{this} \mapsto r_{this}, x \mapsto r_x, p_1 \mapsto r_1, ..., p_k \mapsto r_k) \ \wedge \\ &\qquad \Gamma(\mathsf{this}) = C\langle ls\rangle \ \wedge \ \Gamma(x) = T \ \wedge \ \Delta, \Gamma \vdash e \triangleright \_, \langle ss\rangle\end{aligned}$$

2nd step:

$$\begin{aligned}&(\!|P|\!) \equiv \Sigma \Leftrightarrow \ \forall cd \in P. \\ &\qquad cd = \mathsf{class}\ C\langle\_\rangle\{cl_1, ..., cl_k; \_\} \ \wedge \ \mathcal{F}s(C) = \{f_1\colon C_1\langle\_\rangle, ..., f_n\colon C_n\langle\_\rangle\} \ \wedge \\ &\qquad \Sigma(C) = ((f_1, ..., f_n), ps) \ \wedge \ ps = (cs_1, ..., cs_k) \ \wedge \\ &\qquad \forall i \in 1..k.\ (cl_i = \mathsf{cluster}\ \{f_1\colon C_1'\langle\_\rangle, ..., f_j\colon C_j'\langle\_\rangle\} \implies cs_i = (C_1', ..., C_j'))\end{aligned}$$

C Configuration and Value Equivalence

In this section we define equivalence between high-level and low-level configurations. Note in particular the definition of equivalent heaps of Definition 4. It checks that a heap layout as in Fig. 5 is equivalent with the heap layouts of Fig. 7 and Fig. 8. Note the auxiliary functions $h(\ell)$ and $h(\wp, n)$, which are defined in Appendix F, and that return the object allocated in the given locations. If the object is allocated in the pool, then its different records should be reassembled.

Definition 4 (Equivalence between High and Low level configurations). *Using the bijection*

$B : (\mathit{Address} \to (\mathit{ObjAddr} \cup \mathit{Reference})) \times (\mathit{PoolLoc} \to \mathit{PoolAddr})$

we define equivalence between high-level (χ, φ) and low level (h, fr) configurations, as $\chi, \varphi \simeq_{B\Delta} h, fr$. Configurations are equivalent if their heaps are equivalent to each other, as well as their stack frames.

Equivalent configurations: $\chi, \varphi \simeq_{B\Delta} h, fr \Leftrightarrow \chi \simeq_B h \ \wedge \ \varphi \simeq_{B\Delta} fr$

Equivalent heaps: $\chi \simeq_B h \Leftrightarrow$

$$\begin{aligned}
\forall \alpha \in \chi.\ [\chi(\alpha) = (C, \lambda_, f_1 \mapsto \beta_1, ..., f_n \mapsto \beta_n) \implies \\
(\lambda = \mathsf{none} \implies (B(\alpha) \in \mathit{ObjAddr}) \wedge \\
(\lambda \neq \mathsf{none} \implies (B(\alpha) \in \mathit{Reference}) \wedge \\
h(B(\alpha)) = (C, f_1 \mapsto v_1, ..., f_n \mapsto v_n) \ \wedge \ \forall i \in 1..n. \beta_i \simeq_B v_i)] \wedge \\
\forall \pi \in \chi.\ [\exists \wp. B(\pi) = \wp \ \wedge \ h(\wp) = (C, clt_1, ..., clt_n) \implies \\
\Sigma(C)\downarrow_2 = (cs_1, ..., cs_1) \ \wedge \ \forall i \in 1..n. (h \vDash clt_i \triangleleft cs_i \ \wedge \ |clt_1| = |clt_i|)]
\end{aligned}$$

Equivalent frames: $\varphi \simeq_{B\Delta} fr \Leftrightarrow$

$$\begin{aligned}
[\forall x \in \mathrm{dom}(\varphi). \exists r. \Delta(x) = r \ \wedge \ \varphi(x) \simeq_B fr(r)] \wedge \\
[\forall p \in \mathrm{dom}(\varphi). \exists r'. \Delta(p) = r \ \wedge \ B(\varphi(x)) = fr(r')]
\end{aligned}$$

Equivalent values: $\beta \simeq_B v \Leftrightarrow \beta = \mathsf{null} = v \ \vee \ (\beta = \alpha \ \wedge \ B(\alpha) = v)$

Cluster agreement:

$$\begin{aligned}
h \vDash (rec_1, ..., rec_n) \triangleleft Cs \ \Leftrightarrow \forall i \in 1..n. \ h \vDash rec_i \triangleleft Cs \\
h \vDash (v_1, ..., v_n) \triangleleft C_1, ..., C_n \ \Leftrightarrow \forall i \in 1..n. \ h \vDash v_i \triangleleft C_i \\
h \vDash v \triangleleft C \ \Leftrightarrow v = \mathsf{null} \ \vee \ h(v)\downarrow_1 = C
\end{aligned}$$

D OHMM^{hl} Full Code for the Running Example

In this section we present the full OHMM^{hl} code for our example. The code is shown below, and the respective layout in memory is the one of Fig. 8.

```
class VideoList⟨o1, o2, o3⟩
  cluster {
    head: Node⟨o2, o3⟩
  }

  def popularVideos(above: int): void
    // No changes required
    let cur = this.head in
    while (cur != null) {
      let v = cur.video in
      if v.views > pivot then
        print(v.id, v.views, v.likes);
      cur = cur.next
    }

  def populate(data: String): void
    /** ... */

class Node⟨o1, o2⟩
  cluster {
    video: Video⟨o2⟩
    next: Node⟨o1, o2⟩
  }
```

```
class Video⟨o⟩
  cluster {
    id: int
    likes: int
  }
  cluster {
    views: int
  }

class Main
  // it has a single instance
  // no need for pool allocation
  def main(): void
    Pool
      p1 of Node
      p2 of Video in
    let
      data = readFile("videos.txt")
      videos = new VideoList⟨none, p1, p2⟩
    in {
      videos.populate(data);
      videos.popularVideos(50000);
    }
```

E OHMM^{ll} Full Code for the Running Example

```
fun main(r_this) ⇒
  r0 := pcreate(Node);
  r1 := pcreate(Video);
  r2 := "videos"
  r3 := readFile(r_this);
  r4 := alloc(VideoList);
  r5 := populate_n_p1_p2(r1, r3, r0, r1);
  r6 := 50000;
  r7 := popularVd_n_p1_p2(r4, r6, r0, r1);
  r := null; r // void
```

```
fun popularVd_n_n_n(r_this, r_x) ⇒
  [[Δ(this) = r_this ∧ Δ(above) = r_x]]
  [[Γ(this) = VideoList⟨none, none, none⟩]]
  r_0 := read(r_this, head);
  while(r_0 not null) {
    r_1 := read(r_0, video);
    r_2 := read(r_1, views);
    if r_2 > r_x then
      r_3 := read(r_1, id);
      r_4 := read(r_1, views);
      r_5 := read(r_1, likes);
      r_6 := print(r_3, r_4, r_5);
    r_0 = read(r_0, next)
  };
  r := null; r // void
```

```
fun popularVd_n_n_p(r_this, r_x, r) ⇒
  [[Δ(this) = r_this ∧ Δ(above) = r_x]]
  [[Γ(this) = VideoList⟨none, none, p_2⟩]]
  [[Δ(p) = r]]
  r_0 := read(r_this, head);
  while(r_0 not null) {
    r_1 := read(r_0, video);
    r_2 := pread(r_1, 1, 0);
    if r_2 > r_x then
      r_3 := pread(r_1, 0, 0);
      r_4 := pread(r_1, 1, 0);
      r_5 := pread(r_1, 0, 1);
      r_6 := print(r_3, r_4, r_5);
    r_0 = read(r_0, next)
  };
  r := null; r // void
```

```
fun popularVd_n_p1_p2(r_this, r_x, r_1, r_2) ⇒
  [[Δ(this) = r_this ∧ Δ(above) = r_x]]
  [[Γ(this) = VideoList⟨none, p_1, p_2⟩]]
  [[Δ(p_1) = r_1 ∧ Δ(p_2) = r_2]]
  r_0 := read(r_this, head);
  while(r_0 not null) {
    r_1 := read(r_0, video);
    r_2 := pread(r_1, 1, 0);
    if r_2 > r_x then
      r_3 := pread(r_1, 0, 0);
      r_4 := pread(r_1, 1, 0);
      r_5 := pread(r_1, 0, 1);
      r_6 := print(r_3, r_4, r_5);
    r_0 = read(r_0, next)
  };
  r := null; r // void
```

```
fun popularVd_n_p1_p2_p3(r_this, r_x, r_1, r_2, r_3) ⇒
  [[Δ(this) = r_this ∧ Δ(above) = r_x]]
  [[Γ(this) = VideoList⟨p_1, p_2, p_3⟩]]
  [[Δ(p_1)=r_1 ∧ Δ(p_2)=r_2 ∧ Δ(p_3)=r_3]]
  r_0 := pread(r_this, 0, 0);
  while(r_0 not null) {
    r_1 := pread(r_0, 0, 1);
    r_2 := pread(r_1, 1, 0);
    if r_2 > r_x then
      r_3 := pread(r_1, 0, 0);
      r_4 := pread(r_1, 1, 0);
      r_5 := pread(r_1, 0, 1);
      r_6 := print(r_3, r_4, r_5);
    r_0 = pread(r_0, next)
  };
  r := null; r // void
```

```
fun popularVd_n_p_n(r_this, r_x, r) ⇒
  [[Δ(this) = r_this ∧ Δ(above) = r_x]]
  [[Γ(this) = VideoList⟨none, p, none⟩]]
  [[Δ(p) = r]]
  ...
```

```
fun popularVd_p_n_n(r_this, r_x, r) ⇒
  [[Δ(this) = r_this ∧ Δ(above) = r_x]]
  [[Γ(this) = VideoList⟨p, none, none⟩]]
  [[Δ(p) = r]]
  ...
```

```
fun popularVd_p1_p2_n(r_this, r_x, r_1, r_2) ⇒
  [[Δ(this) = r_this ∧ Δ(above) = r_x]]
  [[Γ(this) = VideoList⟨p_1, p_2, none⟩]]
  [[Δ(p_1) = r_1 ∧ Δ(p_2) = r_2]]
  ...
```

```
fun popularVd_p1_n_p2(r_this, r_x, r_1, r_2) ⇒
  [[Δ(this) = r_this ∧ Δ(above) = r_x]]
  [[Γ(this) = VideoList⟨p_1, none, p_3⟩]]
  [[Δ(p_1)=r_1 ∧ Δ(p_2)=r_2]]
  ...
```

F Auxiliary Definitions

F.1 OHMM^{hl} auxiliary definitions.

The definitions below assume a high-level program P globally accessible.

Class owners:

$\mathcal{O}(C) \equiv \{o_1, ..., o_n\} \Leftrightarrow \text{classOf}(P, C) = \mathsf{class}\ C\langle o_1, ..., o_n\rangle\{\_\_\}$

Where is the field

$\mathcal{W}(C, f) \equiv (n, n') \Leftrightarrow \text{classOf}(P, C) = \mathsf{class}\ C\langle\_\rangle\ \{cls\ \_\}\ \wedge\ cls[n, n'] = (f\colon \_)$

Field lookup:

$\mathcal{F}(C, f) \equiv T \Leftrightarrow \text{classOf}(P, C) = \mathsf{class}\ C\_\{cls\_\}\ \wedge\ \exists i.\ (f\colon T) \in cls[i]$

Field lookup 2:

$$\mathcal{F}(C\langle ls\rangle, f) \equiv \begin{cases} \mathcal{F}(C, f)[ls/\mathcal{O}(C)] & \text{if } ls[0] = \mathsf{none} \\ (\mathcal{F}(C, f)[ls/\mathcal{O}(C)], i, j) & \text{if } ls[0] \neq \mathsf{none}\ \wedge\ \mathcal{W}(C, f) = (i, j) \end{cases}$$

All fields:

$\mathcal{F}s(C) \equiv \{(f\colon T) | \mathcal{F}(C, f) \neq \bot\}$

Method lookup:

$\mathcal{M}(C, m) \equiv (T, T', e) \Leftrightarrow \text{classOf}(P, C) = \mathsf{class}\ C\_\{\_mds\}$
$\qquad \wedge\ (\mathsf{def}\ m(x : T') : T\{e\}) \in mds$

Method lookup 2:

$\mathcal{M}_T(C\langle ls\rangle, m) \equiv (T, T', e, fn) \Leftrightarrow \mathcal{M}(C, m)[ls/\mathcal{O}(C)] = (T, T', e)$
$\qquad \wedge\ \mathcal{T}(C, m, bools(ls)) = fn$

Field read:

$\chi(\alpha, f) \equiv \chi{\downarrow_1}(\alpha){\downarrow_3}(f)$

Field write:

$\chi[(\alpha, f) \mapsto \beta] \equiv \chi' \Leftrightarrow \chi{\downarrow_1} \setminus \{\alpha\} = \chi'{\downarrow_1} \setminus \{\alpha\}\ \wedge\ \chi'{\downarrow_1}(\alpha) = \chi'{\downarrow_1}(\alpha)[f \mapsto \beta]\ \wedge\ \chi{\downarrow_2} = \chi'{\downarrow_2}$

$\omega[f \mapsto \beta] \equiv \omega' \Leftrightarrow \omega = (C, \lambda s, (f \mapsto \_, f_1 \mapsto \beta_1, ..., f_n \mapsto \beta_n))$
$\qquad \wedge\ \omega' = (C, \lambda s, (f \mapsto \beta, f_1 \mapsto \beta_1, ..., f_n \mapsto \beta_n))$

Add pool location:

$\chi \oplus \pi \equiv \chi' \Leftrightarrow \chi'{\downarrow_1} = \chi{\downarrow_1}\ \wedge \chi'{\downarrow_2} = \chi{\downarrow_2} \uplus \{\pi\}$

Object init:

$\text{init}(C\langle\lambda s\rangle) \equiv (C, \lambda s, f_1 \mapsto \mathsf{null}, ..., f_n \mapsto \mathsf{null}) \Leftrightarrow \mathcal{F}s(C) = \{f_1, ..., f_n\}$

Locations in the frame:

$\varphi(l_1, ..., l_n) \equiv (\lambda_1, ..., \lambda_n) \Leftrightarrow \forall i \in 1..n.\ (l_i = \mathsf{none} = \lambda_i)\ \vee\ (l_i \neq \mathsf{none}\ \wedge\ \lambda_i = \varphi(l_i))$

Get registers:

$regs_{\Delta}(C\langle ls\rangle) \equiv (r_1, ..., r_n) \Leftrightarrow \forall i \in 1..n. \exists l_i \in ls.\ \Delta(l_i) = r_i$

F.2 OHMM^{ll} auxiliary definitions.

Function lookup

$\mathcal{F}\mathsf{un}(fn) \equiv (ss, rs) \Leftrightarrow (\mathsf{fun}fn(rs) \Rightarrow ss) \in P^{ll}$

Get object

$h(\ell) \equiv h{\downarrow_1}(\ell)$

Get object from pool

$h(\wp, n) \equiv (C, f_1 \mapsto v_1, ..., f_n \mapsto v_n) \Leftrightarrow$
$\quad h{\downarrow_2}(\wp) = (C, clts) \ \wedge \ \forall i \in 1..n.(\mathcal{W}(C, f_i) = (j, k) \ \wedge \ clts[j][n][k] = v_i)$

Pool read

$clts[i, j, k] \equiv clts[i][j][k]$

Pool write

$clts[(i, j, k) \mapsto v] \equiv clts' \Leftrightarrow$
$\quad \forall i', j', k'.((i, j, k) \neq (i', j', k') \implies clts[i', j', k'] = clts'[i', j', k']) \ \wedge \ clts[i, j, k] = v$

Object read

$h(\ell, f) \equiv h{\downarrow_1}(\ell){\downarrow_2}(f)$

Object write

$h[(\ell, f) \mapsto v] \equiv h' \Leftrightarrow h(\ell) = (C, (f \mapsto \_, f_1 \mapsto v_1, ..., f_n \mapsto v_n))$
$\quad \wedge \ h'(\ell) = (C, (f \mapsto v, f_1 \mapsto v_1, ..., f_n \mapsto v_n))$
$\quad \wedge \ h \setminus \{\ell\} = h' \setminus \{\ell\}$

Object init

$\text{initObj}(C) \equiv (C, (f_1 \mapsto \mathsf{null}, ..., f_n \mapsto \mathsf{null}))$
$\quad \Leftrightarrow \Sigma(C){\downarrow_1}= (f_1, ..., f_n)$

Record init

$\text{initRec}(C, k) \equiv (\mathsf{null}_1, ..., \mathsf{null}_n) \Leftrightarrow \Sigma(C){\downarrow_2}= (cs_1, ..., cs_m) \ \wedge \ cs_k = (C_1, ..., C_n)$

G Auxiliary Lemmas

G.1 Auxiliary lemmas

The proofs of Theorems 1, 2 and 3 use the following auxiliary lemmas.

Lemma 1.

$$\forall h, fr, as, r.\ \textbf{If}\ h, fr, as; r \rightsquigarrow h', v\ \textbf{then}\ \exists fr'.\ h', fr', r \rightsquigarrow h', v\ \wedge\ fr'(r) = v$$

Lemma 2.

$$\forall h_0, fr_0, as_0, r_0, h_1, fr_1, as_1, r_1.\ \textbf{If}\ h_0, fr_0, as_0; r_0 \rightsquigarrow h_1, v_1\ \wedge\ h_1, fr_1, as_1; r_1 \rightsquigarrow h_1, v_2\ \textbf{then}\ h_0, fr_0, as_0; as_1; r_1 \rightsquigarrow h_2, v_2$$

Lemma 3.

$$\forall h, fr, as, r, r', f.\ \textbf{If}\ h, fr, as; r' \coloneqq \mathsf{read}(r, f); r' \rightsquigarrow h, v\ \textbf{then}\ \exists \ell \in h.\ h, fr, as; r \rightsquigarrow h, \ell$$

Lemma 4.

$$\forall h, fr, as, r, r', i, j.\ \textbf{If}\ h, fr, as; r' \coloneqq \mathsf{pread}(r, i, j); r' \rightsquigarrow h, v\ \textbf{then}\ \exists n. \exists_{\wp} \in h.\ h, fr, as; r \rightsquigarrow h, (\wp, n)$$

Lemma 5.

$$\forall h, fr, as, r, r', f.\ \textbf{If}\ h, fr, as; r' \coloneqq \mathsf{write}(r, f, \_); r' \rightsquigarrow h, v\ \textbf{then}\ \exists \ell \in h.\ h, fr, as; r \rightsquigarrow h', \ell$$

Lemma 6.

$$\forall h, fr, as, r, r', i, j.\ \textbf{If}\ h, fr, as; r' \coloneqq \mathsf{pwrite}(r, i, j, \_); r' \rightsquigarrow h, v\ \textbf{then}\ \exists n. \exists_{\wp} \in h.\ h, fr, as; r \rightsquigarrow h', (\wp, n)$$

Lemma 7 (Generation lemma).

If $\Delta, \Gamma \vdash e \mapsto T, ss, \Delta'$

then

1. $e = \mathsf{this} \implies T = \Gamma(\mathsf{this}) \ \wedge \ ss = \Delta(\mathsf{this})$
2. $e = x \implies T = \Gamma(x) \ \wedge \ ss = \Delta(x)$
3. $e = \mathsf{null} \implies \exists r. r \notin \Delta \ \wedge \ ss = r \coloneqq ; r$
4. $e = \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \implies$
   $\exists as, r, T', ss'.\ \Delta, \Gamma \vdash e_0 \mapsto T', as; r, \Delta' \wedge$
   $\Delta'[x \mapsto r], \Gamma[x \mapsto T'] \vdash e_1 \mapsto T, ss', \_ \ \wedge \ ss = as; ss'$
5. $e = \mathsf{pool}\ p\ \mathsf{of}\ C\ \mathsf{in}\ e_0 \implies$
   $\exists r, ss'.\ r \notin \Delta \ \wedge \ \Delta[p \mapsto r], \Gamma[p \mapsto C] \vdash e_0 \mapsto T, ss', \_ \wedge$
   $ss = r \coloneqq \mathsf{pcreate}(C); ss'$
6. $e = \mathsf{new}\ C\langle \mathsf{none}, ls \rangle \implies$
   $T = C\langle \mathsf{none}, ls \rangle \ \wedge \ \exists r. r \notin \Delta \ \wedge \ ss = r \coloneqq \mathsf{alloc}(C); r$
7. $e = \mathsf{new}\ C\langle p, ls \rangle \implies$
   $T = C\langle p, ls \rangle \ \wedge \ \exists r, r'. r' \notin \Delta \ \wedge \ \Delta(p) = r \ \wedge \ ss = r' \coloneqq \mathsf{palloc}(r); r'$
8. $e = e_0.f \implies$
   $\exists T', \Delta', as, r, r', rhs.$
   $\Delta, \Gamma \vdash e_0 \mapsto T', as; r, \Delta' \ \wedge \ r' \notin \Delta' \ \wedge \ ss = as; r' \coloneqq rhs; r' \ \wedge$
   $(\mathcal{F}(T', f) = (T, i, j) \ \wedge \ rhs = \mathsf{pread}(r, i, j)) \vee$
   $(\mathcal{F}(T', f) = T \ \wedge \ rhs = \mathsf{read}(r, f))$
9. $e = e_0.f = e_1 \implies$
   $\exists T', \Delta', \Delta'', as, as', r, r', r'', rhs.$
   $\Delta, \Gamma \vdash e_0 \mapsto T', as; r, \Delta' \ \wedge \ \Delta', \Gamma \vdash e_1 \mapsto T, as'; r', \Delta'' \wedge$
   $r'' \notin \Delta'' \ \wedge \ ss = as; as'; r'' \coloneqq rhs; r' \ \wedge$
   $(\mathcal{F}(T', f) = (T, i, j) \ \wedge \ rhs = \mathsf{pwrite}(r, i, j, r')) \vee$
   $(\mathcal{F}(T', f) = T \ \wedge \ rhs = \mathsf{write}(r, f, r'))$
10. $e = e_0.m(e_1) \implies$
    $\exists T', T'', \Delta', \Delta'', as, as', r, r', r'', rs.$
    $\Delta, \Gamma \vdash e_0 \mapsto T', as; r, \Delta' \ \wedge \ \Delta', \Gamma \vdash e_1 \mapsto T'', as'; r', \Delta'' \wedge$
    $r'' \notin \Delta'' \ \wedge \ ss = as; as'; r'' \coloneqq fn(r, r', rs); r'' \ \wedge$
    $\mathcal{M}_T(T', m) = (T, T'', \_, fn) \ \wedge \ rs = reg_{\Delta''}(T')$