Floorplan: Spatial Layout in Memory Management Systems

Karl Cronburg
Tufts University
karl@cs.tufts.edu

Samuel Z. Guyer
Tufts University
sguyer@cs.tufts.edu

Abstract

In modern runtime systems, memory layout calculations are hand-coded in systems languages. Primitives in these languages are not powerful enough to describe a rich set of layouts, leading to reliance on ad-hoc macros, numerous interrelated static constants, and other boilerplate code. Memory management policies must also carefully orchestrate their application of address calculations in order to modify memory cooperatively, a task ill-suited to low-level systems languages at hand which lack proper safety mechanisms.

In this paper we introduce Floorplan, a declarative language for specifying high level memory layouts. Constraints formerly implemented by describing how to compute locations are, in Floorplan, defined declaratively using explicit layout constructs. The challenge here was to discover constructs capable of sufficiently enabling the automatic generation of address calculations. Floorplan is implemented as a compiler for generating a Rust library. In a case study of an existing implementation of the immix garbage collection algorithm, Floorplan eliminates 55 out of the 63 unsafe lines of code: 100% of unsafe lines pertaining to memory safety.

Keywords Memory Management, Runtime Systems

1 Introduction

The design of a memory manager is often hidden away in the runtime system and rarely discussed the way more prominent language features, such as syntax and semantics, are. A number of factors contribute to this state of affairs. First, each implementation of a managed language typically has its own memory manager, built from scratch, resulting in an almost total absence of shared code. Second, runtime system code is difficult to comprehensively understand: low-level and intricate, with a premium placed on performance. Finally, crucial design elements are often buried in the code, such as in simple yet pervasive pointer arithmetic and bitwise manipulations. These operations have ramifications on design elements across the entire system. As a result, these design elements are intrinsically hard to get correct the first time, and hard to diagnose when they are incorrect. Without a specification of these design elements, properties of a memory management algorithm are difficult or impossible to check and reason about formally. Documentation, when present, is in the form of informal and often inaccurate or ambiguous comments. Traditional memory safety tools [17] fall short because they typically assume that the memory allocator is allocating memory correctly in the first place.

In this work we take a first step toward remedying this situation: we present a declarative, domain-specific language (DSL), called *Floorplan*, for describing the structure of a heap as laid out by a memory manager. Floorplan is inspired by PADS [10], a language for describing ad hoc data file formats. A Floorplan specification looks like a grammar, augmented with memory management specific features. Floorplan provides powerful ways to specify the sizes, alignments, and relationships among chunks of memory, resulting in very compact descriptions. The key idea is that any correct state of the heap can be represented as a string (a sequence of bytes or tokens) derivable from a Floorplan grammar. Grammars are a natural choice because they match the configuration of most modern memory managers, which comprise layers of code that carve up memory into smaller and smaller pieces. Every [1], [2], [5], [6], [8], [11], [12], [13], [15], [16], memory manager we've studied exhibits this allocation scheme.

Note that Floorplan does not attempt to capture the policy details of any particular memory management algorithm. The closest Floorplan gets to capturing policy details is in its ability to logically connect multiple pieces of memory, e.g.

a bit map representing allocated cells in a block. The Floorplan compiler generates the low-level mechanisms — pointer calculations, bit masking, etc. — that the developer calls in order to implement some memory management policy. For example, the Floorplan compiler automates the synthesis of constants and pointer calculations for accessing an object liveness bitmap while saying nothing about how liveness or reachability are computed. Ongoing future work aims to leverage Floorplan specifications to debug algorithmic errors resulting in memory corruption. Such temporal errors are not easily detectable with frameworks like Valgrind [17] and PIN [14]. These frameworks can be leveraged more methodically with a layout specification language like Floorplan.

1.1 Contributions

To summarize, this work makes the following contributions:

- A declarative specification language based in part on a novel formalization of union types in Section 5. Floorplan allows users to express a memory layout as a *specification*, defining the spatial relationships among one or more system-defined types of memory.
- The Floorplan specification of the layout of a state-of-the-art garbage collection algorithm: immix as implemented in Rust.
- Formal rules for translating surface syntax to a core expression language, and a denotational semantics for how toreduce a memory layout to a set of trees with bytes at their leaves.
- A Floorplan compiler targeting Rust.
- Boilerplate reduction and memory safety results from integrating a Floorplan specification with the Rust implementation of immix [13].

2 Motivation

Spatial layout is fundamental to the problem of dynamic memory management. Memory managers employ a variety of layout schemes to carve up raw memory, and each scheme is influenced by the particular algorithm being implemented. Great care goes into designing a layout which permits highly efficient operation of crucial layout operations. For example, a generational garbage collector might be laid out such that the nursery is in a lower part of memory than the older space. This choice allows the write barrier to be implemented exclusively with address comparisons. Similarly, a free-list allocator might divide pages into cells of equal size, like an array, with a bit map of free cells at the start of each page. This design allows the meta-data to be found by simply masking off the low bits of any cell address; the corresponding bit can then be computed easily by dividing the low bits of the address by the cell size. These optimizations improve performance, but are only valid if the layout permits them.

Software maintenance. In all the memory managers we've studied, spatial layout is only formally expressed by the code

```
int SCALAR_HEADER_SIZE =
  JAVA_HEADER_BYTES + OTHER_HEADER_BYTES;
int ARRAY_HEADER_SIZE =
  SCALAR_HEADER_SIZE + ARRAY_LENGTH_BYTES;
/** offset of object reference
    from the lowest memory word */
Offset TIB_OFFSET = JAVA_HEADER_OFFSET;
Offset STATUS_OFFSET = TIB_OFFSET.plus(STATUS_BYTES);
Offset AVAILABLE_BITS_OFFSET =
  VM.LittleEndian ?
      STATUS_OFFSET
    : STATUS_OFFSET.plus(STATUS_BYTES - 1);
int HASH_CODE_SHIFT = 2;
Word HASH_CODE_MASK =
  Word.one()
    .lsh(10)
    .minus(Word.one())
    .lsh(HASH_CODE_SHIFT);
/** How many bits are allocated to a thin lock? */
int NUM_THIN_LOCK_BITS = ADDRESS_BASED_HASHING ? 22 : 20;
/** How many bits to shift to get the thin lock? */
int THIN_LOCK_SHIFT = ADDRESS_BASED_HASHING ? 10 : 12;
```

Figure 1. Code fragment from Jikes RVM [2] showing some of the Java header related constants.

that implements it. Figure 1 is a typical example, taken from MMTk [7], the memory management toolkit. Notice, in particular, the calculation of the hash code mask – clearly, great care is required to write, modify, and maintain such code. While MMTk is among the most meticulously engineered of memory mangers, this memory manager consists of boilerplate code in excess of $2,399$ lines of address arithmetic calulations as per the following bash command:

```
$ find MMTk/ rvm/ -name *.java -exec egrep \
-e "\.(one|lsh|plus|minus|rshl|and|EQ)\(" \
-e "\.(zero|isZero|diff|store|load)\(" "{}" \; \
| wc -l
2399
```

Static typing. A common problem in the memory management field occurs when a memory manager is implemented with generic pointer types exhibiting memory-related bugs. Often such generic pointers are distinguishable based on their value. Suspicious pointer values are manually detected based on intrinsic properties, such as alignment checks and assertions pertaining to relationships with other known in-memory structures, e.g. containing blocks and regions.

Complicating matters further, specialized pointer types need not even be distinguishable from one another by value dynamically. For example the Rust code in Figure 2 shows code implementing a block of memory cells. This code operates over memory with raw address types, and consists of numerous address calculations on generic pointer types. By this design, the address of the start of a block is the same as the address of the first cell in that block. Code that uses this class could call either method – it does not matter which. If the layout changed, though, for example by adding a bit

```
#[repr(C)]
#[derive(Copy, Clone, Eq, Hash)]
pub struct Address(usize);
pub struct Block { start : Address }
impl Block {
  pub fn start(&self) -> Address {
    self.start }
  pub fn first_cell(&self) -> Address {
    self.start }}
```

Figure 2. Abbreviated snippet of Rust code from an implementation of immix [13].

map, then the methods *would* be different. Code that calls the `block.start()` method expecting a pointer to a cell would now fail at runtime in confusing ways. These failures motivate the approach of generating specialized address types.

Dynamic typing. In a runtime system the configuration of the heap changes over time in highly mechanistic, and largely superficial, ways. For example, the heap's configuration changes when a piece of addressable memory in the heap changes type. This results in new offset and address calculations being allowed on that address. These calculations are used to implement various allocation schemes which *combine*, *carve up*, or *interchange* pieces of memory.

Some allocation schemes *combine* multiple operating system level pieces of memory into larger pieces of memory. For example, multiple contiguous pages can be combined to form a single block. This combination is typically implemented with a simple multiplication or bit-shifting operation.

Other allocation schemes *carve* a single piece of memory into multiple subcomponents. For example, a block may be carved up into cells, with a bitmap at the beginning of the block. Carving up of memory is typically implemented with a simple offset added to an address, and a subsequent bounds check address comparison to detect block overflow.

Finally some allocation schemes define two or more pointer types to be *interchangeable*. For example, a cell of memory is either allocated or free, with differing internal layouts. A free cell controlled by a doubly-linked list policy typically contains two pointers. Accesses to a free cell must therefore be implemented with an offset addition to the cell's base address. Such core layout operations are simple in isolation, yet the design choices describing their composition are complex.

Efficiency. Address calculations need to behave such that an amortized analysis of an allocation scheme yields a highly efficient implementation. Existing handwritten calculations exhibit this efficiency, so generating address calculations to semantically and stylistically match handwritten code makes sense. Precise control over the form of generated code ensures efficiency-motivated size, alignment, and padding

Code	Nonterminal	Explanation
`||sz||`	$\langle mag\rangle$	"has size sz"
`@(sz)`	$\langle align\rangle$	"sz address alignment"
`@|sz|@`	$\langle magAlign\rangle$	"same $\langle align\rangle$ and $\langle mag\rangle$"
`# Bar`	$\langle demarc\text{-}val\rangle$	"some number of **Bar**s"
`foo : Bar`	$\langle field\rangle$	"field foo contains a **Bar**"
`Bar, Baz`	$\langle seq\rangle$	"**Bar** followed by **Baz**"
`Foo -> Bar`	$\langle layer\rangle$	"**Foo** consists of **Bar**"
`Bar | Baz`	$\langle union\rangle$	"one of **Bar** or **Baz**"
`FOO | BAR`	$\langle enum\rangle$	"in state **FOO** or **BAR**"

Figure 3. Informal semantics of constructs and operators in Floorplan. **Bar** and **Baz** represent arbitrary $\langle demarc\text{-}val\rangle$ values, **FOO** and **BAR** represent state flags of an $\langle enum\rangle$, **Foo** represents an identifier, and sz is some $\langle size\text{-}arith\rangle$.

invariants hold. Generating code also forgoes the manual writing of numerous lines of stylistically similar code.

Existing memory managers lack precise and formal specifications of their memory layouts. Memory managers can benefit from support for various forms of analysis, debugging, and code generation which this work tackles. In this paper we take a generative approach: we describe a specification language, its translation to a core calculus, and a compiler for generating Rust code.

3 Language overview with examples

The most fundamental operation in memory management is to take an unstructured piece of memory and to give it structure through *demarcation*. Demarcation is the dividing up of a layer of memory into a partitioning of components. Multiple layers of memory form an allocation hierarchy.

In order to allocate a piece of memory, a memory manager tracks metadata distinguishing a free piece from the same allocated piece. The state of this piece of memory, free or allocated, determines its layout. Existing systems written in C model this behavior with unions. For example, the first word of a free-list based allocator's free piece might contain a pointer, while that same word of memory once allocated might contain an object header. In order to access this allocated object's payload, a memory manager calculates the payload's offset from the base of the containing piece of memory. The ordering of fields in this piece of memory, header and payload, define its layout. Existing systems often model the ordering of fields with offset constants. For example, a memory manager computes the location of a payload in terms of the size of its header. In this section, we introduce Floorplan with similarly motivated examples.

Grammar 1, below, through Grammar 4 specify the syntactic constructs of a Floorplan specification in EBNF form. For a quick reference guide on how to read a Floorplan specification, refer to Figure 3. The grammars below are inline figures, which we recommend inspecting in the order they are presented before reading the remainder of this section.

Grammar 1: Literal lexemes. Layers & fields are types, formals represent natural numbers, and flags for enums.

⟨*layer-id*⟩ ::= [A-Z][a-zA-Z_]*	⟨*literal*⟩ ::= ⟨*bin*⟩ | ⟨*int*⟩
⟨*field-id*⟩ ::= [a-z][a-zA-Z_]*	⟨*bin*⟩ ::= 0b[01]+
⟨*formal-id*⟩ ::= [a-z][a-zA-Z_]*	⟨*int*⟩ ::= [0-9]+
⟨*flag-id*⟩ ::= [A-Z][A-Z_]*	⟨*prim*⟩ ::= 'bits' | 'bytes' | 'words' | 'pages'

Grammar 2: Arithmetic language for memory sizes.

```
⟨lit-arith⟩     ::= ⟨literal⟩ | '(' ⟨lit-arith⟩ ')'
                 |  ⟨lit-arith⟩ ⟨lit-arith-op⟩ ⟨lit-arith⟩
⟨lit-arith-op⟩  ::= '+' | '−' | '∗' | '/' | '^'
⟨size-arith⟩    ::= ⟨lit-arith⟩? ⟨prim⟩ | '(' ⟨size-arith⟩ ')'
                 |  ⟨size-arith⟩ ⟨size-arith-op⟩ ⟨size-arith⟩
⟨size-arith-op⟩ ::= '+' | '−'
```

Grammar 3: Layers of memory with annotated magnitudes, alignments, simultaneous annotations (⟨*magAlign*⟩), scoped formal parameter declarations, and containment (⟨*contains*⟩) compiler annotation hints[1].

```
⟨layer-simple⟩ ::= ⟨layer-id⟩ ('<' ⟨formals⟩ '>')? (⟨mag⟩? ⟨align⟩?
                   | ⟨magAlign⟩?) ⟨contains⟩* '->' ⟨demarc-val⟩
⟨layer⟩        ::= ⟨layer-simple⟩ | '(' ⟨layer-simple⟩ ')'
⟨mag⟩          ::= '||' ⟨size-arith⟩ '||'
⟨align⟩        ::= '@' '(' ⟨size-arith⟩ ')'
⟨magAlign⟩     ::= '@|' ⟨size-arith⟩ '|@'
⟨formals⟩      ::= ⟨formal-id⟩ (',' ⟨formal-id⟩)* ','?
⟨contains⟩     ::= 'contains' '(' ⟨layer-id⟩ ')'
```

Grammar 4: Demarcatable atomic units of memory.

```
⟨demarc-val⟩ ::= ('#' | ⟨formal-id⟩)? (⟨enum⟩ | ⟨bits⟩ | ⟨union⟩
              |  ⟨seq⟩ | ⟨ptr⟩ | ⟨size-arith⟩ | ⟨macro⟩)
⟨seq⟩        ::= 'seq' '{' ⟨demarc⟩ (',' ⟨demarc⟩)* ','? '}'
⟨union⟩      ::= 'union' '{' ⟨demarc⟩ ('|' ⟨demarc⟩)* '|'? '}'
⟨demarc⟩     ::= ⟨field⟩ | ⟨layer⟩ | ⟨demarc-val⟩
⟨field⟩      ::= ⟨field-id⟩ ':' ⟨demarc-val⟩
⟨ptr⟩        ::= (⟨layer-id⟩ | ⟨field-id⟩) 'ptr'
⟨enum⟩       ::= 'enum' '{' ⟨flag-id⟩ ('|' ⟨flag-id⟩)* '|'? '}'
⟨bits⟩       ::= 'bits' '{' ⟨bits-exp⟩ (',' ⟨bits-exp⟩)* ','? '}'
⟨bits-exp⟩   ::= ⟨field-id⟩ ':' ⟨size-arith⟩
⟨macro⟩      ::= ⟨layer-id⟩ ('<' ⟨args⟩ '>')?
⟨arg⟩        ::= ⟨formal-id⟩ | ⟨literal⟩
⟨args⟩       ::= ⟨arg⟩ (',' ⟨arg⟩)* ','?
```

3.1 What is a Floorplan demarcation

In Grammar 4 we introduced the syntactic form for the notion of a demarcation. A *demarcation* is a partitioning[2] of a layer of a heap. A boundary position in memory defining the partition of two or more ⟨*layer*⟩ and ⟨*field*⟩ types may (and often does) coincide with another layer's boundary.

For instance in our block-containing-cells motivating example (Figure 2) the beginning boundary of a block coincides with the boundary of that block's first cell. We can encode this memory layout as follows:

```
Cell -> seq { Header  -> 1 words,
              Payload -> 7 words }
Block ||2^16 bytes|| -> # Cell
```
(F1)

This code declares a block of cells with total size 2^{16} bytes. The "#" operator indicates that the `Cell` declaration should be repeated as many times as necessary in order to exactly fill the total size. The `Cell` reference on the last line of F1 parses as a ⟨*macro*⟩ expression[3] which must reference a top-level ⟨*layer-id*⟩ declaration of the specification file (`.flp` filename extension). A ⟨*macro*⟩ expression is syntactically replaced with its corresponding declaration.

From the layout in F1 the compiler generates specialized address types for pointers to a `Cell`, `Header`, `Payload`, and `Block` respectively. For safety reasons, a memory manager must only be able to cast from a `Block` address to a `Cell` address and not to, say, a `Payload` address. Therefore the compiler generates (simplified here) Rust code identical in purpose to that of Figure 2:

Types & casts generated for Code F1

```
pub struct CellAddr(usize);
pub struct HeaderAddr(usize);
pub struct PayloadAddr(usize);
pub struct BlockAddr(usize);
impl BlockAddr {
  pub fn get_first_cell(&self) -> CellAddr {
    CellAddr::from_usize(self.as_usize()) } }
```
(R1)

While this code is implementable by hand, the complier systematically enforces which conversions are memory-safe. Memory-safety in Floorplan is heavily influenced by where

[1]These instruct the compiler to generate functions for converting to the containing ⟨*layer-id*⟩ and vice-versa when memory alignments permit.

[2]Including finitely many partitions of size zero.

[3]Macros are not formally specified: they are a pre-processing pass to the compiler. Recursive macros are forbidden.

coinciding boundaries occur. These occur wherever two ⟨*layer*⟩ or ⟨*field*⟩ declarations are nested inside of one another under one condition: the nested path traverses neither the tail of a ⟨*seq*⟩ nor ⟨*demarc-val*⟩ annotated with a repetition[4]. Under this condition, Floorplan semantics (Section 5) guides the compiler in generating safe address conversions. Statically unsafe conversions are disallowed by construction.

3.2 Implementing bit-fields and repetitions

A header word on an object in a memory manager typically relies on intricately implemented offset constants to function, like back in Figure 1. For example, we might want to modify the `Header` portion of Code F1 to support bit-level manipulation in a traditional mark-sweep garbage collector:

```
Header @|1 words|@ -> bits {
  MARK : 1 bits, REF : 7 bits,
  UNUSED  : (1 words - 1 bytes) }                    (F2)
```

First, Code F2 constrains the alignment of header words to start on a `@|1 words|@` boundary. In addition, the memory manager needs to be able to access (read and write) the contents of the `MARK` and `REF` bits in order to mark and record the location of pointers in the payload, respectively. To facilitate this requirement, the compiler generates, e.g., the following constants and accessors:

Offset constants generated from Code F2

```
struct HeaderAddr(usize);
impl HeaderAddr {
  pub const MARK_LOW_BIT : usize = 0;
  pub const MARK_NUM_BITS : usize = 1;
  pub const MARK_MASK : u8 = 0b00000001;
  pub const REF_LOW_BIT : usize = 1;                 (R2)
  pub const REF_NUM_BITS : usize = 7;
  pub const REF_MASK : u8 = 0b11111110;
  pub fn set_MARK_bit(&self, val: bool) {
    self.store::<u8>(val as u8) }
  pub fn get_MARK_bit(&self) -> bool) {
    (self.load::<u8>() as bool) } }
```

Furthermore, a memory manager must be able to allocate pointers in the payload and mark their location in the `REF` field.[5] For example, the layout can dictate that pointer fields in an application object comprise the first n words of the payload by replacing the `Payload` in F1 with:

```
Payload ||7 words|| -> seq {
  refs: # (Cell ptr), rem:  # (1 words) }            (F3)
```

Notice here that the two "#" operators act together to fill the necessary space (7 words) available to them. Code F3 denotes 8 distinct layouts: the number of permutations by which two natural numbers can sum to 7. These permutations include (0 pointers, 7 words), (1 pointer, 6 words), and so on until (7 pointers, 0 words). In order to allocate some number of pointers, the compiler needs to give us a way to (1) access the `refs` field of a `Payload`, (2) initialize a pointer to the `rem` field, and (3) allocate an additional cell pointer. Code R3 below exhibits these functions:

Allocation pattern generated from Code F3

```
impl PayloadAddr {
  pub fn cast_payload_to_refs(&self)
    -> RefsAddr { // #1
      RefsAddr(self.as_usize()) }
  pub fn init_rem_after_refs(p1: RefsAddr          (R3)
    , bytes: usize) -> RemAddr { // #2
      debug_assert!(bytes%BYTES_IN_POINTER==0);
      p1.plus::<RemAddr>(bytes) }
  pub fn bump_new_Cell_ptr(rhs: RemAddr)
    -> (CellAddr, RemAddr) { // #3
      (rhs.plus(0),rhs.plus(BYTES_IN_POINTER)}}
```

Take for granted that we have access to the `PayloadAddr` of some cell. Function #1 above accesses the `refs` field of our payload. From this address we can initialize, with #2, the remainder (`rem`) to start zero bytes after the start of the payload. With #3 we can then allocate a new pointer with our `RemAddr` returned by #2. To allocate more pointers we iterate as necessary over #3, because #3 returns an updated `RemAddr`. The compiler knows to generate this allocation pattern because two adjacent fields each contain a repetition.

3.3 Implementing union types

In contrast to Code F3, we might want a more permissive object field layout where pointer fields can appear in any order in the payload. For example:

```
Payload ||7 words|| ->
  # union { Cell ptr | (1 words) }                   (F4)
```

In F4, the "#" operator acts to fill precisely 7 words of memory. In doing so, this particular "#" operates equivalently to the POSIX Extended Regular Expression (ERE) limited repetition expression `(a|b){7}`. As with this regex, F4 denotes $2^7 = 128$ distinct layouts: the number of permutations (with repeats) of the elements of the union fitting into 7 words. If instead we made a typo and wrote `(10 words)` in place of `(1 words)`, the compiler reports to us a consistency warning: the `(10 words)` branch of the union in F4 is dead code which does not contribute to a valid payload.

3.4 Implementing lookup tables

A memory manager often relies on metadata in lookup tables and byte maps. To indicate the relationship between metadata and memory it describes, the same ⟨*formal-id*⟩, `cnt`, can logically link two or more pieces of memory:

[4] More on ⟨#⟩ and ⟨*formal-id*⟩ repetitions four paragraphs from here.
[5] How the runtime determines which `REF` bit marks which payload word is outside the scope of this work.

```
Cell<sz> -> sz (1 words)
SizeKls<sz, cnt> @|2^16 bytes|@ -> seq {                (F5)
  cells: cnt Cell<sz>, map: cnt (1 bytes) }
Kls16 -> SizeKls<16>
```

Code F5 implements a 16-word size-class block of memory, with a byte map at the end of each block. Note that macro expressions are curried, so only the first argument need be expanded on the last line above. The compiler generates functions capable of translating between a cell and its corresponding byte entry in the map. For example in order to update the byte entry of some cell, we can call the set function in Code R5 on that cell's address, along with the value we want the map to remember:

Mapping code generated from Code F5

```
pub struct Cell_16([usize; 16]);
pub struct Cell2Byte {
  pub fBase: CellAddr, // from
  pub tBase: ByteAddr, // to                          (R5)
  pub e: Kls16EndAddr }
impl Cell2Byte {
  pub fn set(&self, fA: CellAddr, val: u8) {
  debug_assert!(fA >= self.fStart);
  let idxV = (fA - self.fBase) >> 7;
  let loc = self.tBase.offset::<Cell_16>(idxV);
  debug_assert!(self.e > loc); loc.store(val); } }
```

Dozens more functions are generated alongside `set()` in Code R5. We struggled to define meaningful naming schemes for generated address-types. For example `Cell_16` above comes from the ⟨*macro*⟩ expressions on the third and fourth lines of Code F5. Code R5 also exemplifies generated debugging assertions. Again, while these assertions can be manually written, formally deriving the largely trivial ones such as these bounds checks is feasible.

4 Study: Immix in Rust

In this section, Figure 4 introduces the notion of a demarcation diagram and Figure 5 shows the Floorplan specification of immix in Rust. For a precise handling of Floorplan semantics, see Section 5. Throughout this section subscripts on words$_1$ indicate line numbers in Figure 5.

4.1 Immix specification

Figure 5 shows the Floorplan specification for the Rust implementation [13] of the immix garbage collection algorithm. The heap is represented as a Region$_1$ parametrized by three formal arguments: the number of blocks, lines, and number of words `wrds` in the region. Note that once `num_blocks` is fixed, the other two take on fixed values.[6] This constraint we have made is self-imposed, and not a part of Floorplan

[6]The default immix heap is half a gigabyte of memory: 8000 blocks, more than 2 million lines, and over 65 million words on a 64-bit machine.

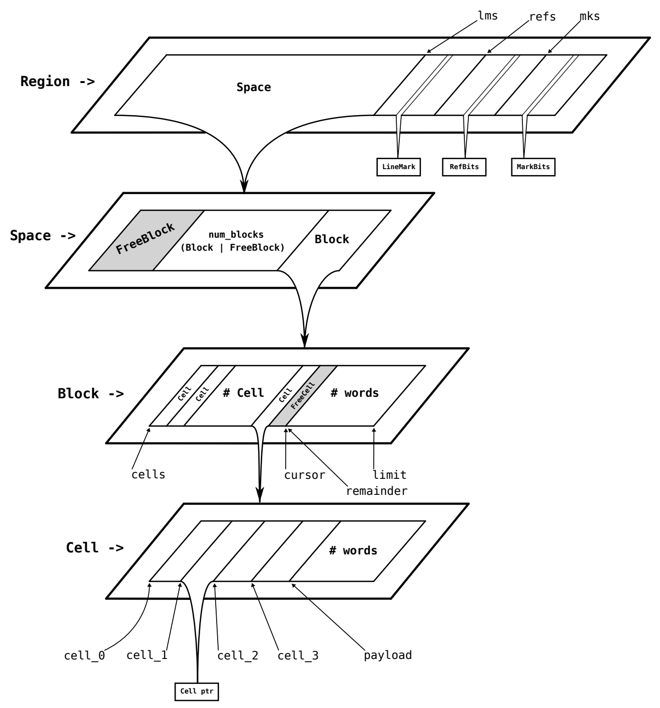


Figure 4. A four-layered demarcation diagram depicting the immix-rust layout. Each layer corresponds to a ⟨*layer*⟩ definition from Figure 5. Each layer is then further demarcated into ⟨*field*⟩ fields, ⟨*demarc-val*⟩ values, and other ⟨*layer*⟩ layers.

semantics. We debated including a version of the ⟨*union*⟩ operator which enforces size-equivalence of constituents, but decided against it for simplicity reasons: two flavors of the union operator arguably degrades comprehensibility.

A Region$_1$ layer consists of a single Space$_2$ followed by some metadata fields for marking lines$_{16}$, looking up reference bytes$_{17}$, and setting mark bits$_{18}$. RefBits$_{24}$ and MarkBits$_{30}$ both represent bit-fields which consume one byte of memory. Note that bit order for a RefBits$_{24}$ is defined such that the `OBJ_START`$_{26}$ bit occurs at a less significant bit than the `REF`$_{27}$ bits. `SHORT_ENCODE`$_{25}$ is the least significant (ones) bit.

Notice here$_7$ that a block of memory is annotated with the fact that it contains lines. The annotation indicates to the Floorplan compiler that it should generate code for converting between a Line$_{14}$ and its containing Block$_6$, and vice-versa. The conditions under which this code gets generated relies on the presence of known sizes and alignments for lines and blocks respectively.

In this [13] version of immix, objects do not have a header word. Instead each cell's corresponding RefBits$_{24}$ in the refs$_{17}$ array tracks which words$_{13}$ of memory in the Space$_2$ correspond to the start of an object, `OBJ_START`$_{26}$. The implementation of the immix algorithm determines how many heap references are in some cell by looking up the first 4 bits of the corresponding `REF`$_{27}$ field of that cell's RefBits$_{17}$.

```
1   Region<num_blocks, lines, wrds> -> seq {
2     Space @(2^19 bytes)@ -> union {
3         num_blocks union {
4             FreeBlock @(2^16 bytes)@ -> seq {
5                 2^16 bytes }
6           | Block ||2^16 bytes|| @(2^16 bytes)@
7                 contains(Line) -> seq {
8                 cells : # union {
9                     FreeCell @(1 words)@ -> # words
10                  | Cell },
11                remainder : # words,
12                limit     : 0 words } }
13      | wrds (1 words)
14      | lines Line @|2^8 bytes|@
15            contains(Cell) -> # bytes },
16    lms   : lines LineMark,
17    refs  : wrds RefBits,
18    mks   : wrds MarkBits }
19  Cell @(1 words)@ contains(Word) -> union {
20    seq { cell_0   : Cell ptr,
21          cell_1   : Cell ptr, cell_2 : Cell ptr,
22          cell_3   : Cell ptr, payload : # words }
23    | # words }
24  RefBits ||1 bytes|| -> bits {
25      SHORT_ENCODE  : 1 bits,
26      OBJ_START     : 1 bits,
27      REF           : 6 bits }
28  LineMark -> enum { Free | Live | FreshAlloc
29                   | ConservLive | PrevLive }
30  MarkBits ||1 bytes|| -> bits { MARK : 8 bits }
31  Stk -> seq { stack : # seq { Cell ptr },
32               lowWater : 0 words }
33  Registers -> seq { regs : # seq { Cell ptr },
34                     regsEnd : 0 words }
```

Figure 5. The Floorplan specification of immix as implemented in Rust [13].

Core value syntax

Address	α	$\in$	$\mathbb{N}$
Identifier	ℓ, f	$\in$	Strings
Values	v	$::=$	1 bytes | 0 bytes | T v_1 v_2 | N ℓ v
Type	τ	$::=$	$\{ v \}$

Figure 6. Syntactic forms of core Floorplan values. ℓ is for ⟨*layer-id*⟩ and ⟨*field-id*⟩, while f is for a ⟨*formal-id*⟩.

This implementation of immix extracts the application's root set directly from the stack and registers. The Stk$_{31}$ and Registers$_{33}$ are both assumed to be some number of Cell pointers$_{31,33}$ followed by a lowWater$_{32}$ mark and regsEnd$_{34}$ ending address respectively. The implementation performs conservative stack and register scanning.

5 Semantics

In this section we present the semantics of a Floorplan specification to be the set of heaps which satisfy the specification, with satisfaction as defined in Figure 9. Note that Floorplan semantics do not suffice to ingest raw pages.

5.1 Concrete value semantics

We represent an instance of a memory layout as a tree, as in Figure 6. Addresses are natural numbers representing locations in a flat addressable sequence of bytes. A value is a rooted binary tree with leaves each representing either zero or one byte. Trees may be interspersed with named "N" components, mapping directly back to named types in a Floorplan specification, as will become apparent by the semantics in the following Section 5.2. A finite set of trees represents a concrete type of memory.

An in-order traversal over a tree defines the order in which bytes at the leaves of the tree occur contiguously in memory. Finally, leaves(v) computes the number of 1-byte leaves in the tree as defined below, equivalent to the number of bytes the tree consumes in memory.

$$\begin{aligned}
\text{leaves}(1\text{ bytes}) &= 1 \\
\text{leaves}(0\text{ bytes}) &= 0 \\
\text{leaves}(T\ v_1\ v_2) &= \text{leaves}(v_1) + \text{leaves}(v_2) \\
\text{leaves}(N\ \ell\ v) &= \text{leaves}(v)
\end{aligned}$$

5.1.1 Example: the trees of a specification

Before introducing the core calculus, take the following Floorplan declaration:

```
K<n> |5 bytes| -> seq { hd : n (1 bytes),        (F6)
  tl : n seq { lft : 1 bytes, rgt : # bytes } }
```

This code represents the three distinct memory layouts as depicted in Figure 7, one for each feasible assignment of natural numbers to n and the "#". The $n = 0$ case is not feasible because that case consumes $0 \neq 5$ bytes. Similarly the $n = 3$ case is not feasible because the hd consumes 3 bytes and the `tl` consumes *at least* 3 bytes, one for each copy of `lft`, which sums to at least $6 \neq 5$ bytes. Formally, for constants $n, \#_i \in \mathbb{N}$, memory layout instances must satisfy the following constraint satisfaction [3] equation:

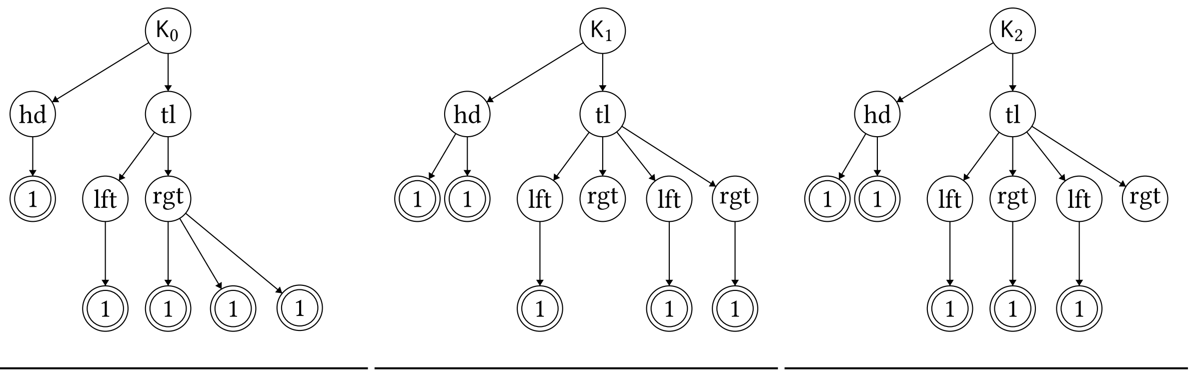
$$n + \sum_{i=0}^{n-1} n * (1 + \#_i) = 5 \qquad (5.1)$$

Equation 5.1 above was written by hand, and is not formally synthesized by the compiler. We will see in Section 5.2.1 how to reduce Code F6 to tree K$_0$ from Figure 7.

5.2 Abstract expression semantics[7]

Now we define the core expression language as in Figure 8. Each expression e denotes a memory layout. A memory layout has a corresponding (possibly empty) set of values v representing a type τ computable by the memory layout modeling function γ in Figure 9. A primitive expression (Prim n) denotes a contiguous (possibly empty$_7$) sequence$_9$ of n bytes. Similarly, a constrained expression (Con n e) denotes

[7]Subscripts$_1$ on$_2$ words$_3$correspond to lines in Figure 9



K_0: $n = 1$, $\#_0 = 3$ bytes K_1: $n = 2$, $\#_0 = 0$ bytes, $\#_1 = 1$ byte K_2: $n = 2$, $\#_0 = 1$ byte, $\#_1 = 0$ bytes

Figure 7. The three layouts for Code F6, with satisfying assignments to Equation 5.1.

Core calculus

Nats	n, m, c	$\in$	$\mathbb{N}$
Alignment	$\hat{a}$	$\in$	$\mathbb{N}^+$
Exp	e	$::=$	$\text{Prim } n \mid \text{Con } n\ e \mid e\ @\ \hat{a}$
		$\mid$	$e_1 + e_2 \mid e_1 \parallel e_2 \mid y :: \ e \mid \exists\ f\ .\ e$
		$\mid$	$f\ \#\ e$
Size	δ	$::=$	m
Environment	θ	$::=$	$\{f \mapsto n\}$
Config	χ	$::=$	(α, δ, e)

Figure 8. Core expression language representing a Floorplan specification. A function $\bar{\gamma}$ with type $\chi \longrightarrow \tau$ models the semantics of a memory layout.

a contiguous sequence of n bytes, but only for the (possibly non-existent$_{13}$) memory layout instances for which the substructure denoted by e fits precisely$_{12}$ into n bytes. An aligned expression ($e\ @\ \hat{a}$) denotes a memory layout for which the address of the first byte of memory of the layout must be a natural number multiple$_{15}$ of $\hat{a}$ bytes.

The remaining operators are the concatenation "+" and union "||" binary operators, as well as name binding with $y ::\ e$. A concatenation of two expressions denotes the contiguously laid out sequence$_{2–4}$ of those two expressions. A union of two expressions denotes a left-most aligned instance of either the first$_5$ or the second$_6$ expression. A named expression $y ::\ e$ binds$_{17,18}$ the name y to the expression e. An existentially quantified expression $\exists\ f\ .\ e$ brings the variable f into scope$_{20}$ in the subexpression e.

A variable on a repetition, the f in ($f\ \#\ e$), may be referenced multiple times. Each reference must also take on the same fixed value. This feature causes a Floorplan specification (i.e. grammar) to be non-regular: there exist Floorplan grammars which fail the Pumping Lemma.

Memory layout model $\bar{\gamma}$

1	$\gamma \llbracket (\alpha, m, \theta, e_1 + e_2) \rrbracket$	
2	$= \{ T\ r_1\ r_2 \mid r_1 \in \bigcup_{i=0}^{m} \gamma \llbracket (\alpha, i, \theta, e_1) \rrbracket$	
3	$, r_2 \in \gamma \llbracket (\alpha + \text{leaves}(r_1)$	
4	$, m - \text{leaves}(r_1), \theta, e_2) \rrbracket \}$	
5	$\gamma \llbracket (\alpha, m, \theta, e_1 \parallel e_2) \rrbracket$	$= \gamma \llbracket (\alpha, m, \theta, e_1) \rrbracket$
6		$\bigcup \gamma \llbracket (\alpha, m, \theta, e_2) \rrbracket$
7	$\gamma \llbracket (\alpha, 0, \theta, \text{Prim } 0) \rrbracket$	$= \{ 0 \text{ bytes} \}$
8	$\gamma \llbracket (\alpha, m, \theta, \text{Prim } n) \rrbracket$	
9	$\mid m \equiv n = \{ T\ (1 \text{ bytes})_1\ (\cdots\ \text{T}\ (1 \text{ bytes})_n\ (0 \text{ bytes})) \}$	
10	$\mid m \neq n = \emptyset$	
11	$\gamma \llbracket (\alpha, m, \theta, \text{Con } n\ e) \rrbracket$	
12	$\mid m \equiv n = \gamma \llbracket (\alpha, m, \theta, e) \rrbracket$	
13	$\mid m \neq n = \emptyset$	
14	$\gamma \llbracket (\alpha, m, \theta, \text{e}\ @\ \hat{a}) \rrbracket$	
15	$\mid \alpha \bmod \hat{a} \equiv 0 = \gamma \llbracket (\alpha, m, \theta, e) \rrbracket$	
16	$\mid \alpha \bmod \hat{a} \neq 0 = \emptyset$	
17	$\gamma \llbracket (\alpha, m, \theta, \ell ::\ e) \rrbracket$	$= \{ N\ \ell\ r$
18		$\mid r \in \gamma \llbracket (\alpha, m, \theta, e) \rrbracket \}$
19	$\gamma \llbracket (\alpha, m, \theta, \exists\ f\ .\ e) \rrbracket =$	
20	$\bigcup_{i=0}^{m} \gamma \llbracket (\alpha, m, \theta\{f \mapsto i\}, e) \rrbracket$	
21	$\gamma \llbracket (\alpha, m, \theta, f\ \#\ e) \rrbracket$	
22	$\mid f \notin \text{dom}(\theta) = \emptyset$	
23	$\mid m \equiv \theta(f) \equiv 0 = \{ T\ (0 \text{ bytes})\ (0 \text{ bytes}) \}$	
24	$\mid \theta(f) \equiv 0 = \emptyset$	
25	$\mid \theta(f) > 0$	
26	$= \{ T\ r_1\ r_2$	
27	$\mid r_1 \in \bigcup_{i=0}^{m} \gamma \llbracket (\alpha, i, \theta, e) \rrbracket$	
28	$, r_2 \in \gamma \llbracket (\alpha + \text{leaves}(r_1), m - \text{leaves}(r_1)$	
29	$, \theta\{f \mapsto (\theta(f) - 1)\}, f\ \#\ e) \rrbracket$	
30	$, m \equiv \text{leaves}(r_1) + \text{leaves}(r_2) \}$	
31	$\bar{\gamma} \llbracket (\alpha, m, e) \rrbracket$	$= \gamma \llbracket (\alpha, m, \emptyset, e) \rrbracket$

Figure 9. Denotational semantics of Floorplan.

5.2.1 Denotations: reducing Code F6 to core values

Figure 9 shows our core denotational semantics, the first three parameters of which are γ: α, m, and θ. α represents the base address of a memory layout, m represents the precise number of bytes in which the layout must fit, and θ represents a name environment. As the compilation rules (upcoming in Section 5.3) are not particularly important to understand core Floorplan semantics, we give Code F6 translated to the core calculus here:

$$\begin{aligned}
&\mathsf{K} :: (\exists\, n\ .\ \mathrm{Con}\ 5\ (\\
&\quad (\mathtt{hd} ::\ n\ \#\ (\mathrm{Prim}\ 1)) + \\
&\quad (\mathtt{tl} ::\ n\ \#\ ((\mathtt{lft} :: (\mathrm{Prim}\ 1)) \\
&\qquad + (\mathtt{rgt} :: (\exists\, f_0\ .\ f_0\ \#\ (\mathrm{Prim}\ 1))))))
\end{aligned} \tag{C6}$$

Of note on the fourth line of C6, the existentially bound f_0 variable materializes by way of Rule (2) of Figure 10. Furthermore, listed below are the steps through the semantics in that figure for reducing Code C6 to the hd sub-branch of the left-most tree K_0 of Figure 7:

Line	**Exp**	**Trees**	**Step**
17,18	$\mathsf{K} ::\ e_1$	$\gamma \llbracket 0, 5, \emptyset, e_1 \rrbracket$	Pick $m = 5$
19,20	$e_1 = \exists\, n\ .\ e_2$	Let $\theta_1 = \{n \mapsto 1\}$ in $\gamma \llbracket 0, 5, \theta_1, e_2 \rrbracket$	Pick $i = 1$
11,12	$e_2 = \mathrm{Con}\ 5\ e_3$	$\gamma \llbracket 0, 5, \theta_1, e_3 \rrbracket$	Reduce Con
1,2	$e_3 = e_{\mathtt{hd}} + e_{\mathtt{tl}}$	$\gamma \llbracket 0, 1, \theta_1, e_{\mathtt{hd}} \rrbracket$	Pick $i = 1$
17,18	$e_{\mathtt{hd}} = \mathrm{hd} ::\ e_4$	$\gamma \llbracket 0, 1, \theta_1, e_4 \rrbracket$	Reduce name
21,27	$e_4 = n\,\#\,e_5$	$\gamma \llbracket 0, 1, \theta_1, e_5 \rrbracket$	Pick $i = 1$
8,9	$e_5 = \mathrm{Prim}\ 1$	T (1 bytes) (0 bytes)	Eval tree
3,4	$e_4 = n\,\#\,e_5$	$\gamma \llbracket 0 + 1, 1 - 1, \theta_1, e_4 \rrbracket$	Resume (#) $\hookleftarrow$
23	$e_4 = n\,\#\,e_5$	T (0 bytes) (0 bytes)	Eval tree
3,4	$e_{\mathtt{hd}} + e_{\mathtt{tl}}$	$\gamma \llbracket 0{+}1, 5{-}1, \theta_1, e_{\mathtt{tl}} \rrbracket$	Resume (+) $\hookleftarrow$
17,18	$e_{\mathtt{tl}} = \mathrm{tl} ::\ e_6$	$\gamma \llbracket 1, 4, \theta_1, e_6 \rrbracket$	Skip $e_{\mathtt{tl}}$

We manually picked values for m and i in order to derive tree K_0. These derivation steps compute the first line of the following tree in data type form, with B0 and B1 representing 0 bytes and 1 bytes respectively:

```
N "K" (T (N "hd" (T (T B1 B0) (T B0 B0)))
  (N "tl" (T (T (N "lft" (T B1 B0))
                (N "rgt" (T (T B1 B0)
      (T (T B1 B0) (T (T B1 B0) (T B0 B0)))))
  ) (T B0 B0))))
```

Certain properties of the denotational semantics from Figure 9 have been proved[8] correct in Coq. Such properties include that γ always returns trees with m one-byte leaves and that γ is a total computable function.

5.3 Compilation rules

Figure 10 shows the rules for compiling a Floorplan surface syntax declaration into a core Floorplan expression. Figure 11 contains the definitions for translating an arithmetic expression into natural numbers.

$$\begin{aligned}
&\mathbb{C}\llbracket \langle \textit{layer-simple} \rangle \rrbracket = \\
&\mathbb{C}\llbracket\ \langle \textit{layer-id} \rangle\ (\text{‘}\mathtt{<}\text{’}\ \langle \textit{formals} \rangle\ \text{‘}\mathtt{>}\text{’})?\ \langle \textit{mag} \rangle?\ \langle \textit{align} \rangle?\ \text{‘}\mathtt{->}\text{’} \\
&\qquad \langle \textit{demarc-val} \rangle\ \rrbracket,\ f_i \in \langle \textit{formals} \rangle \\
(1)\quad &\qquad \vDash \langle \textit{layer-id} \rangle :: (\exists\, f_0\ .\ \cdots \exists\, f_n\ . \\
&\qquad\quad (\mathbb{M}\llbracket \langle \textit{mag} \rangle \rrbracket \\
&\qquad\qquad (\mathbb{C}\llbracket \langle \textit{demarc-val} \rangle \rrbracket\ @\ (\Delta_{\text{byte}} \llbracket \langle \textit{align} \rangle \rrbracket))))
\end{aligned}$$

$$\begin{aligned}
&\mathbb{C}\llbracket \langle \textit{demarc-val} \rangle \rrbracket = \mathbb{C}\llbracket\ \text{‘}\#\text{’}\ \langle \textit{demarc-val} \rangle\ \rrbracket \\
(2)\quad &\qquad \vDash \text{let } f = \mathtt{fresh}(\langle \textit{demarc-val} \rangle) \\
&\qquad\quad \text{in } \exists\, f\ .\ f\,\#\,\mathbb{C}\llbracket \langle \textit{demarc-val} \rangle \rrbracket
\end{aligned}$$

$$\begin{aligned}
&\mathbb{C}\llbracket \langle \textit{demarc-val} \rangle \rrbracket = \mathbb{C}\llbracket\ \langle \textit{formal-id} \rangle\ \langle \textit{demarc-val} \rangle\ \rrbracket \\
(3)\quad &\qquad \vDash \langle \textit{formal-id} \rangle\,\#\,\mathbb{C}\llbracket \langle \textit{demarc-val} \rangle \rrbracket
\end{aligned}$$

$$\begin{aligned}
&\mathbb{C}\llbracket \langle \textit{seq} \rangle \rrbracket = \mathbb{C}\llbracket\ \text{‘}\mathtt{seq}\text{’}\ \text{‘}\{\text{’}\ \langle \textit{demarc} \rangle_0 \cdots \langle \textit{demarc} \rangle_n\ \text{‘}\}\text{’}\ \rrbracket \\
(4)\quad &\qquad \vDash \mathbb{C}\llbracket \langle \textit{demarc} \rangle_0 \rrbracket + \cdots + \mathbb{C}\llbracket \langle \textit{demarc} \rangle_n \rrbracket
\end{aligned}$$

$$\begin{aligned}
&\mathbb{C}\llbracket \langle \textit{union} \rangle \rrbracket = \\
&\mathbb{C}\llbracket\ \text{‘}\mathtt{union}\text{’}\ \text{‘}\{\text{’}\ \langle \textit{demarc} \rangle_0 \text{‘}|\text{’} \cdots \text{‘}|\text{’} \langle \textit{demarc} \rangle_n \text{‘}\}\text{’} \rrbracket \\
(5)\quad &\qquad \vDash \mathbb{C}\llbracket \langle \textit{demarc} \rangle_0 \rrbracket \parallel \cdots \parallel \mathbb{C}\llbracket \langle \textit{demarc} \rangle_n \rrbracket
\end{aligned}$$

$$\begin{aligned}
&\mathbb{C}\llbracket \langle \textit{field} \rangle \rrbracket = \mathbb{C}\llbracket\ \langle \textit{field-id} \rangle\ \text{‘}\mathtt{:}\text{’}\ \langle \textit{demarc-val} \rangle\ \rrbracket \\
(6)\quad &\qquad \vDash \langle \textit{field-id} \rangle ::\ \mathbb{C}\llbracket \langle \textit{demarc-val} \rangle \rrbracket
\end{aligned}$$

$$\begin{aligned}
(7)\quad &\mathbb{C}\llbracket \langle \textit{ptr} \rangle \rrbracket = \mathbb{C}\llbracket\ (\langle \textit{layer-id} \rangle \mid \langle \textit{field-id} \rangle)\ \text{‘}\mathtt{ptr}\text{’}\ \rrbracket \\
&\qquad \vDash \mathbb{C}\llbracket\ 1 \text{ word}\ \rrbracket
\end{aligned}$$

$$\begin{aligned}
&\mathbb{C}\llbracket \langle \textit{enum} \rangle \rrbracket = \mathbb{C}\llbracket\ \text{‘}\mathtt{enum}\text{’}\ \text{‘}\{\text{’}\ \langle \textit{flag-id} \rangle_0 \cdots \langle \textit{flag-id} \rangle_n\ \text{‘}\}\text{’}\ \rrbracket \\
(8)\quad &\qquad \vDash \mathrm{Prim} \left\lceil \log_2(n+1) * \frac{1 \text{ byte}}{8 \text{ bits}} \right\rceil
\end{aligned}$$

$$\begin{aligned}
&\mathbb{C}\llbracket \langle \textit{bits} \rangle \rrbracket = \mathbb{C}\llbracket\ \text{‘}\mathtt{bits}\text{’}\ \text{‘}\{\text{’}\ \langle \textit{bits-exp} \rangle_0 \cdots \langle \textit{bits-exp} \rangle_n\ \text{‘}\}\text{’}\ \rrbracket \\
(9)\quad &\qquad \vDash \mathrm{Prim} \left\lceil \left(\sum_{i=0}^{n} (\Delta_{\text{bit}} \langle \textit{bits-exp} \rangle_i) \right) * \frac{1 \text{ byte}}{8 \text{ bits}} \right\rceil
\end{aligned}$$

$$(10)\quad \mathbb{C}\llbracket \langle \textit{size-arith} \rangle \rrbracket \vDash \mathrm{Prim} \left(\Delta_{\text{byte}} \langle \textit{size-arith} \rangle \right)$$

Figure 10. Compilation rules for translating surface syntax to a core expression. Syntax inside oxford-like brackets[9] is surface syntax, and syntax after a double-turnstile[10] $\vDash$ is a core expression. Formals support list membership, $\in$.

For the translation $\mathbb{C}\llbracket \langle \textit{layer} \rangle \rrbracket$ to be defined, a $\langle \textit{layer} \rangle$ must satisfy a few properties. First, all $\langle \textit{macro} \rangle$ constructs must have been eliminated and syntactically replaced with their top-level declarations. Second, the surface declaration must be validly scoped, meaning every use of a $\langle \textit{formal-id} \rangle$ must be scoped inside a $\langle \textit{layer} \rangle$ defining it. Floorplan is lexically scoped with shadowing.

In Rule (1) of Figure 10 there are three optional constructs. Each construct compiles to an expression wrapping the compilation of the containing value: $\mathbb{C}\llbracket \langle \textit{demarc-val} \rangle \rrbracket$. For

[8]Available in extended version.

[9] $\llbracket \ldots \rrbracket$ separates raised syntax (inside brackets) from lowered expressions.

[10]A double-turnstile, Foo $\llbracket \ldots \rrbracket \vDash$ Bar, reads as "Bar models Foo $\llbracket \ldots \rrbracket$".

$$
\begin{aligned}
&\mathbb{M}\llbracket\, \text{`|'}\,\langle \textit{size-arith}\rangle\,\text{`|'}\, \rrbracket(e) \models \mathrm{Con}\ \left(\Delta_{\text{bytes}}\llbracket\, \langle \textit{size-arith}\rangle\, \rrbracket\right)\ e \\
&\Delta_{\text{bit}} \equiv \Delta \\
&\Delta_{\text{byte}} \equiv \left\lceil \Delta_{\text{bit}} * \tfrac{1\text{ byte}}{8\text{ bits}} \right\rceil \\
&\Delta\llbracket\, \text{`}\texttt{bits}\text{'}\, \rrbracket \models \tfrac{1\text{ bits}}{1\text{ bit}},\ \Delta\llbracket\, \text{`}\texttt{bytes}\text{'}\, \rrbracket \models \tfrac{8\text{ bits}}{1\text{ byte}} \\
&\Delta\llbracket\, \text{`}\texttt{words}\text{'}\, \rrbracket \models \tfrac{c_w\text{ bits}}{1\text{ word}},\ \Delta\llbracket\, \text{`}\texttt{pages}\text{'}\, \rrbracket \models \tfrac{c_p\text{ bits}}{1\text{ page}} \\
&\Delta\llbracket\, \langle \textit{int}\rangle\, \rrbracket \models \langle \textit{int}\rangle \\
&\Delta\llbracket\, \langle \textit{bin}\rangle\, \rrbracket \models \texttt{int}(\langle \textit{bin}\rangle) \\
&\Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\,\text{`+'}\,\langle \textit{lit-arith}\rangle_r\, \rrbracket \models \Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\, \rrbracket + \Delta\llbracket\, \langle \textit{lit-arith}\rangle_r\, \rrbracket \\
&\Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\,\text{`-'}\,\langle \textit{lit-arith}\rangle_r\, \rrbracket \models \Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\, \rrbracket - \Delta\llbracket\, \langle \textit{lit-arith}\rangle_r\, \rrbracket \\
&\Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\,\text{`*'}\,\langle \textit{lit-arith}\rangle_r\, \rrbracket \models \Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\, \rrbracket * \Delta\llbracket\, \langle \textit{lit-arith}\rangle_r\, \rrbracket \\
&\Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\,\text{`/'}\,\langle \textit{lit-arith}\rangle_r\, \rrbracket \models \lfloor \Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\, \rrbracket / \Delta\llbracket\, \langle \textit{lit-arith}\rangle_r\, \rrbracket \rfloor \\
&\Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\,\text{`\^{}'}\,\langle \textit{lit-arith}\rangle_r\, \rrbracket \models (\Delta\llbracket\, \langle \textit{lit-arith}\rangle_l\, \rrbracket)^{\Delta\llbracket\, \langle \textit{lit-arith}\rangle_r\, \rrbracket} \\
&\Delta\llbracket\, \langle \textit{lit-arith}\rangle\ \langle \textit{size-prim}\rangle\, \rrbracket \models \Delta\llbracket\, \langle \textit{lit-arith}\rangle\, \rrbracket * \Delta\llbracket\, \langle \textit{size-prim}\rangle\, \rrbracket \\
&\Delta\llbracket\, \langle \textit{size-arith}\rangle_l\,\text{`+'}\,\langle \textit{size-arith}\rangle_r\, \rrbracket \models \Delta\llbracket\, \langle \textit{size-arith}\rangle_l\, \rrbracket + \Delta\llbracket\, \langle \textit{size-arith}\rangle_r\, \rrbracket \\
&\Delta\llbracket\, \langle \textit{size-arith}\rangle_l\,\text{`-'}\,\langle \textit{size-arith}\rangle_r\, \rrbracket \models \Delta\llbracket\, \langle \textit{size-arith}\rangle_l\, \rrbracket - \Delta\llbracket\, \langle \textit{size-arith}\rangle_r\, \rrbracket \\
&\Delta\llbracket\, \langle \textit{field-id}\rangle\,\text{`:'}\,\langle \textit{size-arith}\rangle\, \rrbracket \models \Delta\llbracket\, \langle \textit{size-arith}\rangle\, \rrbracket
\end{aligned}
$$

Figure 11. The rules for computing the in-memory size of Floorplan arithmetic. $\mathbb{M}$ defines core expressions, while Δ models computations over rational numbers. The constants c_w and c_p are architecture-specific. The `int()` function casts a binary term to an unsigned natural number n.

brevity we do not show all 9 permutations of the $\langle$*layer-simple*$\rangle$ rule, i.e. Rule (1), which represents cases where:

- If $\langle$*formals*$\rangle$ is missing, "$\exists\ f_0\ .\ \cdots \exists\ f_n\ .$" disappears.
- If $\langle$*mag*$\rangle$ is missing, "$\mathbb{M}\llbracket\, \langle \textit{mag}\rangle\, \rrbracket$" disappears.
- If $\langle$*align*$\rangle$ is missing, "$@(\Delta_{\text{byte}}\llbracket\, \langle \textit{align}\rangle\, \rrbracket)$" disappears.
- A $\langle$*magAlign*$\rangle$ becomes a $\langle$*mag*$\rangle$ and an $\langle$*align*$\rangle$.

6 Rust libraries generated and results

The Floorplan language is implemented as a compiler targeting Rust code. This section discusses the mechanics of the Floorplan library interface, i.e. how a Floorplan specification integrates with a memory manager. Curious readers should look at the Floorplan compiler source repository[11] to see all the library interfaces generated. Throughout this section numbers$_1$ on$_2$ words$_3$ refer to line numbers in Figure 12.

6.1 Code generation & library interface

Figure 12 shows a sampling of the Rust library interface generated for the immix memory layout of Figure 5. The compiler generates a struct type for each $\langle$*layer-id*$\rangle$ and $\langle$*field-id*$\rangle$. Address types$_{8,18}$ are wrappers around a word (`usize`) with no runtime overhead. Each address type implements a Rust trait called Address, providing a number of generic pointer and arithmetic operations such as load$_{13}$, store$_{15}$, plus$_{23}$, and sub$_{25}$, among others[12]. This trait requires the four deriving$_{7,17}$ clauses on each address type.

Offset constants$_{1-5}$ are generated with a particular architecture in mind (i.e. 64-bit herein). Offset constants, along with alignment constants$_{9,19}$, are in various places$_{10,20,23,25}$

```
 1  pub const CELL_0_OFFSET : usize = 0;
 2  pub const CELL_1_OFFSET : usize = 8;
 3  pub const CELL_2_OFFSET : usize = 16;
 4  pub const CELL_3_OFFSET : usize = 24;
 5  pub const PAYLOAD_OFFSET : usize = 32;
 6  #[repr(C)]
 7  #[derive(Copy, Clone, Eq, Hash)]
 8  pub struct Cell_1Addr(usize);
 9  pub const CELL_1_BYTES_ALIGN : usize = 1;
10  deriveAddr!(Cell_1Addr, CELL_1_BYTES_ALIGN);
11  impl Cell_1Addr {
12    pub fn get_cell(self) -> CellAddr {
13      self.load::<CellAddr>() }
14    pub fn set_cell(self, ptr: CellAddr) {
15      self.store(ptr); } }
16  #[repr(C)]
17  #[derive(Copy, Clone, Eq, Hash)]
18  pub struct CellAddr(usize);
19  pub const CELL_ALIGN : usize = 3;
20  deriveAddr!(CellAddr, 1 << CELL_ALIGN);
21  impl CellAddr {
22    pub fn cell_1(self) -> Cell_1Addr {
23      self.plus::<Cell_1Addr>(CELL_1_OFFSET) }
24    pub fn from_cell_1(x: Cell_1Addr) -> Self {
25      x.sub::<Self>(CELL_1_OFFSET) } }
```

Figure 12. Snippets taken from the Rust library generated for the immix memory layout of Figure 5.

throughout generated Rust code. The Floorplan compiler generates code which mimics the modularity of existing memory management systems [12, 15, 16] and frameworks [6, 7]. This form enables pain-free manual inspection of generated code.

Finally we have the four functions$_{12,24,22,24}$ generated in our example of Figure 12. The first function, get_cell$_{12}$, requires a valid `Cell_1Addr` in order to call it and returns the contents of the `cell_1` field of a cell wrapped in a `CellAddr`. The Floorplan compiler and interface provide behind-the-scenes unwrapping, accessing, and rewrapping of values (with no dynamic runtime overhead) into Rust types. In this paradigm the Rust type system enforces address-level type safety. The abundance of generated Rust address types also provided us with continual syntactic cues, telling us which address types were involved in some computation.

The main cost we see in our approach to integrating a Floorplan specification with an existing garbage collector pertains to how a generated library gets called. Upon modifying the immix specification dozens of lines of GC code would become stale, requiring manual modifications to various library call-sites. Such Rust compiler errors naturally provided us with a task list of places in the GC code to update.

While integrating generated code into the immix code base we had to make a few modest changes. The most extensive change involved modifying type signatures of nearly every functions in the garbage collector to refer to the generated

[11] https://github.com/RedlineResearch/floorplan

[12] Generic access operations are not programmer-accessible, by default.

L_o	L_f	U_o	U_f	File
24	-	3	-	common/address_map.rs
48	-	3	-	common/address_bitmap.rs
97	-	15	-	common/bitmap.rs
132	-	9	-	common/mod.rs
16	17	0	0	heap/mod.rs
27	21	3	0	objectmodel/mod.rs
28	12	0	0	heap/immix/mod.rs
42	42	0	0	obj_init.rs
51	53	2	1	mark.rs
52	53	2	0	trace.rs
72	68	3	0	lib.rs
94	94	1	1	heap/freelist/mod.rs
173	171	4	0	heap/immix/immix_mutator.rs
222	224	8	2	heap/immix/immix_space.rs
285	304	10	4	heap/gc/mod.rs
-	47	-	0	heap/flp/layout.flp
1363	1107	63	8	Total: (19% L, 87% U)
-	530	-	32	heap/flp/mod.rs
-	188	-	7	heap/flp/address.rs

Figure 13. Lines of immix source code, comparing the original code of [13] with our Floorplan-integrated version. L_o and L_f are the total number of non-empty lines in the original and Floorplan version respectively. The U_o and U_f columns indicate unsafe lines of code. The subsequent two columns shows the reduction in the number of unsafe statements in the code. Entries with a '–' indicate the file is not present in that version. The Floorplan compiler generates "heap/flp/-mod.rs" from the file "heap/flp/layout.flp". We calculate line counts ignoring blank lines, comments, and sole curly braces.

Benchmark	Original (s)	Floorplan (s)	GCs	Live (MB)
`gcbench`	30.10 ± 1.28	28.94 ± 1.84	96	134
`initobj`	12.54 ± 0.96	12.87 ± 1.21	28	114
`exhaust`	15.91 ± 0.63	15.86 ± 1.80	86	359
`trace`	12.61 ± 0.88	12.52 ± 0.58	28	114

Figure 14. Runtimes and GCs triggered per benchmark.

address types. The next most extensive change involved finding each pointer calculation in the code and replacing it with a generated version. This part was less extensive because there were fewer pointer calculations than type signatures in the code. Nearly every change made involved a one-to-one replacement of individual lines.

6.2 Results

In Figure 13, we see that the programmer must write 19% fewer lines of code, including the Floorplan specification. The first four lines of the figure indicate the address map and bitmap files are completely eliminated by switching to Floorplan. These files were replaced by "layout.flp', just above the Total line. Most other changes were line-for-line replacements such as changing untyped address variables into their correspondingly typed Floorplan address types.

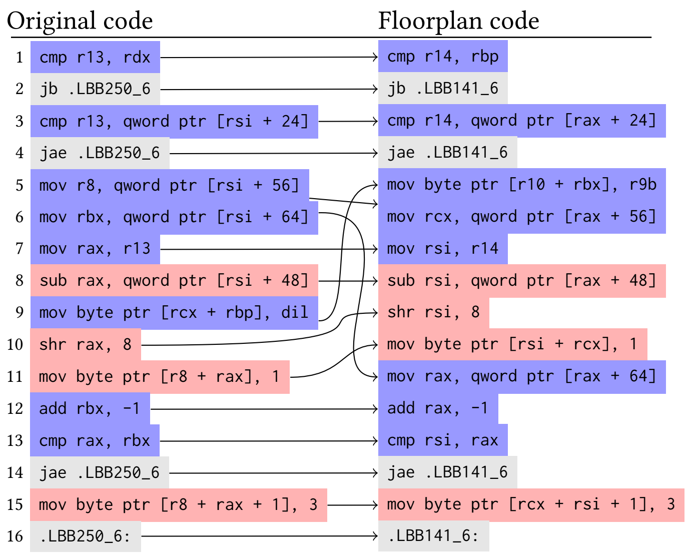
In Figure 13, we account for the number of unsafe statements of code in the implementation before and after integrating with Floorplan. A plurality of unsafe statements in the original code occur in special-purpose data structures (bitmaps) which were obviated by Floorplan. In total, the number of unsafe statements in the runtime system decreased by 87%: only 8 statements remain. Of the remaining statements, four main categories emerge: system-level allocation (2), error-handling (1), a stack-scanning FFI for C (4), and Rust vector access optimization (1).

Floorplan could reasonably handle system-level allocation, but we leave this up to the programmer for increased flexibility. The stack-scanning and error-handling lines are unsafe as a result of program control-flow, making Floorplan wholly unsuited to the task. Lastly, the unsafety of an optimized vector access would seem to be suitable for Floorplan to handle but required converting the representation of Rust data structures into Floorplan-constructed ones.

Benchmarks: We ran four benchmarks provided with the immix implementation, respectively named exhaust, initobj, gcbench, and trace. All benchmarks had internal parameters modified in order to trigger substantially more GCs than originally written for, and we recorded average runtimes and standard deviations for 100 runs of each benchmark as detailed below. A set of 5 warm-up runs of each benchmark were run prior to the 100 runs, with a 10 second cool-down in-between benchmarks. Benchmarks ran on a 12−core, 2.80 GHz Intel Xeon (X5660) processor running Arch Linux with 12 GB of RAM installed and an immix heap of 400 MB.

The benchmarks are called gcbench, initobj, trace, and exhaust; they respectively (1) construct application-level trees of certain depths, (2) stress test initialization, (3) trace freshly allocated objects, and (4) induce high memory pressure. In all cases Figure 14 shows no discernible difference between Floorplan's performance and the original benchmarks, with runtimes ranging from 10 − 30 seconds per run. This result agrees with our initial hypothesis: Floorplan generated code abstracts away common memory layout patterns without changing the performance of address computations.

We also manually inspected the assembly code generated for accessing of bitmaps for immix line liveness, reference bytes, and mark bits. Figure 15 shows the segment of code for line marking, extracted from the GC's object tracing procedure. Importantly, lines 8, 10, 11, and 15 of the original code (highlighted in red) correspond directly to four lines in the Floorplan-generated version. Those four lines respectively compute a byte offset of a cell into the heap$_8$, compute the index of the corresponding line$_{10}$, mark the line as live $_{11}$ (1 is `Live` from Figure 5), and mark the *next*$_{12}$ line as conservatively live (3 is `ConservLive`).



```
1  cmp r13, rdx
2  jb .LBB250_6
3  cmp r13, qword ptr [rsi + 24]
4  jae .LBB250_6
5  mov r8, qword ptr [rsi + 56]
6  mov rbx, qword ptr [rsi + 64]
7  mov rax, r13
8  sub rax, qword ptr [rsi + 48]
9  mov byte ptr [rcx + rbp], dil
10 shr rax, 8
11 mov byte ptr [r8 + rax], 1
12 add rbx, -1
13 cmp rax, rbx
14 jae .LBB250_6
15 mov byte ptr [r8 + rax + 1], 3
16 .LBB250_6:
```

```
cmp r14, rbp
jb .LBB141_6
cmp r14, qword ptr [rax + 24]
jae .LBB141_6
mov byte ptr [r10 + rbx], r9b
mov rcx, qword ptr [rax + 56]
mov rsi, r14
sub rsi, qword ptr [rax + 48]
shr rsi, 8
mov byte ptr [rsi + rcx], 1
mov rax, qword ptr [rax + 64]
add rax, -1
cmp rsi, rax
jae .LBB141_6
mov byte ptr [rcx + rsi + 1], 3
.LBB141_6:
```

Figure 15. x86 Intel assembly code for marking immix lines.

Additionally this code detects cells outside the heap$_{1-4}$, and detects$_{12-14}$ the last line index in the heap.[13] Control-flow instructions$_{2,4,14,16}$ are highlighted in gray, and the remaining instructions (in blue) load metadata$_{3,5,6}$ about the heap from a Rust struct. Modulo register allocation and precise instruction ordering, the purpose of each line of assembly is computed with an identical instruction opcode.

6.3 Discussion

We observe a reduction in code-base size by nearly 20% in immix-rust. This alleviates some of the technical debt of maintaining a memory manager: eliminating numerous interrelated offset constants and pointer arithmetic operations. These operations corrupt memory when applied improperly. These errors could eventually be obviated with theorem-proving techniques over Floorplan specifications.

In lieu of obviating errors, we intend to develop debugging infrastructure capable of detecting memory corruption at the first sign of layout integrity failure. A layout integrity failure occurs when a load or store operation conflicts with the addressee's intended type. The intended type of a piece of memory derives from policy decisions made earlier in a memory manager's execution. For example, after the mark phase of a mark-sweep garbage collector, certain memory cells implicitly have type "free cell". A buggy deallocation scheme can only corrupt memory in generated (unsafe) address calculations. These calculations can, and we've discovered do, encompass most all unsafe lines of code. Generated code can readily be instrumented by the Floorplan compiler.

[13] Allocating an extra entry in the line mark table would obviate these lines.

7 Related work

7.1 Declarative layout specifications

Our work is inspired by PADS [9, 10], a declarative embedded DSL for describing and parsing ad hoc data structures (PADS). PADS excels at describing log files containing textual data. For example, a PADS description encodes arrays of partitioned data. PADS captures the structure of such an array as a *type*. Floorplan too declaratively describes arrays of data. In contrast to PADS, Floorplan excels at describing heap layouts containing binary data. A Floorplan specification alone is not sufficient in order to parse raw pages.

The authors of FlashRelate [4] presented work on "a novel domain specific language called Flare that extends traditional regular expressions with [two-dimensional] spatial constraints." The underlying spatial principle of the Flare language inspired that of Floorplan: a novel domain specific language augmenting a context-free grammar with one-dimensional layout constraints. The work on FlashRelate is motivated by data-cleaning tasks and thus aims to heuristically solve the parsing of semi-structured two-dimensional data. In contrast, this work is motivated by the runtime system development task of implementing a memory manager and thus aims to deductively specify the memory layout of an unstructured one-dimensional virtual address space.

7.2 Memory management frameworks

An imperative heap layout abstraction framework known as Heap Layers [6] tackles the problem of implementing "clean, easy-to-use allocator interfaces" which are "based on C++ templates and inheritance." Heap Layers' use of template parameters is very similar to this work's notion of declaratively specifying the properties of a memory layout. Similarly, the Memory Management Toolkit (MMTk) [7] tackles the problem of implementing garbage collectors where the "resulting system is more robust, easier to maintain, and has fewer defects than monolithic collectors." As for defects related to memory layout, work on implementing an immix GC in Rust [13] aims to eliminate safety defects with static safety.

8 Conclusion

In this paper we presented a declarative language, Floorplan, for implementing the memory layout of memory managed systems in Rust. We presented a 47 line Floorplan specification for the memory layout of the state-of-the-art garbage collection algorithm immix. The compiler generated 877 lines of Rust code replacing 67 lines of pointer arithmetic, 25 lines of offset constants, and 169 lines of bitmap code.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1717373.

References

[1] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, Multicore-scalable, Low-fragmentation Memory Allocation Through Large Virtual Memory and Global Data Structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 451–469. https://doi.org/10.1145/2814270.2814294

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. 2005. The Jikes Research Virtual Machine Project: Building an Open-source Research Community. *IBM Syst. J.* 44, 2 (Jan. 2005), 399–417. https://doi.org/10.1147/sj.442.0399

[3] Krzysztof Apt. 2003. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA.

[4] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-structured Spreadsheets Using Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 218–228. https://doi.org/10.1145/2737924.2737952

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Not.* 35, 11 (Nov. 2000), 117–128. https://doi.org/10.1145/356989.357000

[6] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2001. Composing High-performance Memory Allocators. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 114–124. https://doi.org/10.1145/378795.378821

[7] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 137–146. http://dl.acm.org/citation.cfm?id=998675.999420

[8] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 22–32. https://doi.org/10.1145/1375581.1375586

[9] Kathleen Fisher, Nate Foster, David Walker, and Kenny Q. Zhu. 2011. Forest: A Language and Toolkit for Programming with Filestores. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 292–306. https://doi.org/10.1145/2034773.2034814

[10] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-specific Language for Processing Ad Hoc Data. *SIGPLAN Not.* 40, 6 (June 2005), 295–304. https://doi.org/10.1145/1064978.1065046

[11] Bradley C. Kuszmaul. 2015. SuperMalloc: A Super Fast Multithreaded Malloc for 64-bit Machines. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 41–55. https://doi.org/10.1145/2754169.2754178

[12] Doug Lea. 1991. A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html Accessed: 2018-09-28.

[13] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2016. Rust As a Language for High Performance GC Implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 89–98. https://doi.org/10.1145/2926697.2926707

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[15] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. 2008. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1375634.1375637

[16] Sun Microsystems. 2006. Hotspot. http://openjdk.java.net/groups/hotspot/ Accessed: 2018-09-28.

[17] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746