Compiling a functional array language with non-semantic memory information

Philip Munksgaard
philip@munksgaard.me
University of Copenhagen
Denmark

Cosmin Oancea
cosmin.oancea@diku.dk
University of Copenhagen
Denmark

Troels Henriksen
athas@sigkill.dk
University of Copenhagen
Denmark

ABSTRACT

We present a technique for introducing a notion of memory in the compiler representation for a parallel functional array language, in a way that allows for transformation and optimization of memory access patterns and uses, while preserving value-based semantics.

Functional languages expose no notion of memory to the user. There is no explicit allocation or deallocation of memory, nor any mapping from arrays to memory locations. Instead, the compiler must infer when and where to perform allocations as well as how values should be represented in memory. Because this affects performance, an optimizing compiler will need freedom to express various memory optimizations. Typically this is done by lowering the functional language to an imperative intermediate representation with a direct notion of memory, losing the ability to use high-level functional reasoning while performing said optimizations. We present a compiler representation where memory information is *non-semantic*, in the sense that it does not affect the program result, but only the operational details.

We start by defining a simple functional language without memory, Fun, and give it static and dynamic semantics. Next, we define an extended language, FunMem, which is Fun with non-semantic memory information in the form of lmads and an allocation statement. We give the extended language static and dynamic semantics and we provide an algorithm for transforming a Fun program into FunMem. We likewise introduce a simple imperative language Imp, which we use to illustrate how one might translate FunMem into lower-level code. Finally, we show an example of a useful transformation in FunMem, *memory expansion*, which is used to hoist allocations from parallel GPU kernels.

ACM Reference Format:
Philip Munksgaard, Cosmin Oancea, and Troels Henriksen. 2022. Compiling a functional array language with non-semantic memory information. In *Symposium on Implementation and Application of Functional Languages (IFL 2022), August 31-September 2, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3587216.3587218

1 INTRODUCTION

Functional array languages enable easy manipulation of multi-dimensional arrays, for example by means of (i) *second-order array combinators*, such as `map`, `reduce` and `scan`, that take arrays as inputs and produce new arrays as output; and (ii) *change-of-layout* operations, such as slicing, transposition or reshaping, that create new arrays by re-ordering or selecting a subset of existing arrays, but without changing their values. Such languages are promising for high level programming of parallel computers, such as GPUs.

Generating efficient array code depends on the compiler's ability to make good choices for how arrays should be allocated and represented in memory. For example:

(1) destination-passing style [13] has been used to avoid unnecessary copying of the array result of higher-order functions,
(2) functional representations of arrays, such as pull arrays [1], model fusion of parallel loops by means of function composition, thus potentially replacing accesses to memory with much cheaper accesses to registers, and
(3) region inference [14] clusters objects together whose life times are lexically bounded in order to optimize (de)allocations and to guarantee the absence of memory leaks.

Most related approaches, e.g., (1-2), use code transformations that indirectly optimize memory, but do not directly support a notion of memory in the intermediate representation (IR). Instead, memory is typically introduced when switching to an imperative representation. For example, destination-passing style cannot perfectly accommodate arbitrarily nested parallel operations, and operations that cannot be fused may require manifesting results of change-of-layout operations in memory, which is sub-optimal.

Region inference [14] (3) does propose a non-semantic[1] memory extension of the IR, but solely for tracking the lifetime of objects in order to ensure their efficient allocation and deallocation, not for expressing object layout or eliminating unnecessary copying.

This paper presents an extension of a functional IR with a non-semantic notion of memory that (i) enables efficient lowering of the memory-agnostic IR and (ii) is amenable to further optimizations. This IR is a simplified form of the one used in the compiler for Futhark [4]; a functional array language.

Our IR supports allocation of memory blocks, and, more importantly, arrays are statically associated with a memory block m together with an index function L that dictates the memory layout of its elements: e.g., for a k-dimensional array, the element at index $[i_1, \ldots, i_k]$ will be laid out at the flat memory offset $L(i_1, \ldots, i_k)$ in m. This representation has the advantage that, e.g., transposing or slicing an array just requires computing a new index function at compile time, with no additional allocation/manifestation required.

We use *linear-memory access descriptors* [5, 10], lmads, as the representation for the index function. An lmad consists of a global

[1]In this context *non-semantic* means that deleting the memory annotations in the IR results in a valid, semantically-equivalent program.

offset together with the number of elements and a total stride[2] for each dimension of the array. The LMAD-based representation:

- has negligible runtime overhead, i.e., only requires carrying around and performing computations on a couple of integers per dimension,
- is closed under affine index transformations,[3]
- represents sets of quasi-affine indices, and as such is amenable to further index-based analyses.

Our discussion is through two toy languages—FUN being memory agnostic and FUNMEM being an extension of FUN with memory information—and we present type rules and big step operational semantics for each, as well as an algorithm that translates a FUN program to a FUNMEM program. In principle, this algorithm can avoid memory manifestation for an array produced by any sequence of affine layout transformations, even when returned from `if` expressions. Although we do not prove the correctness of this algorithm, we do state the properties that must hold for the transformation to be correct. We also show a simple imperative language IMP, which we use to illustrate how a compiler might translate FUNMEM into low-level code, by giving a translation from FUNMEM to IMP.

We demonstrate that FUNMEM is a suitable IR for optimizations by presenting an algorithm for *memory expansion* in section 5. Since dynamic allocation is not efficiently supported from inside GPU kernels, memory expansion hoists inner allocations out of kernels by creating a single large memory block allocated in advance, which is then divided among the threads in non-overlapping (but possibly interleaved) chunks. We show that memory expansion can be expressed for FUNMEM in a simple and elegant way that ensures spatial locality on GPUs (i.e., coalesced access to global memory).

Contrary to FUN, FUNMEM does not guarantee that parallelism is correct by construction[4]. However, it allows the compiler to perform optimizations that are not expressible in FUN, such as

- re-using memory across arrays whose life spans do not overlap—think register allocation on arrays, or
- allowing parallel algorithms to both read and write its input arrays in-place, as showcased in [9].[5]

These transformations are briefly discussed in section 6 and have been demonstrated to result in significant speedups [9], but are otherwise out of scope for this paper.

In summary, the main contributions of this paper are:

(1) specifying the static and dynamic semantics of FUNMEM,
(2) presenting the translation from FUN to FUNMEM, and stating the properties that guarantee the translation correctness,
(3) demonstrating the memory expansion transformation on FUNMEM, which is essential for efficient execution on GPUs.

[2]The total stride is given by the distance in flat memory between two consecutive elements in that dimensions.
[3]Non-affine transformations can be represented by a list of LMADs that are "composed" in order to generate the mapping of an element, but commonly, one LMAD suffices.
[4]In the sense that it is not verifiable by the type checking rules. Parallelism correctness, i.e. data-race freedom, is still certified by the correctness of individual code transformations.
[5]Verifying the parallel semantics in this setting can be as difficult as proving the parallelism of arbitrary loops, which is infeasible for conventional type systems.

2 THE FUN LANGUAGE

We start by defining a tiny purely functional size-dependently typed core language, corresponding to a subset of the IR used in the Futhark compiler. We concern ourselves solely with expressions; not function definitions as a whole. This is insufficient to express real programs, but sufficient to describe our approach. The language can be thought of as a first-order monomorphic functional language with parallel loops.

The grammar is shown in fig. 1. We assume a denumerably infinite set of program variables, ranged over by x, y, z. We will also use superscripts and subscripts to distinguish distinct variables. We write $\overline{\alpha}$ where α is some syntactical metavariable for a sequence of αs whenever we do not need to address individual terms. We write $FV(\alpha)$ for the free variables of α.

All variable bindings are of the form $(x : \tau)$; variables can be either integers or multi-dimensional arrays of integers, with explicit sizes for each dimension. A *statement* (s) is a binding of an expression (e) with some variable bindings (called the *pattern*). A *body* (b) is a sequence of statements terminated by a result.

Expressions only occur in statements, meaning that the language is in *administrative normal form* [11], which for our purposes strongly resembles SSA. Expressions can be either scalar expressions or expressions operating on and creating arrays, and most of them are straightforward. Slices are of the form *offset* : *size*. `kernel`-expressions denote parallel execution of a body, where

$$\texttt{kernel}\ x \leq y\ \texttt{do}\ b$$

executes b y times in parallel, where in each thread x is bound to the thread-id, between 1 and y (both inclusive). Each invocation of b produces the corresponding element of the final array. A `kernel`-expression returns only one array, although the language could easily be extended to allow for multiple return values.

We also define the derived forms

$$\begin{aligned} \texttt{iota}\ x &\equiv \texttt{kernel}\ x_i \leq x\ \texttt{do in}\ (x_i) \\ \texttt{copy}\ x &\equiv \texttt{kernel}\ y_1 \leq z_1\ \cdots \texttt{kernel}\ (y_n \leq z_n)\ \texttt{do} \\ &\qquad \texttt{in}\ (x[y_1, \ldots, y_n]) \end{aligned}$$

where $z_1 \ldots z_n$ are the n dimensions of x.

2.1 Type Rules

The type rules for FUN are shown in fig. 3 and mostly standard. The most interesting part is that whenever an array variable is bound, the *sizes* of that array must be in scope; possibly by being bound within the same pattern. Note that in T-IF, we use $FV(\overline{p})$ to mean free variables in the types of each pattern, specifically array sizes, not the bound names themselves.

The judgement $\Gamma \vdash s$ states that s is a well-typed statement in context Γ. We write $\bullet$ for empty contexts and in general for empty sequences. The judgement $\Gamma \vdash \overline{p} \leftarrow b$ states that the body b can be type-checked in context Γ, and its results bound to the bindings $\overline{p}$. The base case uses a substitution S to require that any arrays returned also have their sizes similarly returned.

The judgement $\vdash b : (r, \ldots, r)$ states that b is a well-typed *program*, meaning it is a body with no free variables that returns arrays of the indicated constant sizes. The restriction is merely to keep the rules simple, and would not be present in a real implementation.

τ	::=		**Types**
	|	`int`	integer
	|	$[x] \cdots [x]$	array
v	::=		**Values**
	|	k	integer
	|	$[v, \ldots, v]$	array
p	::=	$(x : \tau)$	**Binding**
s	::=	`let` $p \cdots p = e$	**Statement**
b	::=		**Body**
	|	$s\ b$	statement
	|	`in` $(x, \cdots, x)$	result
se	::=		**Scalar expression**
	|	k	integer const
	|	x	variable
	|	$se \oplus se$	operator
e	::=		**Expressions**
	|	se	scalar expression
	|	$x[x, \ldots, x]$	index
	|	$x[x : x, \ldots, x : x]$	slice
	|	`transpose` x	transpose
	|	`if` x `then` b `else` b	conditional
	|	`kernel` $x \leq y$ `do` b	parallel loop
r	::=	$[k] \cdots [k]$	**Program result type**

Figure 1: Syntactic objects for Fun.

```
let (z_size : int) (z_arr : [z_size]) =
  if z_cond
   then  let (x : [x_size]) = iota x_size
         in (x_size, x)
   else  let (x : [y_size]) = iota y_size
         in (y_size, y)
```

Figure 2: Example of Fun statement. x_{size}, ,y_{size} **and** z_{cond} **are assumed to have been previously defined. Note how the branches of the** `if` **return not just the array value, but also the size needed for the pattern of the** `if` **binding itself.**

2.2 Operational semantics

The operational semantics for Fun are shown in fig. 4, and constitute a standard value-based semantics. Note that evaluation can "get stuck" for out-of-bounds indexing. Arrays are 1-indexed. We rely on some auxiliary functions for indexing, slicing and transposing arrays, which are defined in fig. 5. The functions "index" and and "slice" perform indexing and slicing of arrays, while "tr" transposes an array. The "nest" function is used in section 3.3.1 to unflatten a one-dimensional array. For instance

$$\text{nest}([1, 2, 3, 4], [2][2]) = [[1, 2], [3, 4]].$$

For brevity, we write $E(\alpha)$ to denote substituting every variable in the syntactic construct α with its value from E, but also to evaluate any arithmetic operations now applied to constants, following conventional arithmetic rules.

3 THE FUNMEM LANGUAGE

FunMem is an extension of Fun with a new type `mem`, a new expression form `alloc`, and a change to the syntax of variable bindings such that arrays denote their memory block and index function. The syntax is shown in fig. 6

In FunMem, `kernel` *creates* a new array in memory—we call these *fresh arrays*. We have complete freedom to decide the index function of fresh arrays, by specifying the desired index function in the pattern. Other expressions, such as `transpose` or slicing, produce *derived arrays*. These arrays are index space transformations of a prior array, and their index functions are derived from the index function of the prior array. The intuition here is that such transformations are free at runtime, which is only possible if the change-of-indexing can be implemented entirely at compile time. An example of a FunMem expression can be seen in fig. 7.

Note that this representation allows multiple arrays to concurrently co-exist in the same memory block. This is a crucial feature of the representation, but it is only safe if uses or lifetimes of the two arrays do not overlap. Although our type rules themselves do not verify this, we will return to this issue in section 3.3.1.

3.1 LMADs in FunMem

A q-dimensional array must be associated with a q-dimensional LMAD, describing the offset as well as the stride and number of elements of each dimension. Application of an LMAD

$$L = se + \{(x_1^{elems} : x_1^{stride}), \ldots, (x_q^{elems} : x_q^{stride})\}$$

is defined as:

$$L(y_1, \ldots, y_q) = se + \sum_{1 \leq i \leq q} y_i \cdot x_i^{stride} \tag{1}$$

Further, the function $\text{slice}_f(L, [y_1 : z_1, \ldots, y_q : z_q])$ defines slicing of a q-dimensional LMAD L with q offset:size pairs:

$$\begin{aligned}\text{slice}_f(L, [y_1 : z_1, \ldots, y_q : z_q]) = \\ (se + \textstyle\sum_i^q y_i x_i^{stride}) + \{(z_1 : x_1^{stride}), \ldots, (z_q : x_q^{stride})\}\end{aligned} \tag{2}$$

Similarly, $\text{index}_f(L, k)$ fixes the outermost dimension of a q-dimensional LMAD, producing a $q - 1$-dimensional LMAD:

$$\begin{aligned}\text{index}_f(L, k) = \\ (se + k \cdot x_1^{stride}) + \{(x_2^{elems} : x_2^{stride}), \ldots, (x_q^{elems} : x_q^{stride})\}\end{aligned} \tag{3}$$

The *domain* $\text{dom}(L)$ of an LMAD L constitutes all valid indexes, while the *image* $\text{img}(L)$ constitutes all addresses reachable by applying it to an in-bounds index. An LMAD can be 0-dimensional, representing the offset at which a single scalar value is stored. These do not occur in our type system, as we do not store scalars in memory, but occur in our operational semantics when defining array indexing.

$\boxed{\Gamma \vdash se : \texttt{int}}$

$$\frac{}{\Gamma \vdash k : \texttt{int}}\ [\text{T-Const}] \qquad \frac{(x : \texttt{int}) \in \Gamma}{\Gamma \vdash x : \texttt{int}}\ [\text{T-Var}] \qquad \frac{\Gamma \vdash se_1 : \texttt{int} \quad \Gamma \vdash se_2 : \texttt{int}}{\Gamma \vdash se_1 \oplus se_2 : \texttt{int}}\ [\text{T-Op}]$$

$\boxed{\Gamma \vdash s}$

$$\frac{\Gamma \vdash se : \texttt{int}}{\Gamma \vdash \texttt{let}\ (x : \texttt{int}) = se}\ [\text{T-Scalar}] \qquad \frac{(y : [z_1, \ldots, z_n]) \in \Gamma \quad \forall_1^n i.(y_i : \texttt{int}) \in \Gamma}{\Gamma \vdash \texttt{let}\ (x : \texttt{int}) = y[y_1, \ldots, y_n]}\ [\text{T-Index}]$$

$$\frac{(x : \texttt{int}) \in \Gamma \quad \Gamma \vdash \overline{p} \leftarrow b_1 \quad \Gamma \vdash \overline{p} \leftarrow b_2 \quad \forall y \in FV(\overline{p}).(y : \texttt{int}) \in \Gamma, \overline{p}}{\Gamma \vdash \texttt{let}\ \overline{p} = \texttt{if}\ x\ \texttt{then}\ b_1\ \texttt{else}\ b_2}\ [\text{T-If}]$$

$$\frac{(x_{arr} : [x_1][x_2]) \in \Gamma}{\Gamma \vdash \texttt{let}\ (y_{arr} : [x_2][x_1]) = \texttt{transpose}\ x_{arr}}\ [\text{T-Transpose}]$$

$$\frac{\forall_1^{2n} i.(y_i : int) \in \Gamma \qquad (x_{arr} : [x_1] \cdots [x_n]) \in \Gamma}{\Gamma \vdash \texttt{let}\ (z_{arr} : [y_2] \cdots [y_{2n}]) = x_{arr}[y_1 : y_2, \ldots, y_{2n-1} : y_{2n}]}\ [\text{T-Slice}]$$

$$\frac{(y : int) \in \Gamma \qquad \Gamma, (x : \texttt{int}) \vdash (z : \tau) \leftarrow b \qquad \forall y' \in FV(\tau).(y' : \texttt{int}) \in \Gamma}{\Gamma \vdash \texttt{let}\ (z : [y]\tau) = \texttt{kernel}\ x \leq y\ \texttt{do}\ b}\ [\text{T-Kernel}]$$

$\boxed{\Gamma \vdash \overline{p} \leftarrow b}$

$$\frac{\begin{array}{c} S = \{y_1 \mapsto x_1, \ldots, y_n \mapsto x_n\} \\ (x_1 : S(\tau_1)) \in \Gamma \quad \cdots \quad (x_n : S(\tau_n)) \in \Gamma \end{array}}{\Gamma \vdash (y_1 : \tau_1) \cdots (y_n : \tau_n) \leftarrow \texttt{in}\ (x_1, \cdots, x_n)}\ [\text{T-Result}] \qquad \frac{\Gamma \vdash \texttt{let}\ \overline{p}^s = e \quad \Gamma, \overline{p}^s \vdash \overline{p} \leftarrow b}{\Gamma \vdash \overline{p} \leftarrow \texttt{let}\ \overline{p}^s = e\ b}\ [\text{T-Stm}]$$

$\boxed{\vdash b : (r, \ldots, r)}$

$$\frac{\bullet \vdash (x_1 : [k_{1,1}] \cdots [k_{1,m_1}]) \cdots (x_n : [k_{n,1}] \cdots [k_{n,m_n}]) \leftarrow b}{\vdash b : ([k_{1,1}] \cdots [k_{1,m_1}], \ldots, [k_{n,1}] \cdots [k_{n,m_n}])}\ [\text{T-Prog}]$$

Figure 3: Fun type rules.

In FunMem, LMADs can use variables in scope. In the operational semantics we will also use LMAD *values* that are assumed to contain only integer constants, not variables or expressions.

As an abbreviation we define $\mathcal{R}(x_1, \ldots, x_n)$ as an index function for arrays of shape $[x_1] \cdots [x_n]$ in row-major order with zero offset.

3.2 Type rules

The type rules for FunMem are shown in fig. 8. The rules are structured similarly to the ones for Fun. The central judgement $\Gamma \vdash s$ states the well-typedness of statement s in a type context Γ. Note how T-Mem-Kernel lets us pick *any* index function for the result, as long as it supports an array of the proper shape.

The judgement $\Gamma \vdash \overline{p} \leftarrow b$ states that in a context Γ, the body b produces results matching the pattern $\overline{p}$. As with Fun, we use a substitution S but this time in addition to array sizes, it requires that any arrays returned also have all their *supporting information* (sizes, memory blocks, variables used in index functions) returned.

The judgement $\Gamma \vdash L : [x_1] \cdots [x_n]$ states that in a context Γ, L is a well-typed index function for an array of type $[x_1] \cdots [x_n]$. Similarly, the judgement $\Gamma \vdash \tau$ states that the context Γ, τ is well-typed, meaning for arrays that the memory block is bound in Γ and that the LMAD is well-typed.

The judgement $\vdash b : (r, \ldots, r)$ states that b is a well-typed program returning arrays of the sizes indicated.

The type system is by design unsound, and FunMem programs that are well-typed can still "go wrong". In particular, we allow programs that perform essentially imperative in-place updates by modifying memory blocks that are in use by other arrays, as well as expressing data races by letting distinct iterations of a `kernel` expression write to the same memory locations. The reasoning behind this rather unusual design will be discussed in section 3.4.

3.3 Operational semantics

A FunMem program can be evaluated in two different ways: using a value-based semantics equivalent to the one for Fun, as well as a

$\boxed{E \vdash e \Downarrow (v, \ldots, v)}$

$$\frac{E(x) \neq 0 \quad E \vdash b_1 \Downarrow (v_1, \ldots, v_n)}{E \vdash \texttt{if } x \texttt{ then } b_1 \texttt{ else } b_2 \Downarrow (v_1, \ldots, v_n)} \text{[E-If-True]} \qquad \frac{E(x) = 0 \quad E \vdash b_2 \Downarrow (v_1, \ldots, v_n)}{E \vdash \texttt{if } x \texttt{ then } b_1 \texttt{ else } b_2 \Downarrow (v_1, \ldots, v_n)} \text{[E-If-False]}$$

$$\frac{v = \text{conventional evaluation of } se}{E \vdash se \Downarrow (v)} \text{[E-Scalar]} \qquad \frac{v = \text{index}(E(x), E(y_1), \ldots, E(y_n))}{E \vdash x[y_1, \ldots, y_n] \Downarrow (v)} \text{[E-Index]}$$

$$\frac{v = \text{tr}(E(x))}{E \vdash \texttt{transpose } x \Downarrow (v)} \text{[E-Transpose]} \qquad \frac{v = \text{slice}(E(x), E(y_1) : E(y_2), \ldots, E(y_{2n-1}) : E(y_{2n}))}{E \vdash x[y_1 : y_2, \ldots, y_{2n-1} : y_{2n}] \Downarrow (v)} \text{[E-Slice]}$$

$$\frac{m = E(y) \quad \begin{array}{c} E, x \mapsto 1 \vdash b \Downarrow (v_1) \\ \vdots \\ E, x \mapsto m \vdash b \Downarrow (v_m) \end{array}}{E \vdash \texttt{kernel } x \leq y \texttt{ do } b \Downarrow ([v_1, \ldots, v_m])} \text{[E-Kernel]}$$

$\boxed{E \vdash b \Downarrow (v, \ldots, v)}$

$$\frac{E \vdash e \Downarrow (v_1^e, \ldots, v_n^e) \quad E, x_1 \mapsto v_1^e, \ldots, x_n \mapsto v_n^e \vdash b \Downarrow (v_1^b, \ldots, v_m^b)}{E \vdash \texttt{let } (x_1 : \tau_1) \cdots (x_n : \tau_n) = e\ b \Downarrow (v_1^b, \ldots, v_m^b)} \text{[E-Let]} \qquad \frac{E(x_i) = v_i \quad \text{for } 0 < i \leq n}{E \vdash \texttt{in } (x_1, \ldots, x_n) \Downarrow (v_1, \ldots, v_n)} \text{[E-In]}$$

Figure 4: Big-step operational semantics for Fun. We make use of some auxiliary functions from fig. 5.

$$\begin{array}{ll}
\text{index}(v, \bullet) & = v \\
\text{index}([v_1, \ldots, v_m], k_1, \ldots, k_n) & = \text{index}(v_{k_1}, k_2, \ldots, k_n) \\
\text{slice}(v, \bullet) & = v \\
\text{slice}([v_1, \ldots, v_m], k_1, k_2, k_3 \ldots, k_{2n}) & = [\text{slice}(v_{k_1}, k_3, \ldots, k_{2n}), \ldots, \text{slice}(v_{k_1+k_2}, k_3, \ldots, k_{2n})] \\
\text{tr}([[v_{1,1}, \ldots, v_{1,n}],\ \ldots, [v_{m,1}, \ldots, v_{m,n}]]) & = [[v_{1,1}, \ldots, v_{m,1}], \ldots, [v_{1,n}, \ldots, v_{m,n}]] \\
\text{nest}(v, \bullet) & = v \\
\text{nest}([v_1, \ldots, v_m], [k_1] \cdots [k_n]) & = [\text{nest}([v_1, \ldots, v_{k_n}], [k_1] \cdots [k_{n-1}]), \ldots,\ \text{nest}([v_{m-k_n}, \ldots, v_m], [k_1] \cdots [k_{n-1}])]
\end{array}$$

Figure 5: Auxiliary functions for transforming values.

heap-based semantics that is close to how it would be implemented on a machine. A FunMem program is *valid* only if its interpretation under these two semantics coincides; a notion we will make precise in section 3.3.1.

The value-based semantics is based on reducing the FunMem program to Fun and then applying the rules from fig. 4. Intuitively, this is done by turning all memory blocks into integers, all allocations into dummy integer literals, and replacing array memory information with the corresponding array type. The pertinent rewrite rules are shown on fig. 9.

The heap-based semantics, shown in fig. 10, is more complicated. We make use of a fairly standard heap abstraction, which we denote H, that maps heap labels ℓ to memory blocks, which are one-dimensional arrays of integers. We write $H[\ell, i]$ to look up the value at offset i in the block with label ℓ, and $H[\ell, i] \mapsto k$ to construct the heap H with the value at offset i in block ℓ changed to k. The `alloc` statement is used to create new labels and add the corresponding mapping to the heap.

The central judgement is

$$E; H \vdash s \Downarrow E'; H'; \boxed{\langle \mathcal{R}, \mathcal{W} \rangle}$$

which denotes the evaluation of a statement s in a value environment E, and heap H, yielding an extended value environment E' and modified heap H', as well as a *trace* of memory accesses. The trace consists of two sets of pairs of heap labels and offsets: a read-set $\mathcal{R}$ and a write-set $\mathcal{W}$. The trace is not semantically significant, but is used as a side condition in the rule E-Mem-Kernel to avoid data races, by saying that a location written in one iteration may not be used in any way by another. This is intended to allow an implementation to execute the iterations of a `kernel` construct concurrently. Given the read and write sets of k iterations, we define

τ	::=		**Types**
	|	`int`	integer
	|	$[x]\cdots[x]@x \to L$	array
	|	`mem`	memory block
v	::=		**Values**
	|	k	integer
	|	ℓ	label
	|	(ℓ, L)	label and LMAD
	|	$[v, \dots, v]$	array
e	::=		**Expressions**
	|	...	Any Fun expression
	|	alloc se	allocation
L	::=	$se + \{(x : x)\cdots(x : x)\}$	**LMAD**
r	::=	$[x]\cdots[x]$	**Program result type**
H	::=	$\ell \mapsto [k, \dots, k], H \quad \| \quad \bullet$	**Heap**

Figure 6: Syntactic objects for FunMem. Most of the grammar is unchanged from Fun, but we require different information in bindings (τ), and we add an `alloc` expression.

```
let (z_size : int)
     (z_mem : mem) (z_arr : [z_size]@z_mem → R(z_size)) =
   if z_cond
    then  let (x_mem : mem) = alloc x_size
          let (x : [x_size]@x_mem → R(x_size)) = iota x_size
          in (x_size, x_mem, x)
    else  let (y_mem : mem) = alloc y_size
          let (x : [y_size]@y_mem → R(y_size)) = iota y_size
          in (y_size, y_mem, y)
```

Figure 7: The example from fig. 2 expanded with FunMem memory information. The branches of the `if` now also return the memory blocks used for their results.

freedom from data races as follows.

$$\begin{aligned}\text{racefree}(\mathcal{R}_1, \dots, \mathcal{R}_k, \mathcal{W}_1, \dots, \mathcal{W}_k) = \\ \forall_i(\mathcal{W}_i \cap \textstyle\bigcup_{j \neq i}(\mathcal{R}_j \cup \mathcal{W}_j)) = \emptyset\end{aligned} \tag{4}$$

Intuitively, this states that the locations written by iteration i must be distinct from the locations read or written by any other iteration. Note that a location is not the same as a memory block—it is fine for two `kernel` iterations to write to the same memory block, as long as it is to *different* offsets within the block.

The only construct that modifies a heap object is `kernel`, which is where arrays are created. The only construct that reads from the heap is indexing. Arrays are represented as a pair of a heap label and an index function containing only constants.

We use yet another auxiliary function, shown in eq. (5) below, that defines how to copy a value to a specified destination, given as heap label and index function, yielding a new heap. If the value to copy is an integer k, then the destination index function must be 0-ary, consisting of only an offset. When copying an array, our type rules ensure that the source and destination index function will have domains of the same size, and we can simply copy element-wise.

$$\begin{aligned}
&\text{memcopy}(H, k, \ell, L) = (H[\ell, L()] \mapsto k, \emptyset, \{(\ell, L())\}) \\
&\text{memcopy}(H, (\ell_{src}, L_{src}), \ell_{dst}, L_{dst}) = (H', \mathcal{R}, \mathcal{W}) \\
&\quad \text{where } H' = H[\ell_{dst}, i] \mapsto k_i \\
&\quad\quad k_i = \begin{cases} H[\ell_{src}, j] & \exists\, j \in \text{dom}(L_{src}) \Rightarrow i = L_{dst}(j) \\ H[\ell_{dst}, i] & \text{otherwise} \end{cases} \\
&\quad\quad \mathcal{R} = \{(\ell_{src}, p) \mid p \in \text{img}(\ell_{src})\} \\
&\quad\quad \mathcal{W} = \{(\ell_{dst}, p) \mid p \in \text{img}(\ell_{dst})\}
\end{aligned} \tag{5}$$

3.3.1 Validity. A FunMem program is valid if it produces the same result under value-based and heap-based semantics. Using the definition of nest from fig. 5, we get the following definition:

Definition 3.1 (Validity). Let b_{mem} be a FunMem program

$$\bar{s}\ \texttt{in}\ (x_1^{mem}, x_1^{val}, \dots, x_n^{mem}, x_n^{val})$$

and b be the corresponding Fun program given by

$$b_{mem} \Rightarrow_{unmem} b$$

Then b_{mem} is *valid* if

$$\vdash b_{mem} : ([k_{1,1}]\cdots[k_{1,m_1}], \dots, [k_{n,1}]\cdots[k_{n,m_n}])$$

and

$$\vdash b_{mem} \Downarrow (\ell_1, (\ell_1, L_1), \dots, \ell_n, (\ell_n, L_n)); H$$

and

$$\vdash b \Downarrow (0, v_1, \dots, 0, v_n)$$

such that for all $1 \leq i \leq n$

$$\text{nest}(H(\ell_i), [k_{i,1}]\cdots[k_{i,m_i}]) = v_i.$$

Any transformation on a FunMem program, as well as the initial creation of a FunMem program from a Fun program that we discuss below, must preserve validity. While we do not present proofs that this is the case for our presented transformations, validity-preservation could be shown using conventional proof techniques.

3.4 The tradeoffs in FunMem

FunMem is rich enough to express observable side effects, but as we desire purely functional semantics, we defined the notion of validity in section 3.3.1. FunMem also allows us to express nondeterministic programs through data races, which is avoided through the complicated side conditions in fig. 10. Why do we not use a type system to rule out these undesirable programs, for example by using a mechanism similar to separation logic to split the heap when type-checking `kernel` constructs?

The reasoning behind FunMem's design is that it is intended as an internal representation in a research compiler, not as a formalism or calculus. While fully verified compilers such as CompCert [7] or CakeML [6] exist, implementing fully verified compiler passes remains very time-consuming. Our design prioritizes ease of transformation over ease of verification. Therefore, while we have tried to make it precise which properties must hold for a program (validity and absence of data races), so it can be argued whether a specific

$$\boxed{\Gamma \vdash s}$$

$$\frac{\Gamma \vdash se : \texttt{int}}{\Gamma \vdash \texttt{let}\ (x : \texttt{int}) = se}\ [\text{T-Mem-Scalar}] \qquad \frac{\Gamma \vdash se : \texttt{int}}{\Gamma \vdash \texttt{let}\ (x : \texttt{mem}) = \texttt{alloc}\ se}\ [\text{T-Mem-Alloc}]$$

$$\frac{(y : [z_1, \ldots, z_n]@y_{mem} \rightarrow L) \in \Gamma \quad \forall_1^n i.(y_i : \texttt{int}) \in \Gamma}{\Gamma \vdash \texttt{let}\ (x : \texttt{int}) = y[y_1, \ldots, y_n]}\ [\text{T-Mem-Index}]$$

$$\frac{(x : \texttt{int}) \in \Gamma \quad \Gamma \vdash \overline{p} \leftarrow b_1 \quad \Gamma \vdash \overline{p} \leftarrow b_2 \quad \forall (y : \tau) \in \overline{p}.\Gamma, \overline{p} \vdash \tau}{\Gamma \vdash \texttt{let}\ \overline{p} = \texttt{if}\ x\ \texttt{then}\ b_1\ \texttt{else}\ b_2}\ [\text{T-Mem-If}]$$

$$\frac{(x_{arr} : [x_1][x_2]@x_{mem} \rightarrow se + \{(x_1 : x_{s,1})(x_2 : x_{s,2})\}) \in \Gamma}{\Gamma \vdash \texttt{let}\ (y_{arr} : [x_2][x_1]@x_{mem} \rightarrow se + \{(x_2 : x_{s,2})(x_1 : x_{s,1})\}) = \texttt{transpose}\ x_{arr}}\ [\text{T-Mem-Transpose}]$$

$$\frac{(x_{arr} : [x_1] \cdots [x_n]@x_{mem} \rightarrow L_x) \in \Gamma \quad L_z = \text{slice}_f(L_x, [y_1 : y_2, \ldots, y_{2n-1} : y_{2n}])}{\Gamma \vdash \texttt{let}\ (z_{arr} : [y_1] \cdots [y_{2n-1}]@x_{mem} \rightarrow L_z) = x_{arr}[y_1 : y_2, \ldots, y_{2n-1} : y_{2n}]}\ [\text{T-Mem-Slice}]$$

$$\frac{(y : int) \in \Gamma \qquad \Gamma, (x : \texttt{int}) \vdash (z : \tau) \leftarrow b \qquad \Gamma \vdash [y]\tau}{\Gamma \vdash \texttt{let}\ (z : [y]\tau) = \texttt{kernel}\ x \leq y\ \texttt{do}\ b}\ [\text{T-Mem-Kernel}]$$

$$\boxed{\Gamma \vdash \overline{p} \leftarrow b}$$

$$\frac{\begin{array}{c} S = \{y_1 \mapsto x_1, \ldots, y_n \mapsto x_n\} \\ (x_1 : S(\tau_1)) \in \Gamma \quad \cdots \quad (x_n : S(\tau_n)) \in \Gamma \end{array}}{\Gamma \vdash (y_1 : \tau_1) \cdots (y_n : \tau_n) \leftarrow \texttt{in}\ (x_1, \cdots, x_n)}\ [\text{T-Mem-Result}] \qquad \frac{\Gamma \vdash \texttt{let}\ \overline{p}^s = e \quad \Gamma, \overline{p}^s \vdash \overline{p} \leftarrow b}{\Gamma \vdash \overline{p} \leftarrow \texttt{let}\ \overline{p}^s = e\ b}\ [\text{T-Mem-Stm}]$$

$$\boxed{\Gamma \vdash L : [x_1] \cdots [x_n]}$$

$$\frac{L = se_o + \{(x_1 : y_1) \cdots (x_n : y_n)\} \quad \forall z \in \text{FV}(L).(z : \texttt{int}) \in \Gamma}{\Gamma \vdash L : [x_1] \cdots [x_n]}\ [\text{T-Mem-LMAD}]$$

$$\boxed{\Gamma \vdash \tau}$$

$$\frac{}{\Gamma \vdash \texttt{int}}\ [\text{T-Mem-Int}] \qquad \frac{(x_{mem} : \texttt{mem}) \in \Gamma \quad \Gamma \vdash L_x : [x_1] \cdots [x_n]}{\Gamma \vdash [x_1] \cdots [x_n]@x_{mem} \rightarrow L_x}\ [\text{T-Mem-Array}]$$

$$\boxed{\vdash b : (r, \ldots, r)}$$

$$\frac{\begin{array}{c} \bullet \vdash p_1^{mem}\ p_1^{val} \cdots p_n^{mem}\ p_n^{val} \leftarrow b \\ \forall i.p_i^{mem} = (x_i^{mem} : \texttt{mem}) \\ \forall i.p_i^{val} = (x_i^{val} : [k_{i,1}] \cdots [k_{i,m_i}]@x_i^{mem} \rightarrow \mathcal{R}(k_{i,1}, \ldots, k_{i,m_i})) \end{array}}{\vdash b : ([k_{1,1}] \cdots [k_{1,m_1}], \ldots, [k_{n,1}] \cdots [k_{n,m_n}])}\ [\text{T-Mem-Prog}]$$

Figure 8: FunMem type rules. We use the definition of slice$_f$ from eq. (2). The rules for checking bodies and scalar expressions the same as for Fun and are found in fig. 3.

FunMem program is correct or not, a proof of this is not embedded in the program itself.

3.5 Transforming Fun to FunMem

Procedures TransformProgram, TransformStms and Transform-Stm show the rules for transforming a Fun program to a FunMem program. TransformProgram is the main entry-point, but works mainly by calling TransformStms and making sure that the results are in row-major order. TransformStms is equally simple, repeatedly calling TransformStm on each statement and updating the type environment in-between. TransformStm works by pattern matching on the expression and pattern inside the given statement, acting accordingly. Scalar expressions and indexing requires no processing. For slicing and transposes, we need to lookup the memory

$$\begin{array}{lll}
\texttt{mem} & \Rightarrow_{\text{unmem}} & \texttt{int} \\
\overline{[x]}@x_{mem} \rightarrow L & \Rightarrow_{\text{unmem}} & \overline{[x]} \\
\texttt{alloc}\ se & \Rightarrow_{\text{unmem}} & 0
\end{array}$$

Figure 9: Turning a FunMem program into a Fun program.

location and index function of the array being sliced or transposed so we can add the necessary information in the result pattern.

For `kernel` expressions, we first transform the inner bodies. Then we create a fresh variable for the memory block of the result and create the allocation statement. We then construct a row-major index function and finally combine it all in the pattern p'.

The case for `if` is the most complicated, as we need to return the supporting information of any arrays in the return values. We start by transforming the statements inside each body. Then, for each pair of values being returned from the branches and the pattern being matched with, we do the following: First we lookup the result types, then extract supporting information using the Support procedure, as defined in procedure Support. The Support function also creates a fresh variable for the scalar expression denoting the lmad offset, and a statement binding it. We append these statements to the transformed list of statements for each body. Then, for each supporting variable, we create a corresponding variable for the outer pattern and a substitution, which we apply to get the transformed type of the pattern.

Note that we return *all* supporting information, even though it might not be necessary. For instance, the size of the arrays returned in two branches of an `if`-statement could be invariant to the branch, meaning that the branches always return arrays of the same size. However, we defer the removal of such redundant return values to a fairly straight-forward simplifier in a later pass.

Procedure TransformProgram(prg)

```
input  : A well-typed Fun program $prg = \overline{s}$ in $(\overline{x})$.
output : A FunMem program corresponding to $prg$.
$\overline{s'}, \Gamma \longleftarrow$ TransformStms($\overline{s}, \bullet$);
$\overline{res} \longleftarrow \bullet$;
foreach $x_i$ in $\overline{x}$ do
    $\overline{[z]}@mem \rightarrow L \longleftarrow lookup(x_i, \Gamma)$;
    $mem', y \longleftarrow fresh$;
    $s_{alloc} \longleftarrow$ let ($mem'$ : mem) = alloc ($\prod z$);
    $s_{lin} \longleftarrow$ let ($y : \overline{[z]}@mem' \rightarrow \mathcal{R}(\overline{z})$) = copy $x_i$;
    Append $s_{alloc}\ s_{lin}$ to $\overline{s'}$;
    Append $y$ to $\overline{res}$;
return $s'$ in $\overline{res}$
```

4 THE IMP LANGUAGE

Imp is a simple imperative language with a parallel loop construct, and is used to show that translating FunMem to imperative code is straightforward.

Figure 11 shows the grammar for Imp, wherein we reuse the scalar expressions from Fun. It supports two types: `int` and `mem`, the latter of which is a single-dimensional integer array that must be allocated before use. Variables can be declared and assigned. Finally, conditionals are supported in the form of `if`, while `kernel` constitutes a parallel loop where each thread gets a thread identifier x. The type rules for Imp are trivial and uninteresting, so we will elide them here. An example of a program is shown in fig. 13.

Procedure TransformStms($\overline{s}$, Γ)

```
input  : A sequence of Fun statements $\overline{s}$ and the
         corresponding type environment $\Gamma$.
output : The transformed FunMem statements $\overline{s'}$ with
         inserted memory and the corresponding type
         environment $\Gamma'$.
$\overline{s'} \longleftarrow \bullet$;
$\Gamma' \longleftarrow \Gamma$;
foreach $s$ in $\overline{s}$ do
    $\overline{t} \longleftarrow$ TransformStm($s, \Gamma'$);
    foreach let $\overline{p} = e$ in $\overline{t}$ do
        Append $\overline{p}$ to $\Gamma'$;
    Append $\overline{t}$ to $\overline{s'}$;
return $s'$
```

Figure 12 shows the dynamic semantics for Imp. The rules are fairly conventional. We have a mutable environment E that maps variable names to values, and a heap H that maps labels to memory blocks. The evaluation judgement

$$H; E \vdash s \rightarrow H'; E'$$

is read "evaluation of statement s with heap H and environment E produces a new heap H' and environment E'.

The actual rules are straightforward. The `alloc` statement creates a new label and adds a corresponding fresh block to the heap.

Procedure FunMemToImp translates a FunMem program to Imp, which is somewhat tedious but fairly straightforward. The most significant detail is that we must apply index functions symbolically to translate accesses of the multidimensional arrays of FunMem into the single-dimensional arrays of Imp, which can be seen in e.g. the case for array indexing.

5 MEMORY EXPANSION

Memory expansion (MemoryExpand) is a technique for hoisting allocations out of kernels. This is critical for compilation targeted at GPUs, since GPU kernels cannot efficiently allocate memory. Although neither Fun nor FunMem prohibit allocations inside `kernel` expressions, generation of working GPU code must use memory expansion to hoist out such allocations. This is accomplished by:

(1) allocating shared memory blocks big enough to accommodate the sum of memory requirements of all threads—i.e., of all (parallel) iterations of the `kernel` body, and
(2) modifying the index functions of the arrays created inside the kernel body to refer to the shared blocks, in a way that satisfies the property that any two threads use non-overlapping partitions of the shared block.

For simplicity of exposition we treat here only the case when the size of the inner allocation is the same for all threads, i.e., invariant

$$\boxed{E; H \vdash s \Downarrow E; H; \boxed{\langle \mathcal{R}, \mathcal{W} \rangle}}$$

$$\frac{E(se) = m \quad \ell\ fresh \quad E' = E, y \mapsto \ell \quad H' = H, \ell \mapsto [\overbrace{0, \ldots, 0}^{m}]}{E; H \vdash \texttt{let}\ (y : \texttt{mem}) = \texttt{alloc}\ se \Downarrow E'; H'; \boxed{\langle \emptyset, \emptyset \rangle}}\ [\text{E-Mem-Alloc}]$$

$$\frac{E(x) = (\ell_x, L_x) \quad E' = E, z \mapsto (\ell_x, \text{slice}_f(L_x, [E(y_1) : E(y_2), \ldots, E(y_{2n-1}) : E(y_{2n})]))}{E; H \vdash \texttt{let}\ (z : \tau) = x[y_1 : y_2, \ldots, y_{2n-1} : y_{2n}] \Downarrow E'; H'; \boxed{\langle \emptyset, \emptyset \rangle}}\ [\text{E-Mem-Slice}]$$

$$\frac{E(x) = (\ell_x, L_x) \quad k = L_x(E(y_1), \ldots, E(y_n)) \quad E' = E, z \mapsto H[\ell_x, k] \quad \mathcal{R} = \{(\ell_x, k)\}}{E; H \vdash \texttt{let}\ (z : \texttt{int}) = x[y_1, \ldots, y_n] \Downarrow E'; H'; \boxed{\langle \mathcal{R}, \emptyset \rangle}}\ [\text{E-Mem-Index}]$$

$$\frac{E(x) = (\ell_x, k_0 + \{(k_1 : k_2)(k_3 : k_4)\}) \quad E' = E, z \mapsto (\ell_x, k_0 + \{(k_3 : k_4)(k_1 : k_2)\})}{E; H \vdash \texttt{let}\ (y_{arr} : \tau) = \texttt{transpose}\ x_{arr} \Downarrow E'; H; \boxed{\langle \emptyset, \emptyset \rangle}}\ [\text{E-Mem-Transpose}]$$

$$\frac{\begin{array}{c} E, x \mapsto 1; H_0 \vdash b \Downarrow E_1; H_1; \boxed{\langle \mathcal{R}_1, \mathcal{W}_1 \rangle} \\ (H'_1, \boxed{\mathcal{R}'_1, \mathcal{W}'_1}) = \text{memcopy}(H_1, E_1(z_{res}), z_{mem}, \text{index}_f(L_z, 1)) \\ \vdots \\ E, x \mapsto k; H'_{k-1} \vdash b \Downarrow E_k; H_k; \boxed{\langle \mathcal{R}_k, \mathcal{W}_k \rangle} \\ (H'_k, \boxed{\mathcal{R}'_k, \mathcal{W}'_k}) = \text{memcopy}(H_k, E_k(z_{res}), z_{mem}, \text{index}_f(L_z, k)) \\ E(y) = k \quad \boxed{\text{racefree}(\mathcal{R}_1 \cup \mathcal{R}'_1, \ldots, \mathcal{R}_k \cup \mathcal{R}'_k, \mathcal{W}_1 \cup \mathcal{W}'_1, \ldots, \mathcal{W}_k \cup \mathcal{W}'_k)} \quad E' = E, z \mapsto (E(z_{mem}), E(L_z)) \end{array}}{E; H_0 \vdash \texttt{let}\ (z : [z_1^d] \cdots [z_n^d]@z_{mem} \to L_z) = \texttt{kernel}\ x \leq y\ \texttt{do}\ \bar{s}\ \texttt{in}\ (z_{res}) \Downarrow E'; H'_k; \boxed{\langle \bigcup_{i=1}^{i \leq k} \mathcal{R}_i \cup \mathcal{R}'_i, \bigcup_{i=1}^{i \leq k} \mathcal{W}_i \cup \mathcal{W}'_i \rangle}}\ [\text{E-Mem-Kernel}]$$

$$\frac{E(x) \neq 0 \quad E; H \vdash \bar{s} \Downarrow E_s; H_s; \boxed{\langle \mathcal{R}_s, \mathcal{W}_s \rangle} \quad E' = E, x_1 \mapsto E_s(y_1), \ldots, x_n \mapsto E_s(y_n)}{E; H \vdash \texttt{let}\ (x_1 : \tau_1) \cdots (x_n : \tau_n) = \texttt{if}\ x\ \texttt{then}\ \bar{s}\ \texttt{in}\ (y_1, \ldots, y_n)\ \texttt{else}\ b_2 \Downarrow E'; H_s; \boxed{\langle \mathcal{R}_s, \mathcal{W}_s \rangle}}\ [\text{E-Mem-If-T}]$$

$$\frac{E(x) = 0 \quad E; H \vdash \bar{s} \Downarrow E_s; H_s; \boxed{\langle \mathcal{R}_s, \mathcal{W}_s \rangle} \quad E' = E, x_1 \mapsto E_s(y_1), \ldots, x_n \mapsto E_s(y_n)}{E; H \vdash \texttt{let}\ (x_1 : \tau_1) \cdots (x_n : \tau_n) = \texttt{if}\ x\ \texttt{then}\ b_1\ \texttt{else}\ \bar{s}\ \texttt{in}\ (y_1, \ldots, y_n) \Downarrow E'; H_s; \boxed{\langle \mathcal{R}_s, \mathcal{W}_s \rangle}}\ [\text{E-Mem-If-F}]$$

$$\boxed{E; H \vdash \bar{s} \Downarrow E; H; \boxed{\langle \mathcal{R}, \mathcal{W} \rangle}}$$

$$\frac{E; H \vdash \texttt{let}\ \overline{p}_1 = e_1 \Downarrow E_1; H_1; \boxed{\langle \mathcal{R}_1, \mathcal{W}_1 \rangle} \quad \cdots \quad E_{n-1}; H_{n-1} \vdash \texttt{let}\ \overline{p}_n = e_n \Downarrow E_n; H_n; \boxed{\langle \mathcal{R}_n, \mathcal{W}_n \rangle}}{E; H \vdash \texttt{let}\ \overline{p}_1 = e_1\ \cdots\ \texttt{let}\ \overline{p}_n = e_n \Downarrow E_n; H_n; \boxed{\langle \bigcup_{i=1}^{i \leq n} \mathcal{R}_i, \bigcup_{i=1}^{i \leq n} \mathcal{W}_i \rangle}}\ [\text{E-Stms}]$$

$$\boxed{\vdash b \Downarrow (v, \ldots, v); H}$$

$$\frac{\bullet; \bullet \vdash \bar{s} \Downarrow E; H; \boxed{\langle \mathcal{R}, \mathcal{W} \rangle}}{\vdash \bar{s}\ \texttt{in}\ (x_1, \ldots, x_n) \Downarrow (E(x_1), \ldots, E(x_n)); H}\ [\text{E-Mem-Prog}]$$

Figure 10: Heap-based big-step operational semantics rules for FunMem. The $\boxed{\text{boxed}}$ parts serve to detect data races, but are not otherwise significant for the evaluation result. We use the definition of slice$_f$ from eq. (3) and racefree from eq. (4). For space reasons we elide the rules for scalar expressions, as they are conventional.

```
Procedure TransformStm($s$,$\Gamma$)
input   : A Fun statement $s$ : let $\overline{p} = e$ and a FunMem type
          environment $\Gamma$.
output  : FunMem statements $\overline{s'}$ corresponding to $s$ with
          inserted memory annotations and memory
          allocations if necessary.
case $e \equiv se$ or $e \equiv x[y_1, \ldots, y_n]$ do
    return $s$
case $\overline{p} \equiv (z : [z_1] \cdots [z_n])$ and
 $e \equiv x[y_1 : y_{n+1}, \ldots, y_n : y_{n+n}]$ do
    $[x_1] \cdots [x_n]@x_{mem} \rightarrow L_x \equiv lookup(x, \Gamma)$;
    $L_y \longleftarrow slice(L_x, [y_1 : y_{n+1}, \ldots, y_n : y_{n+n}])$;
    return let $(z : [z_1] \cdots [z_n]@x_{mem} \rightarrow L_y) = e$
case $\overline{p} \equiv (y : [n][m])$ and $e \equiv$ transpose $x$ do
    $[m][n]@x_{mem} \rightarrow se + \{(m : s_m)(n : s_n)\} \longleftarrow$
     $lookup(x, \Gamma)$;
    return
     let $(y : [n][m]@x_{mem} \rightarrow se + \{(n : s_n)(m : s_m)\}) = e$
case $e \equiv$ kernel $i \leq x$ do $\overline{s'}$ in $(x_{res})$ and
 $\overline{p} \equiv (y : [z_1] \cdots [z_n])$ do
    $\overline{s_{inner}}, \Gamma' \longleftarrow$ TransformStms($\overline{s'}, \Gamma$);
    $y_{mem} \longleftarrow fresh$;
    $s_{alloc} \longleftarrow$ let $(y_{mem}$ : mem) = alloc $(\prod_{i=1}^{n} z_i)$;
    $L_y \longleftarrow \mathcal{R}(z_1, \ldots, z_n)$;
    $p' \longleftarrow (y : [z_1] \cdots [z_n]@y_{mem} \rightarrow L_y)$;
    return $s_{alloc}$ let $p'$ = kernel $i \leq x$ do $\overline{s_{inner}}$ in $(x_{res})$
case $e$ = if $c$ then $\overline{s_{then}}$ in $(\overline{x})$ else $\overline{s_{else}}$ in $(\overline{y})$ do
    $\overline{s'_{then}}, \Gamma_{then} \longleftarrow$ TransformStms($\overline{s_{then}}, \Gamma$);
    $\overline{s'_{else}}, \Gamma_{else} \longleftarrow$ TransformStms($\overline{s_{else}}, \Gamma$);
    $\overline{p_{res}}, \overline{x_{res}}, \overline{y_{res}} \longleftarrow \bullet, \bullet, \bullet$;
    foreach $x_i, y_i, (z_i : \tau_z)$ in $\overline{x}, \overline{y}, \overline{p}$ do
        $\tau_x \longleftarrow lookup(x_i, \Gamma_{then})$;
        $\tau_y \longleftarrow lookup(y_i, \Gamma_{else})$;
        $(\overline{s_x}; \overline{x_{supp}}) \longleftarrow$ Support($\tau_x$);
        $(\overline{s_y}; \overline{y_{supp}}) \longleftarrow$ Support($\tau_y$);
        Append $\overline{s_x}, \overline{s_y}$ to $\overline{s'_{then}}, \overline{s'_{else}}$ respectively;
        $S, \overline{p'_i} \longleftarrow \bullet, \bullet$;
        foreach $x'$ in $\overline{x_{supp}}$ do
            $x_{res} \longleftarrow fresh$;
            Append $x' \mapsto x_{res}$ to $S$;
            Append $(x_{res}$ : int) to $\overline{p'_i}$;
        Append $(z_i : S(\tau_x))$ to $\overline{p'_i}$;
        Append $\overline{p'_i}, \overline{x_{supp}}, \overline{y_{supp}}$ to $\overline{p_{res}}, \overline{x_{res}}, \overline{y_{res}}$
         respectively
    return
     let $\overline{p_{res}}$ = if $c$ then $\overline{s'_{then}}$ in $\overline{x_{res}}$ else $\overline{s'_{else}}$ in $\overline{y_{res}}$
```

to the kernel, although the implementation in the Futhark compiler also supports certain cases of kernel-variant allocation sizes.

The procedure MemoryExpand performs memory expansion on a FunMem program. It assumes that allocations and kernel-invariant computations have been hoisted as much as possible. The

```
Procedure Support($\tau$)
input   : A FunMem type $\tau$
output  : The supporting information of $\tau$, and a statement
          binding the offset scalar expression to a variable, if
          there is one.
case $\tau \equiv$ int do
    return $(\bullet; \bullet)$
case
 $\tau = [x_1] \cdots [x_n]@x^{mem} \rightarrow se{+}\{(x_1 : x_{n+1}) \cdots (x_n : x_{2n})\}$
 do
    $y \longleftarrow fresh$;
    return (let $y = se; x^{mem}, y, x_1, \ldots, x_{2n}$)
```

τ	::=		**Types**
	|	int	integer
	|	mem	memory
s	::=		**Statements**
	|	skip	no-op
	|	$s; s$	sequencing
	|	var x : τ	declaration
	|	$x \leftarrow se$	assignment
	|	$x \leftarrow$ alloc se	allocation
	|	$x \leftarrow x[se]$	read
	|	$x[se] \leftarrow se$	write
	|	if x then s else s fi	conditional
	|	kernel $x \leq se$ do s done	parallel loop

Figure 11: Grammar for Imp, a tiny imperative, structured, and statement-oriented language. Reuses the scalar expressions from the functional representation.

```
Procedure Copy($p_x, x^{idx}, p_y$)
input   : A pattern $p_x$, an index $x^{idx}$, and a pattern $p_y$.
output  : Imp statement copying from $p_y$ to $p_x[x^{idx}]$.
$p_x \equiv (x : [z_1] \cdots [z_n]@x^{mem} \rightarrow L_x)$ ;
case $p_y \equiv (y : [z_2] \cdots [z_n]@y^{mem} \rightarrow L_y)$ do
    return
     kernel $z_2^{idx} \leq z_2$ do    $\cdots$    kernel $z_n^{idx} \leq z_n$ do
      $x^{mem}[L_x(x^{idx}, z_2^{idx}, \ldots, z_n^{idx})] \longleftarrow$
        $y^{mem}[L_y(z_2^{idx}, \ldots, z_n^{idx})]$
      done ... done
case $p_y \equiv (y$ : int) do
    return $x^{mem}[L_x(x^{idx})] \leftarrow y$
```

procedure repeatedly pattern matches the first statement s of a kernel body with an allocation of a memory block z, which is not used by the result of the kernel body. If the match succeeds, the procedure first creates a fresh variable z' to hold the allocation of the hoisted variable. Then, all array types referencing z inside the kernel-body are replaced with array types referencing z', by means of a (new) index function that exhibits a kernel-variant offset, i.e.,

$$\boxed{H;E \vdash s \rightarrow H;E}$$

$$\frac{}{H;E \vdash \texttt{var}\ x\ :\ \texttt{int} \rightarrow H;E,x \mapsto 0}\ \text{[E-Imp-Dec-Int]} \qquad \frac{}{H;E \vdash \texttt{var}\ x\ :\ \texttt{mem} \rightarrow H;E,x \mapsto \bot}\ \text{[E-Imp-Dec-Mem]}$$

$$\frac{}{H;E \vdash x \leftarrow se \rightarrow H;E,x \mapsto E(se)}\ \text{[E-Imp-Assign-Int]} \qquad \frac{}{H;E \vdash x \leftarrow y \rightarrow H;E,x \mapsto E(y)}\ \text{[E-Imp-Assign-Mem]}$$

$$\frac{}{H;E \vdash x \leftarrow y[se] \rightarrow H;E,x \mapsto H[E(y),E(se)]}\ \text{[E-Imp-Read]} \qquad \frac{}{H;E \vdash x[se_i] \leftarrow se_j \rightarrow H[E(x),E(se_i)] \mapsto E(se_j);E}\ \text{[E-Imp-Write]}$$

$$\frac{E(se)=m \quad \ell \text{ fresh} \quad H' = H, \ell \mapsto [\overbrace{0,\ldots,0}^{m}]}{H;E \vdash x \leftarrow \texttt{alloc}\ se \rightarrow H';E,x \mapsto \ell}\ \text{[E-Imp-Alloc]} \qquad \frac{H_1;E_1 \vdash s_1 \rightarrow H_2;E_2 \quad H_2;E_2 \vdash s_2 \rightarrow H_3;E_3}{H_1;E_1 \vdash s_1;s_2 \rightarrow H_3;E_3}\ \text{[E-Imp-Seq]}$$

$$\frac{E(x) \neq 0 \quad H;E \vdash s_1 \rightarrow H',E'}{H;E \vdash \texttt{if}\ x\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \rightarrow H';E'}\ \text{[E-Imp-If-True]} \qquad \frac{E(x) = 0 \quad H;E \vdash s_2 \rightarrow H';E'}{H;E \vdash \texttt{if}\ x\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \rightarrow H';E'}\ \text{[E-Imp-If-False]}$$

$$\frac{}{H;E \vdash \texttt{skip} \rightarrow H;E}\ \text{[E-Imp-Skip]} \qquad \frac{\begin{array}{c} E(se)=k \\ H;E,x \mapsto 1 \vdash s \rightarrow H'_1;E'_1 \quad \cdots \quad H'_{k-1};E,x \mapsto k \vdash s \rightarrow H'_k;E'_k \end{array}}{H;E \vdash \texttt{kernel}\ x \leq se\ \texttt{do}\ s\ \texttt{done} \rightarrow H'_k;E}\ \text{[E-Imp-Kernel]}$$

Figure 12: Big-step operational semantics rules for Imp. The heap structure is the same as for FunMem.

```
var $z_{size}$ : int; var $z_{mem}$ : mem;
if $z_{cond}$
$then$ var $x_{mem}$ : mem; $x_{mem} \leftarrow$ alloc $x_{size}$;
       kernel $i \leq x_{size}$ do $x_{mem}[i] \leftarrow i$ done;
       $z_{size} \leftarrow x_{size}$; $z_{mem} \leftarrow x_{mem}$
else   var $y_{mem}$ : mem; $y_{mem} \leftarrow$ alloc $y_{size}$;
       kernel $i \leq y_{size}$ do $y_{mem}[i] \leftarrow i$ done;
       $z_{size} \leftarrow y_{size}$; $z_{mem} \leftarrow y_{mem}$
fi
```

Figure 13: The example from fig. 7 transformed to Imp.

which depends on the thread number. Finally, s is replaced with two new statements: the allocation of z' and the kernel-statement with the updated body.

The modification of the index function shown in procedure MemoryExpand essentially corresponds to expanding the original array with an outer dimension whose length is equal to the number of kernel threads y, and to position the local array of thread x to the index x of the expanded array. This leads to good spatial locality for CPU-based systems—consecutive accesses of the same thread are localized—but is very inefficient for GPU execution because it would result in fully-uncoalesced accesses to global memory, i.e., consecutive threads would access the memory with a stride equal to the array size. For GPU execution, the strategy is to insert the expanded dimension innermost (i.e., maintaining the array in transposed form), which is accomplished with the index function:

$$se' + x + \{(x_1 : x_{n+1} \cdot y) \cdots (x_n : x_{2n} \cdot y)\}$$

The buffers for each iteration are now interleaved, but the form of the LMADs ensure that the reads and writes from each thread do not overlap. This procedure shows how an LMAD-based representation lends itself well to high-level analysis and optimization.

6 REMARKS ON IMPLEMENTATION

The Fun and FunMem languages presented above are simplified forms of the IR used in the Futhark compiler. The real IRs contain more constructs—in particular more index transformations than just `transpose`, and more ways to construct fresh arrays than `kernel`. However, the core concepts, particularly the use of LMADs and the way the necessary supporting information is passed around, is identical. The Futhark implementation also supports functions, which are implemented by adding memory blocks and LMAD information as parameters to functions.

The compiler uses type rules similar to those in fig. 8 to sanity-check the result of compiler passes. This has proven very useful for catching compiler bugs that might otherwise result in memory corruption at run time. As discussed in section 3.4, these rules do not rule out all memory errors, and there have certainly been cases where a faulty optimization silently caused miscompilation.

We have also used LMADs for two other optimizations:

Memory block merging, which uses the same memory block for multiple arrays, as long as their lifetimes do not overlap. This can reduce memory footprint in cases where dynamic memory allocation is not practical, such as within GPU kernels.

Memory short circuiting, discussed in detail in [9], which identifies arrays that will eventually be copied to some specific location (e.g. due to being concatenated with some other array), and modifies the memory block and index function of the original array such that it is constructed in-place. The "copy" can then be

```
Procedure FunMemToImp($s$, $\Gamma$)
  input  : A FunMem statement $s \equiv \texttt{let}\ \overline{p} = e$ and a FunMem
           type environment $\Gamma$ containing all patterns in the
           entire program.
  output : An Imp statement.
  case $s \equiv \texttt{let}\ (x : \tau) = x[x_1 : x_2, \ldots, x_{n-1} : x_n]$ do
    return skip
  case $s \equiv \texttt{let}\ (x : \tau) = \texttt{transpose}\ x$ do
    return skip
  case $s \equiv \texttt{let}\ (x : \texttt{int}) = se$ do
    return var $x$ : int; $x \leftarrow se$
  case $s \equiv \texttt{let}\ (x : \texttt{mem}) = \texttt{alloc}\ se$ do
    return var $x$ : mem; $x \leftarrow \texttt{alloc}\ se$
  case $s \equiv \texttt{let}\ (x : \texttt{int}) = y[x_1, \ldots, x_n]$ do
    $\Gamma(y) \equiv \ldots @y^{mem} \rightarrow L^y$ ;
    return var $x$ : int; $x \leftarrow y^{mem}[L^y(x_1, \ldots, x_n)]$
  case $s \equiv \texttt{let}\ \overline{p} = \texttt{if}\ y\ \texttt{then}\ b_1\ \texttt{else}\ b_2$ do
    $b_1 \equiv s_1^t \cdots s_n^t\ \texttt{in}\ \overline{x^t}$ ;
    $b_2 \equiv s_1^f \cdots s_n^f\ \texttt{in}\ \overline{x^f}$ ;
    $\forall (x_i : \tau_i) \in \overline{p} : \tau_i = \texttt{int} \vee \tau_i = \texttt{mem}.(x_i, \tau_i, x_i^t, x_i^f) \equiv$
      $(x'_1, \tau'_1, x_1^{t'}, x_1^{f'}), \ldots, (x'_m, \tau'_m, x_m^{t'}, x_m^{f'})$ ;
    return var $x'_1$ : $\tau'_1$; $\cdots$ ; var $x'_m$ : $\tau'_m$;
      if $y$ then
        FunMemToImp($s_1^t$); $\cdots$ ; FunMemToImp($s_n^t$);
        $x'_1 \leftarrow x_1^{t'}$; $\cdots$ ; $x'_m \leftarrow x_m^{t'}$
      else
        FunMemToImp($s_1^f$); $\cdots$ ; FunMemToImp($s_n^f$);
        $x'_1 \leftarrow x_1^{f'}$; $\cdots$ ; $x'_m \leftarrow x_m^{f'}$
      fi
  case $s \equiv \texttt{let}\ p = \texttt{kernel}\ y_i \leq y_n\ \texttt{do}\ b$ do
    $b \equiv s_1 \cdots s_m\ \texttt{in}\ (x)$;
    $\Gamma(x) \equiv \tau$;
    return kernel $y_i \leq y_n$ do
        FunMemToImp($s_1$); $\cdots$ ; FunMemToImp($s_m$);
        Copy($p, y_i, (x : \tau)$)
      done
```

implemented as a no-op. This is also used to eliminate the implicit copy at the end of `kernel` bodies that produce arrays.

7 RELATED WORK

Most compilers for functional languages will eventually lower the program being compiled to a representation that makes memory allocation, typically coinciding with a general lowering of the abstraction level, such as by translating the program to an imperative form. In our approach, the memory information is an "extension" of an underlying functional language, which maintains purely functional semantics. A similar idea can be found in Destination-Passing Style [8, 13] or region-based memory management [15], although with the key difference that we support non-lexical lifetimes of memory blocks, as well as using index functions to describe object layout. See also the work discussed in section 1.

```
Procedure MemoryExpand($prg$)
  input  : A FunMem program $prg$ where all memory
           annotations have been hoisted as much as possible.
  output : A FunMem program where allocations at the top of
           kernel calls have been expanded out.
  while prg contains a statement
    $s \equiv \texttt{let}\ \overline{p} = \texttt{kernel}\ x \leq y\ \texttt{do}\ \texttt{let}\ z = \texttt{alloc}\ se\ b$
    such that $x \notin FV(se)$ and $z$ is only used in patterns do
      $z' \longleftarrow fresh$;
      $b' \longleftarrow b$ with all $\tau$ of the form
        $[x_1] \cdots [x_n]@z \rightarrow se' + \{(x_1 : x_{n+1}) \cdots (x_n : x_{2n})\}$
        replaced with
        $[x_1] \cdots [x_n]@z' \rightarrow$
          $x \cdot se + se' + \{(x_1 : x_{n+1}) \cdots (x_n : x_{2n})\}$
      Replace $s$ with $s' =$
        $\texttt{let}\ z' = \texttt{alloc}\ (se \cdot y)$
        $\texttt{let}\ \overline{p} = \texttt{kernel}\ x \leq y\ \texttt{do}\ b'$
```

Previous work on dope vectors in ALGOL 60 [12] and APL [3] used dynamic structures similar to LMADs, but as actual metadata carried around at runtime, and with less expressive power. For instance, they cannot express the difference between row-major and column-major layouts.

The functional array language Single Assignment C (SaC) is similar to Futhark, and also uses a memory management strategy similar to ours [2], in particular distinguishing conceptual array values from the memory blocks used to store them, and using reference counting for the latter. One significant difference is that SaC's scheme uses a notion of "sub-allocations" to reference parts of memory blocks, while we use index functions, which also provides a mechanism for describing array layouts.

8 CONCLUSIONS

We have shown how to extend a functional language with a notion of memory that is rich enough to express layout and allocation optimizations. The memory annotations are non-semantic, although we have not provided an algorithm for verifying memory safety.

We have used this design in a compiler for the Futhark programming language, where it has proven effective for several years. In the compiler, the idea of extending a language with memory information is applied to several otherwise distinct intermediate representations, corresponding to the representations the compiler uses for its different compilation pipelines—sequential code, multicore code, GPU code, and so on.

One restriction of our approach is that we assume memory capacity requirements are easy to compute in advance, and object layouts can be described in a simple and systematic manner, in our case with LMADs. This is certainly the case for arrays, but may not be applicable to more complex recursive structures.

Although we have demonstrated our ideas on a purely functional language, they do not depend on purity and would still work in the presence of side effects.

REFERENCES

[1] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*. 21–30.

[2] Clemens Grelck and Kai Trojahner. 2004. Implicit Memory Management for SaC. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, Clemens Grelck and Frank Huch (Eds.). University of Kiel, Institute of Computer Science and Applied Mathematics, 335–348. greltrojifl04.pdf Technical Report 0408.

[3] Leo J Guibas and Douglas K Wyatt. 1978. Compilation and delayed evaluation in APL. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1–8.

[4] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[5] Jay Hoeflinger, Yunheung Paek, and Kwang Yi. 2001. Unified Interprocedural Parallelism Detection. *International Journal of Parallel Programming* 29(2) (2001), 185–215.

[6] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

[7] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[8] Zhitao Lin and Christophe Dubach. 2022. From Functional to Imperative: Combining Destination-Passing Style and Views. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (San Diego, CA, USA) *(ARRAY 2022)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3520306.3534502

[9] Philip Munksgaard, Troels Henriksen, Ponnuswamy Sadayappan, and Cosmin Oancea. 2022. Memory Optimizations in an Array Language. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 31, 15 pages.

[10] Yunheung Paek, Jay Hoeflinger, and David Padua. 2002. Efficient and Precise Array Access Analysis. *ACM Transactions on Programming Languages and Systems* 24(1) (2002), 65–109.

[11] Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-passing Style. *SIGPLAN Lisp Pointers* V, 1 (Jan. 1992), 288–298.

[12] Kirk Sattley. 1961. Allocation of storage for arrays in ALGOL 60. *Commun. ACM* 4, 1 (1961), 60–65.

[13] Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-Passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (Oxford, UK) *(FHPC 2017)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/3122948.3122949

[14] Mads Tofte and Lars Birkedal. 1998. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20, 4 (1998), 724–767.

[15] Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value λ-Calculus Using a Stack of Regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 188–201. https://doi.org/10.1145/174675.177855