Compiling a functional array language with non-semantic memory information

Munksgaard, Philip; Oancea, Cosmin Eugen; Henriksen, Troels

Publication date:
2022

Document version
Publisher's PDF, also known as Version of record

Document license:
Other

Citation for published version (APA):
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 LMADEs 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.

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 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], LMADEs, as the representation for the index function. An LMADE consists of a global...
offset together with the number of elements and a total stride5 for each dimension of the array. The \( \text{LMAD} \)-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—\( \text{Fun} \) being memory agnostic and \( \text{FunMem} \) being an extension of \( \text{Fun} \) with memory information—and we present type rules and big step operational semantics for each, as well as an algorithm that translates a \( \text{Fun} \) program to a \( \text{FunMem} \) program. In principle, this algorithm can avoid memory manifestation for an array produced by any sequence of affine layout transformations, even when returned from \( \text{if} \) 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 \( \text{Imp} \), which we use to illustrate how a compiler might translate \( \text{FunMem} \) into low-level code, by giving a translation from \( \text{FunMem} \) to \( \text{Imp} \).

We demonstrate that \( \text{FunMem} \) 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 \( \text{FunMem} \) in a simple and elegant way that ensures spatial locality on GPUs (i.e., coalesced access to global memory).

Contrary to \( \text{Fun} \), \( \text{FunMem} \) does not guarantee that parallelism is correct by construction\(^4\). However, it allows the compiler to perform optimizations that are not expressible in \( \text{Fun} \), 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 \( \text{FunMem} \),
2. presenting the translation from \( \text{Fun} \) to \( \text{FunMem} \), and stating the properties that guarantee the translation correctness,
3. demonstrating the memory expansion transformation on \( \text{FunMem} \), which is essential for efficient execution on GPUs.

\(^{1}\)The total stride is given by the distance in flat memory between two consecutive elements in that dimensions.

\(^{2}\)Non-affine transformations can be represented by a list of \( \text{LMADs} \) that are "composed" in order to generate the mapping of an element, but commonly, one \( \text{LMAD} \)-suffices.

\(^{3}\)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.

\(^{4}\)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.

\(^{5}\)We do state the properties that guarantee the translation correctness,

\(^{6}\)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.

2 THE \( \text{FUN} \) 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 \( \alpha \) where \( \alpha \) is some syntactical metavariable for a sequence of \( \alpha \) whenever we do not need to address individual terms. We write \( FV(a) \) for the free variables of \( a \).

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 \( \text{offset} : \text{size} \). Kernel-expressions denote parallel execution of a body, where \( \text{kernel } x \leq y \text{ do } b \) 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

\[ \text{iota } x \equiv \text{kernel } x_1 \leq x \text{ do } \text{in } (x_i) \]
\[ \text{copy } x \equiv \text{kernel } y_1 \leq z_1 \cdot \ldots \cdot \text{kernel } (y_n \leq z_n) \text{ do } \text{in } (x[y_1, \ldots, y_n]) \]

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

2.1 Type Rules

The type rules for \( \text{Fun} \) 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 \( \text{T-IR} \), we use \( \text{FV}(\beta) \) 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 \( \Gamma \). We write \( \bullet \) for empty contexts and in general for empty sequences. The judgement \( \Gamma \vdash \beta \Leftarrow b \) states that the body \( b \) can be type-checked in context \( \Gamma \), and its results bound to the bindings \( \beta \). The base case uses a substitution \( \Delta \) to require that any arrays returned also have their sizes similarly returned.

The judgement \( \Gamma \vdash b : (r_1, \ldots, r) \) 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.


<table>
<thead>
<tr>
<th>$\tau$ ::= Types</th>
<th>$\nu$ ::= Values</th>
<th>$p$ ::= $(x : \tau)$ Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>$k$</td>
<td>$(x : \nu)$</td>
</tr>
<tr>
<td>[x] ⋯ [x]</td>
<td>integer</td>
<td>$s$ ::= let $p \cdots p = e$ Statement</td>
</tr>
<tr>
<td>array</td>
<td>$[a, \ldots, a]$</td>
<td>$b$ ::= Body</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid s \ b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{in } (x, \cdots, x)$ result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$se ::= k$ Scalar expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid k$ integer const</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid x$ variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{se } @ \text{se}$ operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$e ::= se$ Expressions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid se$ scalar expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid x[x, \ldots, x]$ index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid x[x : x, \ldots, x : x]$ slice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{transpose } x$ transpose</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{if } x \text{ then } \text{else } b$ conditional</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{kernel } x \leq y \text{ do } b$ parallel loop</td>
</tr>
<tr>
<td>$r ::= [k] \cdots [k]$</td>
<td>Program result type</td>
<td>$se$ ::= $\text{size needed for the pattern of the } if \text{ binding itself.}$</td>
</tr>
</tbody>
</table>

Figure 1: Syntactic objects for FUN.

```
let (zsize : int) (zarr : [zsize]) =
if zcond
then let (x : [xsize]) = iota xsize
in (xsize, x)
else let (x : [ysize]) = iota ysize
in (ysize, y)
```

Figure 2: Example of FUN statement. $xsize$, $ysize$ and $zcond$ 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 “slice” perform indexing and slicing of arrays, while “itr” 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 $\alpha$ 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 $\text{mem}$, a new expression form $\text{alloc}$, 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

\[
L = se + \{(x_1^{\text{elem}} : x_1^{\text{stride}}), \ldots, (x_q^{\text{elem}} : x_q^{\text{stride}})\}
\]

is defined as:

\[
L(y_1, \ldots, y_q) = se + \sum_{1 \leq i \leq q} y_i \cdot x_i^{\text{stride}} \quad (1)
\]

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:stride pairs:

\[
\text{slice}_f(L, [y_1 : z_1, \ldots, y_q : z_q]) = (se + \sum_{i=1}^q y_i x_i^{\text{stride}}) + \{(z_1 : x_1^{\text{stride}}), \ldots, (z_q : x_q^{\text{stride}})\} \quad (2)
\]

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

\[
\text{index}_f(L, k) = (se + k \cdot x_k^{\text{stride}}) + \{(x_1^{\text{elem}} : x_1^{\text{stride}}), \ldots, (x_{k-1}^{\text{elem}} : x_{k-1}^{\text{stride}})\} \quad (3)
\]

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.
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 $\Gamma$. 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 $\Gamma$, 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.

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 $\Gamma$. 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 $\Gamma$, 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.

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

3.3 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
we denote $H$ and construct the heap $H$. We write

cated. We make use of a fairly standard heap abstraction, which
tion with the corresponding array type. The pertinent rewrite rules
into dummy integer literals, and replacing array memory informa-
gram to $Fun$ in section 3.3.1.

The heap-based semantics, shown in fig. 10, is more complicated.
We make use of a fairly standard heap abstraction, which we
do not be used in any way by another. This is intended to allow an
implementation to execute the iterations of a kernel construct con-
currently. Given the read and write sets of $k$ iterations, we define
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 pro-
gram 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
do not be used in any way by another. This is intended to allow an
implementation to execute the iterations of a kernel construct con-
currently. Given the read and write sets of $k$ iterations, we define
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
heap label and index function, yielding a new heap. If the value to
that defines how to copy a value to a specified destination, given as
and an index function containing only constants.

The only construct that reads from the
heap is indexing. Arrays are represented as a pair of a heap label
and an 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.

\[
\text{memcpy}(H, k, \ell, L) = (H[\ell, L()] \mapsto k, \emptyset, ((\ell, L())))
\]

where \(H' = H[\ell', L']\) if \(k_i = \begin{cases} H[\ell', j] & \text{if } j \in \text{dom}(L') \\ \text{otherwise} \end{cases}\)
and \(\mathcal{R} = \{(\ell, p) \mid p \in \text{img}(L')\}\)

\[
\mathcal{W} = \{(\ell, p) \mid p \in \text{img}(L')\}
\]

### 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 \(b_{\text{mem}}\) be a FunMem program

\[
\forall \text{ in } (x_1^{\text{mem}}, x_1^{\text{val}}, \ldots, x_n^{\text{mem}}, x_n^{\text{val}})
\]

and \(b\) be the corresponding Fun program given by

\[
b_{\text{mem}} \Rightarrow \text{unmem} b
\]

Then \(b_{\text{mem}}\) is valid if

\[
\vdash b_{\text{mem}} : ([k_1] \cdots [k_{m_1}], \ldots, [k_n] \cdots [k_{m_n}])
\]

and

\[
\vdash b \downarrow (t_1, (t_1, L_1), \ldots, t_n, (t_n, L_n)); H
\]

such that for all \(1 \leq i \leq n\)

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

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 non-deterministic 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 structures?

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
FunMem程序是正确的或不正确，证明这是不是嵌入在程序本身。

3.5 从Fun到FunMem

TransformProgram, TransformStms和TransformStm程序的规则用于将一个Fun程序转换为FunMem程序。TransformProgram是主要的入口点，但主要通过调用TransformStms并确保结果在行主序下进行，TransformStms简单地重复调用TransformStm的每个语句，并更新上下文环境。TransformStm通过模式匹配在给定的陈述中表达和模式的内部操作，相应地。标量表达式和索引不需要处理。对于切片和转置，我们需要查找内存。

Figure 8: FunMem type rules. We use the definition of slice_ from eq. (2). The rules for checking bodies and scalar expressions the same as for Fun and are found in fig. 3.
Then we create a fresh variable for the memory block of the result and is used to show that translating FunMem to imperative code is straightforward. 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 straightforward simplifier in a later pass.

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.

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
\[
E; H \vdash s \Downarrow E; H; (\mathcal{R}, W)
\]

\[
E(\text{let } (y : \text{mem}) = \text{alloc } s \Downarrow E', H; \langle (\mathcal{R}, W) \rangle)
\]

\[
E(x) = (f_x, L_x) \quad E' = E, y \mapsto f_x, \quad E', H' = H, \ell \mapsto (0, \ldots, 0)
\]

\[
E \vdash \text{let } (z : \tau) = x[y_1 : E(y_1), \ldots, E(y_n)] \Downarrow E', H'; \langle (\mathcal{R}, W) \rangle
\]

\[
E(x) = (f_x, L_x) \quad k = L_x(E(y_1), \ldots, E(y_n)) \quad E' = E, z \mapsto H[f_x, k] \quad \mathcal{R} = \{(f_x, k)\}
\]

\[
E(x) = (f_x, k_0 + \{(k_1 : k_2)(k_3 : k_4)\}) \quad E' = E, z \mapsto (f_x, k_0 + \{(k_3 : k_4)(k_1 : k_2)\})
\]

\[
E, x \mapsto 1; H_0 \vdash b \Downarrow E_1; H_1; (\mathcal{R}_1, W_1)
\]

\[
(H'_1, \mathcal{R}'_1, W'_1) = \text{memcpy}(H_1, E_1(z_{res}), z_{mem}, \text{index}(L_z, 1))
\]

\[
E(x) \neq 0 \quad E, H \vdash \exists \Downarrow E_1; H_1; (\mathcal{R}_x, W_x)
\]

\[
E \vdash \text{let } (x_1 : \tau_1) \cdots (x_n : \tau_n) = \text{if } x \text{ then } \exists \text{ in } (y_1, \ldots, y_n) \text{ else } b_2 \Downarrow E', H'; (\mathcal{R}_x, W_x)
\]

\[
E(x) = 0 \quad E, H \vdash \exists \Downarrow E_1; H_1; (\mathcal{R}_x, W_x)
\]

\[
E \vdash \text{let } (x_1 : \tau_1) \cdots (x_n : \tau_n) = \text{if } x \text{ then } b_1 \text{ else } \exists \text{ in } (y_1, \ldots, y_n) \Downarrow E', H'; (\mathcal{R}_x, W_x)
\]

\[
E, H ; \exists \Downarrow E; H; (\mathcal{R}, W)
\]

\[
E, H \vdash \text{let } \overline{p}_1 = c_1 \Downarrow E_1; H_1; (\mathcal{R}_1, W_1) \quad \cdots \quad E_n; H_n-1 \vdash \text{let } \overline{p}_n = c_n \Downarrow E_n; H_n; (\mathcal{R}_n, W_n)
\]

\[
E; H \vdash \text{let } \overline{p}_1 = c_1 \cdots \text{let } \overline{p}_n = c_n \Downarrow E_n; H_n; (\bigcup_{i=1}^{n} \mathcal{R}_i, \bigcup_{i=1}^{n} W_i)
\]

\[
\vdash b \Downarrow (\bar{u}, \ldots, \bar{v}); H
\]

\[
\bullet : \bullet \Downarrow E; H; (\mathcal{R}, W)
\]

\[
\vdash \exists \text{ in } (x_1, \ldots, x_n) \Downarrow (E(x_1), \ldots, E(x_n)); H
\]

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

-python

```python
Procedure TransformStms(s, Γ)
input : A Fun statement s : let 𝜋 = e and a FunMem type environment Γ.
output : FunMem statements 𝜋’ corresponding to s with inserted memory annotations and memory allocations if necessary.
case e ≡ se ore ≡ x[y₁, ..., yₙ] do
  | return s
  case 𝜋 ≡ (z₁ : [z₁] ... [zₙ]) and e ≡ x[y₁, ..., yₙ] @ xmem → Lₓ ≡ lookup(x, Γ);
  | Lᵧ ← slice(Lₓ, y₁, ..., yₙ + n); return let (z₁ : [z₁] ... [zₙ]) @ xmem → Lᵧ = e
  case 𝜋 ≡ (y : [n][m] and e ≡ transpose x do
  | return let (y : [n][m]) @ xmem → se + ((m : sₘ)(n : sₙ)) = e
  case e ≡ kernel i ≤ x do do in (xres) and
  | 𝜋 ≡ (y : [z₁] ... [zₙ]) do
  | {inner, Γ’s} ← TransformStms(Γ’s, Γ);
  | ymem ← fresh;
  | salloc ← let (ymem : mem) = alloc (Π[i] = zᵢ);
  | Lᵧ ← R(z₁, ..., zₙ);
  | p’ ← (y : [z₁] ... [zₙ]) @ ymem → Lᵧ;
  | return alloc ← let p’ = kernel i ≤ x do (sinner in xres)
  | case e = if c then 𝜋‘ then in (x) else 𝜋‘ else in (y) do
  | 𝜋‘ then in (x) then ← TransformStms(Γ’s, Γ);
  | 𝜋‘ else in (y) else ← TransformStms(Γ’s, Γ);
  | foreach xᵢ, yᵢ, (zᵢ : τᵢ) in (xres, yres): return ← •, •, •,
  | return
  | foreach x’ in (xres) do
  | xres ← fresh;
  | Append x’ to xres to S;
  | Append (xres : int) to p’;
  | Append (zᵢ : S(zᵢ)) to p’;
  | Append (p’, S(xres), yres) to p’;
  | return let p’ = if c then S(then) in xres else S(else) in yres

Procedure Support(τ)
input : A FunMem type τ
output : The supporting information of τ, and a statement binding the offset scalar expression to a variable, if there is one.
case τ ≡ int do
  | return (•, •)
case τ = [x₁] ... [xₙ] @ xmem → se + {(x₁ : xₙ + 1) ... (xₙ : x₂n)} do
  | return let y = se; xmem, y, x₁, ..., x₂n

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

Procedure Copy(pₓ, xidx, pᵧ)
input : A pattern 𝑝ₓ, an index xidx, and a pattern pᵧ.
output : Imp statement copying from pᵧ to pₓ[xidx].
px ≡ (x : z₁ ... zₙ) @ xmem → Lₓ;
px ≡ (y : {z₁ ... zₙ} @ ymem → Lᵧ) do
return
  | kernel xidx ≤ z₂ do ... kernel xidx ≤ zₙ do
  | x = xse
  | x = xse
  | x = x[se]
  | x[se] = se
  | if x then s else s then fold
  | | kernel x ≤ se do s then fold
  | | parallel loop

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.,
var zsize : int; var zmem : mem;
if zcond
then var xmem : mem; xmem ← alloc xsize;
   kernel i ≤ xsize do xmem[i] ← i done;
   zsize ← xsize; zmem ← xmem
else var ymem : mem; ymem ← alloc ysize;
   kernel i ≤ ysize do ymem[i] ← i done;
   zsize ← ysize; zmem ← ymem
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

\[ \text{index}(x) = (\text{index}(x) \cdot \text{length}) + \text{thread} \]

which is localized—but is very inefficient for GPU execution because it would result in fully-uncased reads 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:

\[ \text{index}(x) = (\text{index}(x) \cdot \text{length}) + \text{thread} \]

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.

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

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
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.

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, multi-core 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


