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):
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@sigkiller.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 translating 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:

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 lifetime 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 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

\[ i_1, \ldots, i_k \]

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 for each dimension of the array. The \texttt{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, \footnote{Non-affine transformations can be represented by a list of \texttt{lmads} that are "composed" in order to generate the mapping of an element, but commonly, one \texttt{lmad} suffices.}
- represents sets of quasi-affine indices, and as such is amenable to further index-based analyses.

Our discussion is through two toy languages—\texttt{Fun} being memory agnostic and \texttt{FunMem} being an extension of \texttt{Fun} with memory information—and we present type rules and big step operational semantics for each, as well as an algorithm that translates a \texttt{Fun} program to a \texttt{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 \texttt{Imp}, which we use to illustrate how a compiler might translate \texttt{FunMem} into low-level code, by giving a translation from \texttt{FunMem} to \texttt{Imp}.

We demonstrate that \texttt{FunMem} is a suitable IR for optimizations by presenting an algorithm for \textit{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 \texttt{FunMem} in a simple and elegant way that ensures spatial locality on GPUs (i.e., coalesced access to global memory).

Contrary to \texttt{Fun}, \texttt{FunMem} does not guarantee that parallelism is correct by construction\footnote{The total stride is given by the distance in flat memory between two consecutive elements in that dimensions.}. However, it allows the compiler to perform optimizations that are not expressible in \texttt{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 \cite{9}.

These transformations are briefly discussed in section 6 and have been demonstrated to result in significant speedups \cite{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 \texttt{FunMem},
2. presenting the translation from \texttt{Fun} to \texttt{FunMem}, and stating the properties that guarantee the translation correctness,
3. demonstrating the memory expansion transformation on \texttt{FunMem}, which is essential for efficient execution on GPUs.

\section{The \texttt{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 \(\overline{a}\) where \(a\) is some syntactical metavariable for a sequence of \(a\) 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 \textit{statement} \(s\) is a binding of an expression \(e\) with some variable bindings (called the \textit{pattern}). A \textit{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 \cite{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

\begin{align*}
\text{kernel} \quad x \leq y \quad \text{do} \quad b
\end{align*}

executes \(b\) 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 \texttt{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{align*}
\text{iota} \quad x \equiv \text{kernel} \quad x \leq x \quad \text{do} \quad \text{in} \quad (x_i)
\end{align*}

\begin{align*}
\text{copy} \quad x \equiv \text{kernel} \quad y_1 \leq z_1 \cdots \text{kernel} \quad (y_n \leq z_n) \quad \text{do} \\
\text{in} \quad (x[y_1, \ldots, y_n])
\end{align*}

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

\subsection{Type Rules}

The type rules for \texttt{Fun} are shown in fig. 3 and mostly standard. The most interesting part is that whenever an array variable is bound, the \textit{sizes} of that array must be in scope; possibly by being bound within the same pattern. Note that in \texttt{T-Ir}, we use \(\overline{FV(\overline{b})}\) 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 \(\Gamma \vdash \overline{b} \Leftarrow b\) states that the body \(b\) can be type-checked in context \(\Gamma\), and its results bound to the bindings \(\overline{b}\). The base case uses a substitution \(S\) to require that any arrays returned also have their sizes similarly returned.

The judgement \(\vdash b : (r_1, \ldots, r)\) states that \(b\) is a well-typed \textit{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.
Types
\[ r ::= \begin{array}{l}
| \text{int} & \text{integer} \\
| [x] \cdots [x] & \text{array}
\end{array} \]

Values
\[ v ::= \begin{array}{l}
| k & \text{integer} \\
| [a, \ldots, a] & \text{array}
\end{array} \]

Binding
\[ p ::= (x : r) \]

Statement
\[ s ::= \text{let } p \cdots p = e \]

Body
\[ b ::= \begin{array}{l}
| s b & \text{statement} \\
| \text{in } (x, \cdots, x) & \text{result}
\end{array} \]

Scalar expression
\[ se ::= \begin{array}{l}
| k & \text{integer constant} \\
| x & \text{variable} \\
| se @ se & \text{operator}
\end{array} \]

Expressions
\[ e ::= \begin{array}{l}
| se & \text{scalar expression} \\
| x[x : x, \ldots, x] & \text{index} \\
| x[x : x, \ldots, x] & \text{slice} \\
| \text{transpose } x & \text{transpose} \\
| \text{if } x \text{ then } b \text{ else } b & \text{conditional} \\
| \text{kernel } x \leq y \text{ do } b & \text{parallel loop}
\end{array} \]

Program result type
\[ r ::= [k] \cdots [k] \]

Figure 1: Syntactic objects for FUN.

\[
\text{let } (z_{\text{size}} : \text{int}) (z_{\text{arr}} : [z_{\text{size}}]) = \\
\text{if } z_{\text{cond}} \\
\text{then } \text{let } (x : [z_{\text{size}}]) = \text{iota } x_{\text{size}} \\
\text{in } (z_{\text{size}}, x) \\
\text{else } \text{let } (x : [y_{\text{size}}]) = \text{iota } y_{\text{size}} \\
\text{in } (y_{\text{size}}, y)
\]

Figure 2: Example of FUN statement. \(x_{\text{size}}, y_{\text{size}}\) and \(z_{\text{cond}}\) 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 "slice" perform indexing and slicing of arrays, while "it" 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, \text{kernel} creates a new array in memory—we call these \text{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 \text{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
\[ L = se + \{ (x^1_{\text{elem}}, x^1_{\text{stride}}), \ldots, (x_q^1_{\text{elem}}, x^1_{\text{stride}}) \} \]

is defined as:
\[ L(y_1, \ldots, y_q) = se + \sum_{1 \leq i \leq q} y_i \cdot x^1_{\text{stride}} \] \hspace{1cm} (1)

Further, the function 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:
\[ \text{slice}_f(L, [y_1, z_1, \ldots, y_q, z_q]) = (se + \sum_i y_i x_i^1_{\text{stride}}) + \{(z_1 : x_1^1_{\text{stride}}, \ldots, z_q : x^1_{\text{stride}})\} \] \hspace{1cm} (2)

Similarly, index\(_f\)(\(L, k\)) fixes the outermost dimension of a \(q\)-dimensional LMAD, producing a \(q - 1\)-dimensional LMAD:
\[ \text{index}_f(L, k) = (se + k \cdot x^1_{\text{stride}}) + \{(x_2^1_{\text{elem}}, x^1_{\text{stride}}), \ldots, (x_q^1_{\text{elem}}, x^1_{\text{stride}})\} \] \hspace{1cm} (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.
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.

### 3.2 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.

The judgement $\Gamma \vdash L : [x_1] \cdots [x_n]$ states that in a context $\Gamma$, $L$ 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 $\Gamma$, $\tau$ is well-typed, meaning for arrays that the memory block is bound in $\Gamma$ 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
\[
E \vdash e \Downarrow (v, \ldots, v)
\]

\[
E(x) \neq 0 \quad E \vdash b_1 \Downarrow (v_1, \ldots, v_n) \quad \text{[E-If-True]}
\]

\[
E \vdash \text{if } x \text{ then } b_1 \text{ else } b_2 \Downarrow (v_1, \ldots, v_n) \quad \text{[E-If-FALSE]}
\]

\[
v = \text{conventional evaluation of } se \\
E \vdash se \Downarrow (v)
\quad \text{[E-SCALAR]}
\]

\[
v = \text{index}(E(x), E(y_1), \ldots, E(y_n)) \\
E \vdash x[y_1, \ldots, y_n] \Downarrow (v)
\quad \text{[E-INDEX]}
\]

\[
v = \text{transpose}(E(x)) \\
E \vdash \text{transpose } x \Downarrow (v)
\quad \text{[E-TRANSPOSE]}
\]

\[
v = \text{slice}(E(x), E(y_1), \ldots, E(y_{2n-1}), E(y_{2n})) \\
E \vdash x[y_1, \ldots, y_{2n-1}, y_{2n}] \Downarrow (v)
\quad \text{[E-SLICE]}
\]

\[
E, x \mapsto 1 \vdash b \Downarrow (v_1) \\
m = E(y) \\
E, x \mapsto m \vdash b \Downarrow (v_m)
\]

\[
E \vdash \text{kernel } x \leq y \text{ do } b \Downarrow ([v_1, \ldots, v_m])
\quad \text{[E-KERNEL]}
\]

\[
E \vdash e \Downarrow (v_1', \ldots, v_m') \\
E, x_1 \mapsto v_1', \ldots, x_n \mapsto v_n' \vdash b \Downarrow (v_{1}'', \ldots, v_{m}'')
\quad \text{[E-LET]}
\]

\[
E(x_i) = v_i \quad \text{for } 0 < i \leq n
\quad \text{[E-IN]}
\]

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

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

**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 \(\ell\) 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 \(\ell\), and \(H[\ell, i] \mapsto k\) to construct the heap \(H\) with the value at offset \(i\) in block \(\ell\) changed to \(k\). The alloc statement is used to create new labels and add the corresponding mapping to the heap.

The central judgement is

\[
E; H \vdash s \Downarrow E'; H' \in \langle R, W \rangle
\]

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 \(R\) and a write-set \(W\). The trace is not semantically significant, but is used as a side condition in the rule \(\text{E-Mem-Kernel}\) 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
\[ \tau ::= \begin{array}{l}
| \text{int} \quad \text{integer} \\
| [x] \cdots [x]@x \rightarrow L \quad \text{array} \\
| \text{mem} \quad \text{memory block}
\end{array} \]

\[ \begin{array}{l}
\nu ::= \begin{array}{l}
| k \quad \text{integer} \\
| \ell \quad \text{label} \\
| (\ell, L) \quad \text{label and LMAD} \\
| [a, \ldots, e] \quad \text{array}
\end{array} \\
\end{array} \]

\[ \begin{array}{l}
ev ::= \begin{array}{l}
| \ldots \quad \text{Any Fun expression} \\
| \text{alloc se}
\end{array} \\
\end{array} \]

\[ \begin{array}{l}
L ::= \text{se} + \{(x: x) \cdots (x: x)\} \quad \text{LMAD} \\
r ::= [x] \cdots [x] \quad \text{Program result type} \\
H ::= t \mapsto [k, \ldots, k], H \quad \bullet \quad \text{Heap}
\end{array} \]

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

\[
\begin{align*}
\text{let } (z_{\text{size}} : \text{int}) & \quad z_{\text{mem}} : \text{mem} \quad (z_{\text{mem}} : \text{mem} || z_{\text{mem}} \rightarrow R(z_{\text{size}})) = \\
& \quad \text{if } z_{\text{cond}} \\
& \quad \text{then } (x_{\text{mem}} : \text{mem}) = \text{alloc } x_{\text{size}} \\
& \quad \text{let } (x: [x_{\text{size}}]@x_{\text{mem}} \rightarrow R(x_{\text{size}})) = \iota a x_{\text{size}} \\
& \quad \text{in } (x_{\text{size}}, x_{\text{mem}}, x) \\
& \quad \text{else } (y_{\text{mem}} : \text{mem}) = \text{alloc } y_{\text{size}} \\
& \quad \text{let } (x: [y_{\text{size}}]@y_{\text{mem}} \rightarrow R(y_{\text{size}})) = \iota a y_{\text{size}} \\
& \quad \text{in } (y_{\text{size}}, y_{\text{mem}}, y)
\end{align*}
\]

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.

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
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 TransformStm 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...
Then we create a fresh variable for the memory block of the result scalar expressions from Fun.

The case for \texttt{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.

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 \texttt{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.

\begin{procedure}
\textbf{ Procedure TransformProgram}($\texttt{prg}$)
\begin{flushleft}
\textbf{ input :} A well-typed Fun program \texttt{prg} = $\bar{x}$ in $\Gamma$.
\textbf{ output:} A FunMem program corresponding to \texttt{prg}.
\end{flushleft}
\begin{algorithmic}
\State $\bar{x}, \Gamma \leftarrow$ TransformStms($\bar{x}$, $\bullet$);
\State $\bar{\Gamma} \leftarrow \bullet$;
\For{$x_i$ in $\bar{x}$}
\State $[x]@\text{mem} \rightarrow L \leftarrow \text{lookup}(x_i, \Gamma)$;
\State mem', $y \leftarrow \text{fresh}$;
\State $\bar{s}_{\text{alloc}} \leftarrow \text{let } (\text{mem'} : \text{mem}) = \text{alloc } ([\bar{x}] z)$;
\State $\bar{s}_{\text{init}} \leftarrow \text{let } (y : [\bar{x}]@\text{mem'} \rightarrow \mathcal{R}([\bar{x}])) = \text{copy } x_i$;
\State Append $\bar{s}_{\text{alloc}}$ $\bar{s}_{\text{init}}$ to $\bar{\Gamma}$;
\State Append $y$ to $\bar{s}_{\text{init}}$;
\EndFor
\Return $\bar{s}'$ in $\bar{\Gamma}$
\end{algorithmic}
\end{procedure}

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: \texttt{int} and \texttt{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 \texttt{if}, while \texttt{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 \texttt{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
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\(_f\) 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, Γ)**

**input**: A FUN statement $s$ : let $\bar{p} = e$ and a FunMem type environment $\Gamma$.

**output**: FunMem statements $\mathcal{P}$ corresponding to $s$ with inserted memory annotations and memory allocations if necessary.

1. case $e \equiv se$ or $e \equiv x[y_1, \ldots, y_n]$ do
   - return $s$
2. case $\bar{p} \equiv (z: [z_1] \cdots [z_n])$ and $e \equiv x[y_1, y_{n+1}, \ldots, y_{n+m}]$ do
   - $[x_1] \cdots [x_n] \equiv xmem \rightarrow L_x \equiv \text{lookup}(x, \Gamma)$;
   - $L_y \leftarrow \text{slice}(L_x, y_1, y_{n+1}, \ldots, y_{n+m})$;
   - return let $(z: [z_1] \cdots [z_n]) \equiv xmem \rightarrow L_y = e$
3. case $\bar{p} \equiv (y: [n][m])$ and $e \equiv \text{transp}ose \ x \ do$
   - return let $(y: [n][m]) \equiv xmem \rightarrow se + (m: sm)(n: s_m) = e$
4. case $e \equiv \text{kernel } i \leq x \ do \ \forall r \ in (x_{res})$ and $\bar{p} \equiv (y: [z_1] \cdots [z_n])$
   - $\text{lookup}(x, \Gamma)$;
   - return let $(y: [z_1] \cdots [z_n]) \equiv xmem \rightarrow se + (n: s_m)m = e$

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.

1. case $τ \equiv \text{int } do$
   - return $(\cdot, \cdot)$
2. case $τ \equiv [x_1] \cdots [x_n] \equiv xmem \rightarrow se + \{(x_1: x_{n+1}) \cdots (x_n: x_{2n})\}$
   - $y \leftarrow \text{fresh}$;
   - return $(let y = x; xmem, y, x_1, \ldots, x_{2n})$

 Procedure **Copy(p_x, idxy, p_y)**

**input**: A pattern $p_x$, an index $idxy$, and a pattern $p_y$.

**output**: IMP statement copying from $p_y$ to $p_x[ idxy]$. 

1. case $p_x \equiv (x: [z_1] \cdots [z_n]) \equiv xmem \rightarrow L_x$
   - case $p_y \equiv (y: [z_2] \cdots [z_n]) \equiv ymem \rightarrow L_y$
     - return let $p_{\bar{x}} = c \ then \ s_{\bar{x}} \ else \ s_{\bar{y}}$
     - Append $x' \ in \ idxy$ do
       - $x_{res} \ leftarrow \text{fresh}$;
       - Append $x' \ to \ S$;
       - Append $(x_{res} \ : \ \text{int}) \ to \ \bar{p}_{\bar{x}}$;
     - Append $(z_2: S(z_2)) \ to \ \bar{p}_{\bar{y}}$
     - Append $p_{\bar{x}}, x_{supp}, y_{supp} \ to \ pres, s_{\bar{x}}, y_{res}$ respectively;
     - return let $p_{res} = c \ then \ s_{\bar{x}} \ else \ s_{\bar{y}}$ in $\bar{x}_{res} \ else \ s_{\bar{x}} \ else \ in \ y_{res}$

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

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 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.,

to the kernel, although the implementation in the Futhark compiler also supports certain cases of kernel-variant allocation sizes.
which depends on the thread number. Finally, s is replaced with two new statements: the allocation of \( z'=z \) 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
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 FunMemToImp(s, Γ)

input: A FunMem statement s = let p = e and a FunMem type environment Γ containing all patterns in the entire program.

output: An Imp statement.

case s ≡ let (x : r) = x[x1 : x2, ..., xn-1 : x_n] do
  return skip
case s ≡ let (x : r) = transpose x do
  return skip
case s ≡ let (x : int) = se do
  return var x : int; x ← se

case s ≡ let (x : mem) = alloc se do
  return var x : mem; x ← alloc se

case s ≡ let (x : int) = y[x1, ..., xn] do
  Γ(y) ≡ ... @y^mem → L y ;
  return var x : int; x ← y^mem[Ly(x1, ..., xn)]

case s ≡ let p = if y then b1 else b2 do
  b1 = s1' ··· s_n' in x';
  b2 = s';
  ∀ (x1 : τ1) ∈ p : τ = int ∨ τ = mem.(x1, τ1, x1') ≡
  (x1', τ1', x1', f_1') =
  return var x' : τ1';
  if y then
    FunMemToImp(s1'); ...; FunMemToImp(s_n');
    x_1 ← x_1' ; ...; x_m ← x_m'
  else
    FunMemToImp(s1'); ...; FunMemToImp(s_n');
    x_1' ← x_1'; ...; x_m' ← x_m'
f1

case s ≡ let p = kernel. y_i ≤ y_n do b do
  b ≡ s_1 ··· s_m in (x);
  Γ(x) ≡ τ;
  return kernel y_i ≤ y_n do
    FunMemToImp(s1'); ...; FunMemToImp(s_m);
    copy(p, y_i, (x : r))
done

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 = let p = kernel x ≤ y do let z = alloc se b
  such that x ∉ FV(se) and z is only used in patterns do
     z' ← fresh;
     b' ← b with all τ of the form
     [x_1] ··· [x_n]@z ⇒ se' + {(x_1 : x_1+1) ··· (x_n : x_2n)}
     replaced with
     [x_1] ··· [x_n]@z' ⇒
     x · se + se' + {(x_1 : x_1+1) ··· (x_n : x_2n)}
  Replace s with s' =
  let z' = alloc (se · y)
  let p = kernel x ≤ y 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


