Transformations for array programming

Ole Jørgen Abusdal

Master Thesis
April 4, 2020

Department of Informatics
University of Bergen

Supervisors
Magne Haveraaen
Jakko Jarvi

Abstract

We present transformations and the domain of array programming, different takes, the specification of identities for array programming and how such identities can enable a transformation based approach to array programming.

Acknowledgements

To the people of the PUT research group whose helpful advice, discussions and positive spirit have been crucial and my parents for their support through this time.

eX^3

This thesis has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

Contents

CHAPTER 1

Introduction

1.1 Background

In typed programming languages a declaration of types and operations on those types can be regarded as a specification. This degree of specification can be found in the C programming language. For instance the library function memcpy whose prototype can be found in the C standard library header string.h as

```
void *memcpy(void *dest, const void *src, size_t n);
```

one finds that the more important pieces of information are not conveyed through the function prototype, but its documentation. A precondition for calling the function with well defined behavior is that the pointers must not be null, invalid or point to overlapping memory segments. The documentation is paramount in understanding the effects of calling the function and what the return value is in relation to its parameters.

Certain languages and libraries incorporate some form of specification to describe relationships between types and operations. When the language admits certain properties such as referential transperacy it becomes particularly easy to reason about it algebraicly. An expression is referred to as referentially transparent if it can be replaced by its corresponding value without changing the meaning of the program. Clearly not the case for memcpy although we know the value it would return on a successful call is the destination pointer.

The subset of C that deals with certain arithmethic expressions can be considered referentially transparent, given that we remain within the bounds of the arithmethic type(i.e. we do not cause undefined behavior). We could adopt the point of view that when we write arithmetical expressions in C we only ever express our intentions. A compiler then translates our intentions to some machine instructions realizing them that may however very well entirely reorder our arithmetic expressions according to the usual rules of arithmetic should it prove better by some measure. Indeed compiling a snippet of integer arithmetic in C and looking at the relevant assembly output shows this to us: We see the compiler has chosen the rewrite $(a*b)+(a*c) = a*(b+c)$ as we

```
int distributive
(int a,int b, int c)
{
  return a*b+a*c;
}
```

```
distributive:
lea  eax, [rdx+rsi]
imul eax, edi
ret
```

can only discern one multiplication in the assembly code. Note that the same does not hold true for floating point arithmetic unless special permission is given to apply transformations that may lead to a loss of accuracy. There is

```
float nondistributive
(float a, float b, float c)
{
  return a*b+a*c;
}
```

```
nondistributive:
mulss  xmm1, xmm0
mulss  xmm0, xmm2
addss  xmm0, xmm1
ret
```

some implicit algebra, usually conveyed through a language standard, which the compiler takes advantage of and subjects code to value preserving transformations on the basis of identities present in the algebra.

With algebraic specification libraries can be empowered with the possibility of the very same abilities. Perhaps implicit algebras can be moved from compilers to explicit algebras in libraries?

In a language where libraries can also encode relationships between types and between operations such that code can be transformed according to these relationships we may avoid the careful deliberation needed in using libraries;

instead of crafting optimal combinations of operations often resulting in arcane code, we can write idiomatic code and rest assured that we will still get optimal code. A consideration that can often make expressions diverge quite a lot from what we would typically find in the domain we're modeling is how certain chains of operations incur an overhead of temporary data structures from intermediary results that could be eliminated if the operations were combined.

Relying on transformations, for which a first step is the ability to specify relationships, we could avoid expressing computations in terms of implementation minutia for efficiency reasons and move towards expressing computations as commonly done in various domains. We can summarize the benefits transformations could yield as:

- Separation of concerns: Transformations can be captured as general rules or specializations in a library and do not need to proliferate and thereby obfuscate the intent of code everywhere. Repeatedly transforming certain patterns by hand should be considered a violation of Don't Repeat Yourself (DRY).
- Domain knowledge: Many domains are rich with identities that can be used to direct program transformations and optimizations. By capturing algebras and identities in a suited manner for transformations we can take advantage of domain knowledge.

From a software engineering perspective the former is of great benefit, although we will mostly explore the latter. As programmers we typically consider the whole of formalisms exposed by a library to be an Application Programming Interface (API). An API is our vocabulary as programmers when attempting to express something in terms of what an API models. Bare specifications in APIs just involve types and operations along with documenation. Indeed documentation is often a part of specification; we attempt to convey in natural language properties of the API that may not be possible to convey within the programming language itself.

Gradually parts of what can been considered informal documentation has made its way into the languages proper in some formal form. A type of specification applied to APIs are pre/post-conditions; for example that a stack must not be empty when attempting to pop an item off the stack and that a stack is not empty after pushing an item onto the stack. The use of pre/post-conditions in specification can be found in SPARK/ADA and is possibly making its way into C++ [19]. The possibility of specifying such properties in the language such that it can aid the compiler in verifying code correctness is a step up from attempting to manually abide by contracts presented in documentation.

In contrast to pre/post-condition specification that typically deals with program state pre/post-call of some operation, algebraic specification deals with the relationships between different operations in an API [3].

1.1.1 Summary

Specification is a part of languages and libraries in different forms. For instance usually when we ask a compiler to compile and optimize a program containing integer arithmetic it often does so following an implicitly specified algebra; the usual rules of arithmetic algebra. This algebra may be entirely defined or at least referenced in the language specification. For instance in the case of Java the language specification for additive operations on numeric types [26] properties such as commutativty, associativity and overlfow semantics for signed integers types is specified.

Certain programming languages such as Magnolia [14] go a step further and enable one to specify explicit algebras with its concept language construct, making intended algebras a part of the language proper.

1.2 Outline

This thesis is a treatment of a transformation based approach to array programming with some emphasis on the use of Magnolia, a language developed at BLDL, for specification. We may at times take a less formal and discussion based approach to conveying the material. The thesis is structured as follows:

1. Introduction: We have roughly presented the domain and shown that the use of axioms and transformations in programming is already present in the form of compiler optimizations for builtin types.
2. Transformations: We discuss different approaches to transformations; metaprogramming and rewriting systems.
3. Arrays: We cover the basic array primitives typically available in languages such as C and Fortran their layout in memory. Here we start building a vocabulary of arrays.
4. A mathematics of arrays: We describe parts of a rich formalism for arrays and operations on arrays which abstracts away memory layout

yet accomodates the ability to select memory layouts.

5. Modeling a mathematics of arrays in Magnolia: We model the Mathematics of Arrays in a limited subset of Magnolia.

6. Other array abstractions and libraries: We present two different takes on arrays from two different frameworks; Petalisp and Lift.

7. A Mathemathics of Arrays applied: For "6th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming" we have presented a treatment of applying a mathematics of arrays to an array programming problem in our submission "Finite difference methods fengshui: alignment through a mathematics of arrays.".

8. A Mathemathics of Arrays evaluated: We evaluate implementations generated from our application of a mathematics of arrays in "Finite difference methods fengshui: alignment through a mathematics of arrays.".

9. Conclusion: We summarize our contributions and results.

CHAPTER 2

Transformations

2.1 C++ metaprogramming

An example of a well treaded field is writing a linear algebra library where we wish to express d=add(add(a,b),c)

typically with add being some infix operator so as to match the usual syntactic convention, where variables a,b,c name vectors. Modeling the domain like that could be considered a direct implementation. Evaluation order in nested functions in C++ are implementation defined or unspecified. However consider that the compiler generates code equivalent to the order determined by:

```
t = add(a,b);
d = add(t,c);
```

yielding one temporary `t`. Naturally one might depart from expressing this as a chain of calls and simply have function add3 such that d = add3(a,b,c) where add3 is defined as

```
  Vector add3(const Vector &a, const Vector &b,
  const Vector &c) {
  Vector tmp(a.size());
  for(int i = 0; i < tmp.size(); i++) {
    tmp[i] = a[i]+b[i]+c[i];
  }
  return tmp;
}
```

eliminating the temporary t but departing from the nice close match to the domain's usual syntactic conventions the direct approach yields. What we wish is to keep something syntactically close to d=a+b+c and to have this transform into a call to add3.

Such optimizations are present in the C++ linear algebra library Eigen[13] which exploits an idiom in C++ called Expression Template (ET)s. This idiom is a form of Template Metaprogramming (TMP) where one builds intermediary expressions at compile time that upon complete template instantiation results in a different evaluation of an expression than the usual semantics of C++ would permit for a direct implementation. In particular implementing the aforementioned vector addition example directly in C++ would yield temporaries, whereas the ET approach results in exactly one looping construct as in add3 while keeping the syntactic nicety of the direct implementation. A minimal example of ETs used in this manner can be found in A.1

In the case of C++ reaching a form closer to add3 from the double looping construct and temporary resulting from add(add(a,b),c) can be important for other reasons than reducing loop predicate overhead or eliminating temporaries as the form will be recognized by the compiler and be subject to an optimization called vectorization.

In brief, vectorization when mentioned in regard to C++ compiler optimizations, means transforming a C++ looping construct over arrays such as:

```
for(i=0;i<N;i++)
  c[i] = a[i]+b[i];
```

into something in C++ akin to:

```
for(i=0;i<N;i+=chunksize)
c[i] = a[i]+b[i];
c[i+1] = a[i+1]+b[i+1];
         ⋮
c[i+chunksize-1] = a[i+chunksize-1]+b[i+chunksize-1];
```

where what is several statements in the loop is expressing one operation that loads several array elements, one that sums them and one that stores the result. For the x86 family of Instuction Set Architecture (ISA)s many provide Single Instruction Multiple Data (SIMD) instructions that realize such operations. Many compilers do not just use such instructions internally but expose them as platform dependent compiler intrinsics or builtins, presented to the programmer as if they were library functions.

The example optimization we presented, only scratches the surface of what is possible. In fact a notion of “smart” ET was presented in [9] along with the promise that they “truly allow for a combination of high performance code with the elegance and maintainability of a domain-specific language”[9, p. C42]. The additional techniques that earn these ETs the qualifier of smart is among others that they facilitate SIMD vectorization, fixed size optimizations, unrolling and blocking techniques relating to cache size optimizations. The considerations for the latter is a vast field [4]. A useful simplification or mental model is that one should attempt to arrange memory access patterns in close proximity to reduce cache misses; all loads and stores are not created equal. Access to memory present in cache is an order of magnitude faster than access to memory that must be fetched from the main memory.

Typically we hope to stay within the confines of a language for portability, meaning adhering to a language standard such as say C++17, while presenting the compiler with constructs that are sure to be recognized and be amenable to special optimizations(such as e.g. vectorization) when possible. On the other hand this can lead to tediously hand-optimized and complex expressions that are difficult and time consuming to grasp the meaning of as they are far removed from idiomatic expressions in the domain. As such the pursuit of closely matching or capturing an algebra of a domain is both a matter of maintainability with respect to the domain and to be able to embed knowledge of various relations that may hold which can give rise to automatically deduced optimizations.

2.2 Beyond the programmer’s reach?

Certain optimizations end up hard coded into the compiler and not within reach for the programmer. An example in Java can be found in *The Java Language Specification: 15.18.1. String Concatenation Operator +* stating[25]:

> ...An implementation may choose to perform conversion and concatenation in one step to avoid creating and then discarding an intermediate String object. To increase the performance of repeated string concatenation, a Java compiler may use the StringBuffer

class or a similar technique to reduce the number of intermediate String objects that are created by evaluation of an expression...

What this amounts to in practice is that the compiler upon seeing String concatenations, such as for example s1+s2+s3+s4, will avoid producing temporaries by instead copying the String objects' contents to a buffer and then upon completion construct a String object with the buffer content. A useful optimization, but the programmer has no easy way of providing such an optimization for their own types.

Let us revise some relevant Java details to write out an example. A String in Java is an immutable object or a value object(has value semantics) encapsulating an array of characters. The expression s1+s2 where s1 and s2 are Strings will produce a new String such that it contains an array of characters being the concatenated arrays of characters in s1 and s2. It is implemented by allocating a new array fitting both character arrays and then writing the contents of each respectively to it and wrapping it in a String object. Each additional concatenation will produce an additional temporary allocation.

A StringBuffer is a wrapper object for the concept of an array you can, among other things, append Strings to the end of. Here, if you allocate a big enough buffer, it will only allocate once as you append your Strings to it. In terms of allocation appending to a StringBuffer is an improvement over the repeated allocation, deallocation and copies that occur with successive String concatenation. In terms of complexity we are looking at, for a concatenation sequence of $s_1 + s_2 + ... + s_n$, transforming the left pseudocode into the right pseudocode below: As we see, with Java String semantics, the amount of

```
$in = s_1, s_2, \ldots, s_n$
$t = s_1$
for $i = 2$ to $n$ do {
    $t = t + s_i$
}
$out = t$
```

```
$in = s_1, s_2, \ldots, s_n$
$t$ = empty buffer
for $i = 1$ to $n$ do {
    $t = s_i$ appended to $t$
}
$out$ = convert $t$ to a string
```

characters copied we will be doing for iteration i is given by the recurrence $copy(i) = length(s_i) + copy(i-1)$ where $copy(2) = length(s_1) + length(s_2)$. The entire amount of copies $all = \sum_{i=2}^{n} copy(i)$. By a simple counting argument we note the characters in s_n is copied 1 time, s_{n-1} is copied 2 times and so on. Fixing the length of the strings to a constant 1 we see we have $O(n^2)$ copies, compared to the copies for appending to a buffer which is clearly $O(n)$. One might say trivial, but we make this point to illustrate that rewriting can also be a matter of improved complexity perhaps also in cases where it's not as obvious such as after other transformations on code.

Of course, and as an aside, there is still the issue that a StringBuffer will in principle still require a copy of its contents to a String during construction of a String from it, in spite of the fact that the StringBuffer might not be used anymore and its content could safely be moved into the String(i.e. making the String array refer to the StringBuffer array).

Lets try to capture the optimization with some general algebraic properties. We do this with some informal set theory. One could imagine taking advantage of specifying the simple algebraic structure monoid which can briefly be summarized as:

Let M be a set and $+$ be a binary operator $+ : M \times M \to M$ such that:

- $+$ is associative: $a + (b + c) = (a + b) + c$
- $+$ has an identity element; there exists[1] $e \in M$ such that for any $m \in M$ we have $m + e = e + m = m$

Clearly strings as we tend to know them from programming along with the usual concatenation function satisfies being a monoid. For example String in Java is M and the concatenation function is $+$ which indeed is associative and has an identity element being the empty String; s+""== ""+s.

We could then extend the monoid structure with an additional set B and operation $* : M \times B \to B$ such that:

- For $b \in B$ and $m_1, m_2 \in M$: $(m_1 + m_2) * b = m_2 * (m_1 * b)$
- For all $b \in B$ the identity element $e \in M$ for $+$ satisfies $e * b = b$

This extension to a monoid is called a right monoid action. In terms of programming the extra structure captures a useful relationship between appending strings to a buffer and string concatenation. What we would want, again in a Java context, is to specify that a String is a monoid(M) and that along with a StringBuffer(B) and its append($*$) forming a right monoid action any time we see String concatenations

$$m_1 + m_2 + \ldots + m_n$$

[1]from a practical programming perspective asserting the existence of an element satisfying something is better replaced with a function picking exactly the element satisfying something

we know of the identity:

$$(m_1 + (m_2 + \ldots + m_n)) * b = (m_2 + \ldots + m_n) * (m_1 * b)$$

and repeated rewrites will yield:

$$m_n * (m_{n-1} * (\ldots * (m_2 * (m_1 * b))) \ldots)$$

Thus our entire rewrite leaving out intermediate steps is:

$$(m_1 + m_2 + \ldots + m_n) * b = m_n * \ldots * m_1 * b$$

where $*$ is right-associative or groups from the right.

Now any time the compiler sees long chains of String concatenations it should in principle be able to replace it with appends to a StringBuffer should we also add an additional function $out : B \to M$ corresponding to the toString method of a StringBuffer and a function to get an empty buffer(a StringBuffer constructor, producing the empty buffer) $b_0 \in B$ such that:

- For all $m \in M : out(m * b_0) = m$

Now carrying on with more information we attempt to apply *out* on both sides of our identity, however with $b = b_0$.

$$(m_1 + m_2 + \ldots + m_n) * b_0 = m_n \ldots * m_1 * b_0$$

and end up with

$$out((m_1 + m_2 + \ldots + m_n) * b_0) = out(m_n * \ldots * m_1 * b_0)$$

which we can finally rewrite as

$$m_1 + m_2 + \ldots + m_n = out(m_n * \ldots * m_1 * b_0)$$

Let's spell it out: Chains of String concatenations are the same as taking the Strings in their order, appending them to an empty StringBuffer and constructing a String from the StringBuffer.

There's a couple of things to note here before we move on. In principle only specifying that a String is a monoid and that along with a StringBuffer with its functions that form a right monoid action as well as the information to extract the String from a StringBuffer a compiler with a rewrite system should be able to match and with successive rewrites eventually get from the left side of our concatenation identity to the right. In that case we do not need to be able to specify identities of varying term lengths.

Ideally we would want a rewrite system to form either our resulting identity given enough specification and match sides of identities of varying term lengths or if we want to avoid too much deduction we need enough expressive power to match left-hand sides of identities such as $m_1 + m_2 + \ldots + m_n$ directly.

Transformations such as these should be within reach of the programmer as opposed to being solely an artifact of the compiler and prescribed as a possibility by the language standard.

For most primitive types specified by various languages the same situation presents itself. In particular for most types in languages that implement some sort of arithmethic, as we have seen, one can expect that the compiler will attempt to rearrange the order of evaluation of an expression according to identities such as for example associativity and distributivity.

While achieving the effect of this certainly is possible in C++ through the use of for instance ETs it is arguably not something that can easily be done by someone without good command of the language.

2.2.1 Haskell rewrite rules

An example of a language, or rather a language extension, enabling the programmer to supply rewrite rules to enable domain-specific optimizations can be found in the Glasgow Haskell Compiler (GHC) and described in [21].

In it a simple example of an optimization opportunity that may arise is one that results from inlining, meaning replacing a function call with its definition with variables substituted for call parameters.

Consider that a programmer has written functions widen(B) and shorten(W) such that the former widens a byte sized value to a word and the latter shortens a word sized value to a byte. As a result of inlining calls the expression shorten(widen(K)) occurs. The programmer has knowledge about the platform that this is value preserving, that is K $\equiv$ shorten(widen(K)). It would then be useful for the programmer to be able to codify this knowledge and enable the compiler to perform such a rewrite. With GHC such a trivial rule can be written as

```
import Data.Word(Word32,Word8)
```

```
{-# RULES "preserving" forall k. shorten (widen k) = k #-}

{-# NOINLINE widen #-}
widen :: Word8 -> Word32
widen b = fromIntegral b

{-# NOINLINE shorten #-}
shorten :: Word32 -> Word8
shorten w = fromIntegral w

r = shorten (widen 32)
```

These operate more or less as follows: Match the left-hand-side expression and replace it with the right-hand-side expression. One specifies a substitution rather than an equality. Certain restrictions apply to the left-hand-side expression [21, p. 2]: It must be a function application and the function cannot be quantified in the rule.

To be certain that the rule is applied we disallow any inlining of widen and shorten as no match would occur in such cases. The interplay between inlining and rewrite rules is also expanded on in [2, p.43-44]. There it is noted that for some expression a+f(b) inlining or Partial Evaluation (PE) may result in f(b) being transformed to 0. In that case an identity element rule on monoids might be applicable for a+0 and thus remove an addition. However inlining on + should it be for instance vector addition might result in either a lost opportunity to apply such a rule or having to apply it in the vector addition loop and then there would be further steps needed to realize the loop does nothing and can be removed. This serves as an important example of that the order of transformations matter.

Although a powerful facility in GHC the rewrite system is not without its perils, as mentioned in [21, p. 2] and [5]. It is worthwhile to reiterate some of them here:

- The obvious one is that the onus of proof that the rewrite rule expresses an identity, i.e. if $t \to t'$ then $t = t'$, and thus that the program still expresses the same computation after applying the rewrite rule lies with the programmer. Should there be a mismatch either the program is wrong, that is it doesn't match the programmer's actual intentions, or the rewrite rule is wrong.
- The more subtle issues lay with a programmer's understanding of equality and the need to preserve equality, that is if $a = b$ then $f(a) = f(b)$.

Consider modeling a set with a list and implementing set union as the concatenation of lists and removal of duplicates. Should the programmer leak access to the representation of the set in some manner, let us say through asList(S) which simply gives you the underlying list implementing the set. Consider then that even though for a set the union operation is commutative very clearly we have the situation asList(**union**(A,B)) $\not\equiv$ asList(**union**(B,A)). So asList(S) did not preserve equality due to being able to make the set abstraction leak its representation. A solution is to specify a fixed ordering for asList(S) such that the order of the representation of the set S never leaks.

- Rules are matched and substitutions performed without any regard as to whether the righthand side expression really is better by any measure than the lefthand side expression. Again a consideration for the programmer.
- If several rules match then an arbitrary of these will be selected to have its substitution performed.
- It is the responsibility of the programmer that the set of rules provided are confluent or terminating.

The last issues requires further explanation.

- That a set of rules R are confluent means that for $r_i \in R$ if there are different rewrite chains $t_1 \xrightarrow{r_i} \ldots \xrightarrow{r_j} t$ and $t_1 \xrightarrow{r_x} \ldots \xrightarrow{r_y} t'$ where no further rewrites can be applied to t or t' then $t = t'$
- That a set of rules R are terminating means that there are no infinite rewrite chains $t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} \ldots$

It is easy to write a rule that will not be terminating in GHC, which is also presented in [21, p. 2]. Consider attemping to add the commutative property of addition:

```
{-# RULES "commutative" forall x y. add x y = add y x #-}

add :: Int -> Int -> Int
add x y = x+y

sum = add 2 8
...
```

This ensures that there is always a rewrite possible and thus the compiler will be stuck rewriting infinitely when it encounters add 2 8.

In regards to confluence and termination Peyton Jones, Tolmach, and Hoare [21, p. 2] notes that there is considerable literature on proving confluence or termination of sets of rewrite rules. In particular citing [1] as a considerable investigation into rewrite rules involving associativity and commutativity. However due to complications stemming from from other rewrites GHC may perform(such as inlining) the literature was not directly applicable for the rewrite system implemented in GHC.

A more pragmatic solution to dealing with confluence and termination may be restrictions in the form of rewrite strategies as mentioned in "Constructs & Concepts: Language Design for Flexibility and Reliability" [2, p.42-45]. These for example limit the scope of where a rule may apply.

2.3 Summary

We have surveyed two approaches to transformations in practical use today. The C++ ET way of creating abstract expressions that are mapped to performant realizations of what they express. Extensions to Haskell that allow one to specify simple identities and rewrites directly on any Haskell expression. Both approaches are manners in which to achieve the same end of being able to write self-optimizing libraries, but require careful considerations to be made by the programmer; implementing well made ET in C++ requires expert level C++ proficiency whereas care must be had in Haskell to not make non-terminating rewrite rules. We are of the opinion that neither technique is suited for a domain expert to codify their knowledge without also having a substantial understanding of either technique.

CHAPTER 3

Arrays

3.1 Introduction

Array abstractions come in the form of direct language support or in the form of library support. We will survey some libraries and more involved array abstractions in another chapter, but first it is pertinent to take a look at what two languages offer us directly.

A common notion for the many different variations of an array type is that it can be indexed through an index space I, that is there is an associated indexing function $get : I \to T$ for the array where T is whatever type the array stores. Typically what is commonly provided by a language or is otherwise immediately implementable is a function taking one-dimensional indices $get : I^1 \to T$ and then multidimensional indexing is realized through a function $flat : I^n \to I^1$ and composition, i.e. $get(\mathit{flat}(i_0, \ldots, i_{n-1}))$. The index space is succintly declared in some manner so let us adopt some notation to describe it.

3.2 Index spaces

Any index space I has a dimensionality, denoted $dim(I)$, that is a positive integer such that $I \subseteq \mathbb{Z} \times \ldots \times \mathbb{Z} = \mathbb{Z}^{dim(I)}$. Sometimes we will indicate the dimensionality of the index space I with a superscripted I^n denoting that $dim(I) = n$. Let $lower(I)$ and $upper(I)$ be two $dim(I) = n$ tuples such that:

$$lower(I) = \langle l_0, l_1, \ldots, l_{n-1} \rangle \text{ and } upper(I) = \langle u_0, u_1, \ldots, u_{n-1} \rangle$$

indicating a lower bound and an upper bound for the index space I in the following manner:

$$I = \{ \langle i_0, i_1, \ldots, i_{n-1} \rangle \mid \forall k : l_k \leq i_k \leq u_k \}$$

The index components i_k themselves will for simplicity's sake be taken to be non-negative integers, i.e. $i_k \in \{ n \in \mathbb{Z} \mid n \geq 0 \}$. They are also subject to the following constraint:

$$i_k \in \{ l_k + j \mid 0 \leq j \leq u_k - l_k \}$$

We can consider counting valid index components, that is:

$$i_k \in \{ \underbrace{l_k + 0}_{1}, \underbrace{l_k + 1}_{2}, \ldots, \underbrace{l_k + (u_k - l_k)}_{s_k} \}$$

from which we can clearly discern[1] the relationship that the number of index components is related to the bounds as follows $s_k = u_k - l_k + 1$ or equivalently $u_k = l_k + s_k - 1$ and $l_k = u_k - s_k + 1$.

An index space can also be entirely described by the concept of a shape, a concept we first became aware of from "A Mathematics of Arrays" [15], and a lower bound. A *shape* for an index space is a tuple:

$$shape(I) = \langle s_0, s_1, \ldots, s_{n-1} \rangle$$

such that $s_k = u_k - l_k + 1$ and are the number of valid i_k components of indices. An index space is completely determined by its shape and lower bound as the upper bound is completely determined by $u_k = l_k + s_k - 1$.

The size of the index space is given by:

$$size(I) = \prod shape(I) = s_0 s_1 \cdots s_{n-1}$$

[1]we skip induction for this trivial matter

3.3 Arrays in C

Let us begin by looking at C's support for multidimensional arrays. One declares arrays in C as int **array**[s]; and arrays of "several dimensions" as int **array**$[s_0][s_1]\cdots[s_{n-1}]$;. The first being an array size s of int and the second being an array size s_0 of array size s_1 of ... of array size s_{n-1} of int. Indexing arrays in an indexing expression is done by supplying an index **array**$[i]$ or for arrays of "several dimensions" **array**$[i_0][i_1]\cdots[i_{n-1}]$.

Arrays In C declared as shown can be considered to have an index space of I^n such that:

$$\begin{aligned} shape(I^n) &= \langle s_0, s_1, \ldots, s_{n-1}\rangle \\ upper(I^n) &= \langle s_0 - 1, \ldots, s_{n-1} - 1\rangle \\ lower(I^n) &= \langle 0, \ldots, 0\rangle \end{aligned}$$

Strictly speaking this is not entirely correct, which is why we must emphasize it is only a useful consideration. The correct interpretation is that arrays in C only take indices of one dimension which is a$[i]$. Indexing with what may appear to be indices of several dimensions such as a$[i_0][i_1]\cdots[i_{d-1}]$ is simply $(\ldots((\mathrm{a}[i_0])[i_1])\cdots)[i_{d-1}]$. That is we are indexing into the array a at index i_0 and then indexing into that array at index i_1 and so on until we are indexing into an array at index i_{d-1} as opposed to indexing directly into a with $\langle i_0, i_1, \ldots, i_{d-1}\rangle$. In either case we have the same amount of ways to supply a complete index to an element, so we consider the repeated indexing as a single bigger index space.

The C11 standard [11, p35 6.2.5 Types: 20] states that arrays describe a contiguously allocated memory region. That means the memory layout for arrays are laid out contigously or in other words they have a flat layout. So for any array with some index space I^n there must exist a function *flat* : $I^n \to I^1$.

Let us create an example of two arrays in C and inspect how the compiler lays them out and treats indexing. We have two read only arrays: three of three dimensions and of size 8 and one of one dimension and of size 8.

```c
const int three[2][2][2] =
{ { {1,2},{3,4} }, { {5,6},{7,8} } };
const int one[8] = {1,2,3,4,5,6,7,8};
```

In case the initialization of three is unfamiliar lets review that. The first (braced) list has lists. These lists correspond to the leftmost dimension.

Then either of these have lists which correspond to the middle dimension. Then within these again we have a list for which the elements correspond to the rightmost dimension. We also create a function indexing that compares the two arrays for some indexing over both arrays.

```
int indexing() {
  if(
  three[0][0][0] == one[0] &&
  three[0][0][1] == one[1] &&
  three[0][1][0] == one[2] &&
  three[0][1][1] == one[3] &&
  three[1][0][0] == one[4] &&
  three[1][0][1] == one[5] &&
  three[1][1][0] == one[6] &&
  three[1][1][1] == one[7] )
  return 5;

  return 10;
}
```

and compile with all optimizations enabled, and inspect the assembly output.

```
indexing:
        mov     eax, 5
        ret
one:
        .long   1
        .long   2
        .long   3
        .long   4
        .long   5
        .long   6
        .long   7
        .long   8
three:
        .long   1
        .long   2
        .long   3
        .long   4
        .long   5
        .long   6
        .long   7
        .long   8
```

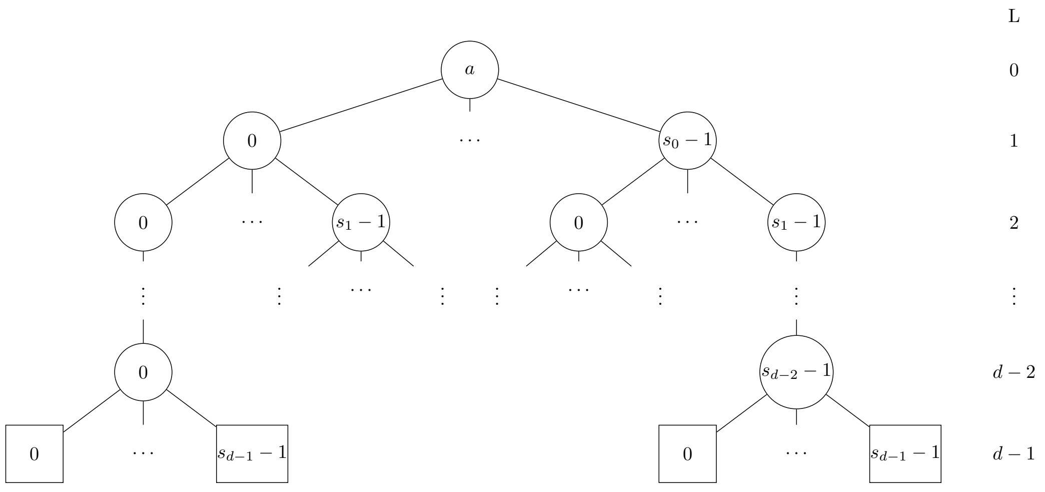
Notice that the compiler has entirely removed any actual access to the arrays for the indexing we perform in the function indexing. Indeed it knows the result will be 5. This indeed also shows us what $flat$ must be:

$$\mathit{flat} = \{\langle 0,0,0\rangle \mapsto 0, \langle 0,0,1\rangle \mapsto 1, \ldots, \langle 1,1,1\rangle \mapsto 7\}$$

In general in C the layout for some "multidimensional array" a is determined by the ordering which can be inferred by the array type. That is for a declaration of a$[s_0][s_1]\ldots[s_{d-1}]$ we have an array sized s_0 of array sized s_1 and so on until array of size s_{d-1} of some type. We can consider this arrangement of arrays within arrays as a tree structure where indexing corresponds to a traversal in the tree.

From the root, at level 0, we have s_0 child nodes to travel to, then from each of those s_1 child nodes to travel to at level 1 and so on until we for practical reasons say we have 1 at level $d-1$ (e.g. staying where we are). We have a tree of d levels with branching factor s_i at level $i < d-1$ as seen in 3.1

Figure 3.1: array indexing as tree traversal



An indexing a$[i_0][i_1]\cdots[i_{d-1}]$ represents a traversal in the tree view of the array declared as a$[s_0][s_1]\cdots[s_{d-1}]$. With this view each leaf is an element in the array and an index $\langle i_0, i_1, \ldots, i_{d-1}\rangle$ corresponds to a traveral in the tree starting at the root and making the choice of going to the node determined by i_0 (a$[i_0]$) and then going to node determined by i_1 (a$[i_0][i_1]$) and so on until we end up in the leaf node determined by i_{d-1} where the element a$[i_0][i_1]\cdots[i_{d-1}]$ is.

Imagine that we first traverse the entire tree depth first by order at level i of child $0, 1, \ldots, s_i - 1$ and for each leaf node we mark it with a number corresponding to amount of leaf nodes seen so far in the traversal. That would give the leaves a marking of 0 to $s_0 s_1 \cdots s_{d-1} - 1$. Given any traversal $\langle i_0, i_1, \ldots, i_{d-1} \rangle$ we would then reach such a marking i in a leaf and this traversal can be seen as retrieving the mapping $\langle i_0, i_1, \ldots, i_{d-1} \rangle \mapsto i$. Constructing i from $\langle i_0, i_1, \ldots, i_{d-1} \rangle$ alone is possible by the property that i corresponds to the number of leaves visited in the depth first traversal such that the paths to them $\langle j_0, \ldots, j_{d-1} \rangle$ were such that $j_k < i_k$ for $0 \leq k \leq d - 1$. We refer to such paths as under exclusions of $\langle i_0, i_1, \ldots, i_{d-1} \rangle$.

For any node at level i let $l(i)$ denote the number of leaves reachable from it. For a node at leaf level, i.e. $i = d - 1$ we have $l(d - 1) = 1$ and for any other node at level $i < d - 1$ we have $l(i) = s_i l(i + 1)$; the branching factor at level i times the number of nodes at the level any of those branches lead to. The latter assumes a full tree at any internal node and there is always a full tree rooted at an internal node as all leaves reachable from an internal node are at the same level. Clearly if some internal node is at level i in the tree then $l(i) = s_i s_{i+1} \cdots s_{d-1}$. Consider a path $\langle i_0, i_1, \ldots, i_{d-1} \rangle$ into the tree. For each i_k from $k = 0$ to $k = d - 1$ we traverse from the node we are in, starting at the root, to a child determined by i_k. Given that we have made a choice i_k we have gone from a node at level k in the tree to a node at level $k+1$. The under exclusions made for the choice i_k can be counted by finding the number of leaves reachable from the choices we didn't make that are under i_k. There are i_k such choices, that is we could have made the choices $0, \ldots, i_k - 1$, and each of those are at level $k + 1$. That means counting the under exclusions caused by choosing i_k is $i_k l(k+1)$. We then see for an entire traversal $\langle i_0, i_1, \ldots, i_{d-1} \rangle$ counting the under exclusions is the sum:

$$\mathit{flat}(i_0, \ldots, i_{d-1}) \sum_{k=0}^{d-1} i_k l(k + 1)$$

where $l(k)$ is defined as:

$$l(k) = s_k s_{k+1} \cdots s_{d-1}$$

We now have a mapping $\langle i_0, \ldots, i_{d-1} \rangle \mapsto i$ and we can by simple arguments verify that the mapping is both injective and surjective and thus a bijection. Given two paths $\langle i_0, \ldots, i_{d-1} \rangle$ and $\langle j_0, \ldots, j_{d-1} \rangle$ differing in one or more components. Can they be mapped to the same i? No, this would imply there are two different paths to a leaf in the tree view of the traversal. By construction we know every leaf has a path to it in the tree. The amount of leaves in the tree is $l(0) = s_0 s_1 \cdots s_{d-1}$. Thus for every $i \in \{0, \ldots, l(0) - 1\}$ there exists a path $\langle i_0, \ldots, i_{d-1}\rangle$ such that $\mathit{flat}(i_0, \ldots, i_{d-1}) = i$.

Let us summarize: For an index $\langle i_0, \ldots, i_{d-1}\rangle$ bounded by the index space defined by $\langle s_0, \ldots, s_{d-1}\rangle$ the flattened ordering is defined by:

$$\begin{aligned}\mathit{flat}(i_0, i_1, \ldots, i_{d-1}) &= \sum_{k=0}^{d-1} i_k l(k+1) \\ &= \sum_{x=0}^{d-1}(i_x \prod_{y=x+1}^{d-1} s_y) = i_0 s_1 \cdots s_{d-1} + \cdots + i_{d-2}s_{d-1} + i_{d-1}\end{aligned}$$

which tends to be rewritten by distributing out the common factors, which is more succintly described with the reccurence relation.

$$\begin{aligned}K_0 &= i_0 \\ K_x &= i_x + s_x K_{x-1}\end{aligned}$$

Such that we can express $flat$ as:

$$\begin{aligned}\mathit{flat}(i_0, i_1, \ldots, i_{d-1}) &= K_{d-1} && (3.1)\\ &= i_{d-1} + s_{d-1}(i_{d-2} + s_{d-2}(\ldots + s_1 i_0)\ldots) && (3.2)\end{aligned}$$

This kind of flattening is called row-major order which groups together indices of equal last index component in the flattened form. That is consider for $\langle s_1, s_2, s_3\rangle = \langle 5, 4, 3\rangle$ we have $\mathit{flat}(2, 1, 3) = 2(4 \cdot 3) + 1(3) + 3 = 30$ and $\mathit{flat}(2, 1, 4) = 2(4 \cdot 3) + 1(3) + 4 = 31$ thus the indices $\langle 2, 1, 3\rangle$ and $\langle 2, 1, 4\rangle$ are next to one another in the flattened view whereas $\mathit{flat}(3, 1, 3) = 3(4\cdot3)+1(3)+3 = 42$ so $\langle 3, 1, 3\rangle$ is placed 12 away from $\langle 2, 1, 3\rangle$. A "reversal" of the computation obtained by re-indexing i_x and s_x in the definition of *flat* by $x \mapsto d - 1 - x$ to obtain flat_r such that we have:

$$\begin{array}{lcllll}\mathit{flat}(i_0, i_1, \ldots, i_{d-1}) & = & i_0 s_1 \cdots s_{d-1} & + \cdots + & i_{d-2}s_{d-1} & + i_{d-1} \\ \mathit{flat}_r(i_0, i_1, \ldots, i_{d-1}) & = & i_{d-1}s_{d-2} \cdots s_0 & + \ldots + & i_1 s_0 & + i_0\end{array}$$

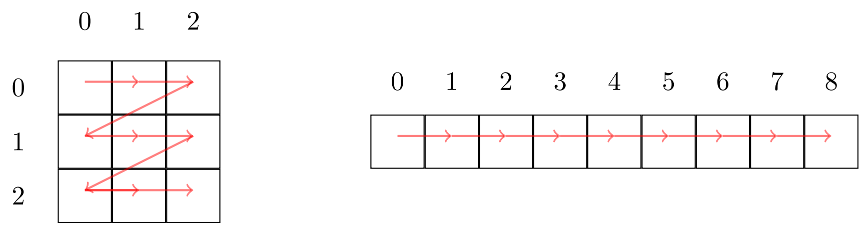
gives us what is called a column-major ordering, which groups together indices of equal first index component(i_0) in the flattened form. The simplest way to think of these two orderings is that row-major ordering is the left-to-right lexicographical ordering of multidimensional indices whereas column-major ordering is the right-to-left or colexicographical ordering of multidimensional indices.

The naming row-major and column-major ordering is an artifact of how either "rows" or "columns" of a two-dimensional array are laid out contigously in a one-dimensional array. If we consider as an example the representation of a two-dimensional array of $\langle i_0, i_1 \rangle$ indices where we consider i_0 to indicate row number and i_1 to indicate column number then a row-major ordering of a $\langle 3, 3 \rangle$ shaped array is given by $\mathit{flat}(i_0, i_1) = i_0 \cdot 3 + i_1$. If we were to iterate through an array laid out with row-major ordering, the default layout for arrays of array in C, with the following C code:

```
int a[3][3] = {...};
for(int i0=0;i0<3;i0++) {
  for(int i1=0;i1<3;i1++) { a[i0][i1] = ...; }
}
```

We would end up with the correspondence seen in figure 3.2 between the "logical" two-dimensional view of the array and its "actual" one-dimensional machine representation. We are visiting indices in lexicographical ordering, from least to greatest, that is $\langle 0,0 \rangle, \langle 0,1 \rangle, \langle 0,2 \rangle, \langle 1,0 \rangle, \langle 1,1 \rangle, \langle 1,2 \rangle, \langle 2,0 \rangle, \langle 2,1 \rangle, \langle 2,2 \rangle$

Figure 3.2: iterating through row-major ordering with column index changing fastest

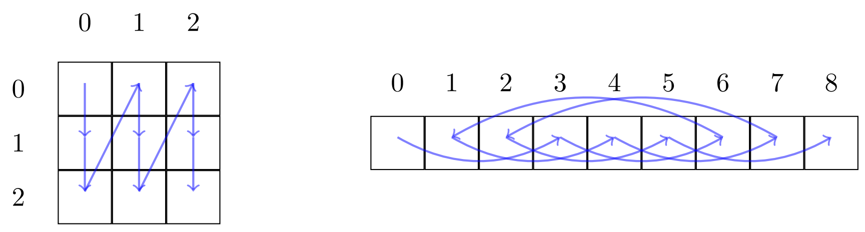


If we ordered the access on the row number first and then the column number, leading to visiting indices in colexicographical order, that is an order of $\langle 0,0 \rangle, \langle 1,0 \rangle, \langle 2,0 \rangle, \langle 0,1 \rangle, \langle 1,1 \rangle, \langle 2,1 \rangle, \langle 0,2 \rangle, \langle 1,2 \rangle, \langle 2,2 \rangle$, as in the following C code:

```
int a[3][3] = {...};
for(int i1=0;i1<3;i1++) {
  for(int i0=0;i0<3;i0++) { a[i0][i1] = ...; }
}
```

We would end up with the correspondence seen in figure 3.3. For the access pattern we end up with here with $\mathit{flat}(i_0, i_1)$ a column-major ordering $\mathit{flat}_r(i_0, i_1) = i_1 \cdot 3 + i_0$ would be a layout that would ensure a linear sweep, as in 3.2, in the array's one-dimensional representation.

Figure 3.3: iterating through row-major ordering with row index changing fastest



3.4 Arrays in Fortran

Let us now turn to Fortran which has traditionally been heavily used in scientific computing, in particular some of its libraries for linear algebra such as Basic Linear Algebra Subprograms (BLAS)/Linear Algebra Package (LAPACK) are said to be some of the most widely used[27, p. 59] scientific computing libraries.

In Fortran the default index space is different from C. Unless otherwise is specified then

$$\mathit{lower}(I^n) = \langle l_0, \ldots, l_r \rangle = \langle 1, \ldots, 1 \rangle$$

Arrays in Fortran are truly multidimensional in the sense that multidimensionality is a part of an array's type whereas in C we only have arrays of one dimension. Still the underlying memory model on a machine is typically flat and unless otherwise is specified then column-major ordering is the default for the underlying memory layout for an array in Fortran.

If we rewrite the C array example in Fortran we will first notice there is no direct initialization possible for multidimensional arrays. That is we cannot supply a list structurally matching the layout of the array that will set the values of the array, except for a one-dimensional array.

For this Fortran supplies the reshape function that will allow us to reinterpret a flat array as a multidimensional one.

```
function indexing() result(r)
  integer :: r
  ! reshape reinterprets an array to have a given shape
  integer, dimension(2,2,2) :: three
  three = reshape ([1,2,3,4,5,6,7,8], [2,2,2])
  integer, dimension(8) :: one = [1,2,3,4,5,6,7,8]
  ! & tells the compiler the statement
  ! continues on the next line
  if (one(1) == three(1,1,1) .and. &
      one(2) == three(2,1,1) .and. &
      one(3) == three(1,2,1) .and. &
      one(4) == three(2,2,1) .and. &
      one(5) == three(1,1,2) .and. &
      one(6) == three(2,1,2) .and. &
      one(7) == three(1,2,2) .and. &
      one(8) == three(2,2,2)) then
        r = 5
   else
      r = 10
   end if
end function indexing
```

which then results in the assembly output that has entirely removed any indexing from the function

```
indexing_:
        mov     eax, 5
        ret
```

indeed confirming the layout. Arrays in Fortran are different from what someone coming from a C/C++ background might be used to. As we already have seen **integer**, **dimension**(2,2,2) :: x declares an array of three dimensions where the index space is $I = \{\langle 1,1,1\rangle, \ldots, \langle 2,2,2\rangle\}$. Typically the number of dimensions is called the rank in Fortran. Array declarations are specifying the shape of an array with its so-called dimension attribute. Fortran allows one to specify starting and ending index by specifying an extent. A declaration of a different array **integer**, **dimension**(0:1,0:1,0:1) :: y will give the index space a C/C++ programmer is used to, that is $I = \{\langle 0,0,0\rangle, \ldots, \langle 1,1,1\rangle\}$, however the shape of y is the same as for x.

Fortran arrays can be operated on elementwise as we are used to from C or C++, e.g. x(1,1,1) gets the element in x at $\langle 1,1,1\rangle$. There are also constructs such as sections that allow selecting across an array

```
integer, dimension(10) :: x
integer, dimension(3) :: y
! initialize x with implicit do loop
x = (/ (i, i=1,10) /)
! initialize  y(1)=x(1)+1 y(2)=x(5)+1 y(3)=x(10)+1
y =  x(1:10:4)+1
! now y = [2,6,10]
x(y) = 0
! now x(2)=0 x(6)=0 x(10)=0
! as x is indexed by y and set to 0
! at each index found in y
```

which are of the form x(start:stop[:stride]) that represent multiple selections of indices and as seen where the [:stride] part is optional and if left out defaults to a stride of one and thus selects a contiguous section of an array. These can then be combined with operators as seen with y=x(1:10:4)+1. An equivalent initialization that doesn't use **reshape** as in our first example, but uses sections is:

```
three(1:2,1,1) = [1,2]
three(1:2,2,1) = [3,4]
three(1:2,1,2) = [5,6]
three(1:2,2,2) = [7,8]
```

We can conclude that the array manipulation primitives in Fortran abstract operations on arrays and arrays themselves to a much greater extent than what we are used to from array primitives in languages such as C/C++.

3.5 layout and performance

Access patterns to a multidimensional array are important when the only abstraction it really offers is a different view of a flattened array. Let us consider some language similar to C or Fortran and looping over a multidimensional array *a* while reading its values for some computation:

```
for $i_0$ = 0 to $s_0 - 1$ {
  for $i_1$ = 0 to $s_1 - 1$ {
        ⋮
    for $i_{d-1}$ = 0 to $s_{d-1} - 1$ { $f(a(i_0, i_1, \ldots, i_{d-1}))$ }...}}
```

For such a looping we could consider $\{i_0\ i_1\ \cdots\ i_{d-1}\}$ a shorthand for the nested loop variables such that the last variable is the fastest changing one as in the nested for loops. We do not care about the computation, simply the access pattern into the underlying flat index space. Let us describe it in terms of I^d with a shape $\mathit{shape}(I^d) = \langle s_0, s_2, \ldots, s_{d-1}\rangle$ and $\mathit{lower}(I^d) = \langle 0, \ldots, 0\rangle$.

Let I^d_{RM} be the ordering of the index space lexicographically(sorting indices left-to-right), that is by picking indices $\langle i_0, i_1, \ldots, i_{d-1}\rangle$ with a loop ordering of $\{i_0\ i_1\ \cdots\ i_{d-1}\}$, and I^d_{CM} be the ordering of the index space colexicographically(sorting indices right-to-left) that is by picking indices $\langle i_0, i_1, \ldots, i_{d-1}\rangle$ with a loop ordering of $\{i_{d-1}\ i_{d-2}\ \cdots\ i_0\}$. We are considering the following two orderings:

$$\begin{aligned}
I^d_{RM} &= \langle\langle 0, \ldots, 0\rangle, \langle 0, \ldots, 1\rangle, \langle 0, \ldots, 2\rangle, \ldots, \langle s_0 - 1, \ldots, s_{d-1} - 1\rangle\rangle \\
I^d_{CM} &= \langle\langle 0, \ldots, 0\rangle, \langle 1, \ldots, 0\rangle, \langle 2, \ldots, 0\rangle, \ldots, \langle s_0 - 1, \ldots, s_{d-1} - 1\rangle\rangle
\end{aligned}$$

Now let $\mathit{flatview}(I^d_X)$ be the ordering where $\mathit{flat} : I^d \to I^1$ has been applied to every element in an ordering X of I^d and equivalently for $\mathit{flatview}_r(I^d_X)$ and $\mathit{flat}_r : I^d \to I^1$. We then have a view of the access pattern we get with two orderings of multidimensional indices in the flattened index space:

$$\begin{aligned}
&\mathit{flat}(i_0, i_1, \ldots, i_{d-1}) = i_0 s_1 \cdots s_{d-1} + \cdots + i_{d-2} s_{d-1} + i_{d-1} \\
&\qquad \mathit{flatview}(I^d_{RM}) = \langle 0, 1, 2, \ldots, (s_0 s_1 \cdots s_{d-1} - 1)\rangle \\
&\qquad \mathit{flatview}(I^d_{CM}) = \langle 0, (1 \cdot s_0 \ldots s_{d-1}), (2 \cdot s_0 \ldots s_{d-1}), \ldots, (s_0 s_1 \cdots s_{d-1} - 1)\rangle
\end{aligned}$$

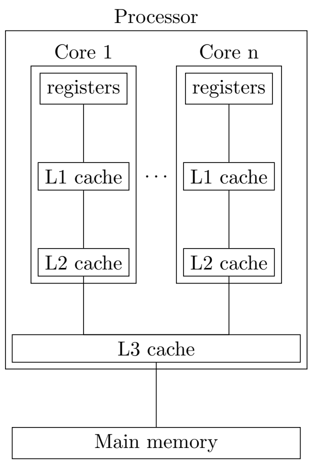
For $\mathit{flatview}_r$ we have the situation reversed:

$$\begin{aligned}
&\mathit{flat}_r(i_0, i_1, \ldots, i_{d-1}) = i_{d-1} s_{d-2} \cdots s_0 + \cdots + i_1 s_0 + i_0 \\
&\qquad \mathit{flatview}_r(I^d_{RM}) = \langle 0, (1 \cdot s_{d-2} \ldots s_0), (2 \cdot s_{d-2} \ldots s_0), \ldots, (s_0 s_1 \cdots s_{d-1} - 1)\rangle \\
&\qquad \mathit{flatview}_r(I^d_{CM}) = \langle 0, 1, 2, \ldots, s_1 s_2 \cdots s_d - 1\rangle
\end{aligned}$$

For a row-major ordering I^d_{RM} access with flat corresponds to a contiguous scan through the flat indices as seen in $\mathit{flatview}(I^d_{RM})$. For a column-major ordering I^d_{CM} access with flat_r as seen in $\mathit{flatview}_r(I^d_{CM})$ corresponds to such a contiguous scan.

In terms of memory access consider one of the orderings that lead to big jumps. For example the column-major ordering where we access $\mathit{flat}(0, \ldots, 0)$ and then access $\mathit{flat}(1, \ldots, 0)$ which we can see in $\mathit{flatview}(I^d_{CM})$ results in a read from the index 0 in the array and then the index $1 \cdot s_0 \ldots s_{d-1}$ in the array. Let's say we are considering a concrete case of an array of shape

Figure 3.4: Memory and cache hierarchies



$\langle 5000, 5000, 5000 \rangle$ storing types of size 8 bytes(i.e. 64 bit words). We read the first location and then read a location $5000 \cdot 5000 \cdot 8\, bytes = 200\, megabytes$ away. This far exceeds most processors cache sizes so we cannot expect that the next read is in the cache from the reading of the former item. The machine this thesis was written on has 6 megabytes. At the time of writing one can expect commodity hardware to have cache sizes in the range of 6-32 megabytes. We will not go into great detail, but it is time to expand on the costs of memory access.

The way most commodity processors work with respect to memory access is that they usually have a hierarchy of cache memories that, when the processor reads or writes to memory, function as intermediaries. An example can be seen in figure 3.4 which is a close match to the organization found in certain processors of x86-64 architecture for which the rest of our discussion of memory and caching will be directly applicable.

The unit of storage for caches or in other words an entry into a cache is called a cache line and a common size for a cache line is 64 bytes [10, 11-4 Vol. 3A]. A cache line can be thought of as a replicated faster view of 64 contigous bytes in main memory. When the processor requests a read from memory the read is first requested from the cache hierarchy. Either the requested memory is present in the cache hierarchy and returned, called a cache hit, or it isn't and a cache line fill is done. A cache line fill means filling a cache line with values from main memory and possibly evicting a cache line to make room for the new cache line. Such an eviction can incur the additional cost of having to propagate writes made to the to be evicted cache line to main memory[10, Vol. 3A 11-5].

Depending on set write policy(we assume write allocate) when the processor requests to write to memory the write can first be requested to be done to the cache hierarchy. Either the requested memory is present in the cache hierarchy, called a write hit, or it isn't and a cache line fill is done followed by a write hit.[10, Vol. 3A 11-5]

An abstract view of cache is as a synchronized and faster view of parts of main memory where you sometimes pay a synchronization penalty(cache line fills and propagation of writes). If we request a read of a memory location we can consider the time it takes to read a cache hit α and the time it takes to read a cache miss β, for which $\alpha < \beta$. We can further differentiate between the cost in time of reads that are in cache by also considering where in the cache hierarchy the read will find the memory it needs.

Now let us go back to our initial issue of iterating through and reading all entries in a multidimensional array. We can devise a very simple cost function for a traversal t in terms of memory read time from an n size array such that the total cost is $hit_t\alpha + miss_t\beta$ where $hit_t + miss_t = n$ and obviously hit_t are cache hits wheras $miss_t$ are cache misses. For a traversal c that results in contiguous memory access there should be a cache line of hits after every miss. For a traveral d that results in discontiguous memory access where jumps in access are always greater in size than the size of a cache line let us assume a worst case of almost only misses, we then have approximately the total cost:

$$\begin{aligned} total_d &\approx miss_d\beta \\ total_c &\approx hit_c\alpha \end{aligned}$$

To get a sense of the costs consider that $\beta \approx 100\alpha$ if we assume misses result in reading from main memory and hits are read from the fastest cache(L1)[12]. In that case we have:

$$total_c \cdot 1/n \approx \alpha < 100\alpha \approx total_d \cdot 1/n$$

Whereas if we change the assumption of hits to be read from the slowest cache(L3)[20, p.22] we have $\beta \approx 2.5\alpha$:

$$total_c \cdot 1/n \approx \alpha < 2.5\alpha \approx total_d \cdot 1/n$$

In either scenario main memory access as a result of cache misses indicates a severe slowdown. To provide some concrete evidence for the effects of cache misses we present, in table 3.1, a small benchmark written in C(run on a i5-8250U Intel CPU) where we perform two simple calculations:

(A) A[i][j] = 2*A[i][j]

(B) A[i][j] = A[i][j]+B[j][i]

yielding different memory access patterns for 10000 × 10000 arrays. A full implementation is provided in B.1 which we summarize here as iterating through the indices and performing the calculations in row-major order(matching the C row-major layout), column-major order and an ordering called tiling. The latter is an approach where instead of sweeping through the entire array along either dimensions in one go we divide the array up into a 1000 × 1000 grid of 10 × 10 tiles each of which we sweep through and thereby minimize jumping to jumping within the tile.

Table 3.1: a 3 run avg. of different orderings in seconds

	row-major	column-major	tiling
A	0.81	0.77	0.28
B	0.04	0.05	0.05

3.6 Strides

Recall that a row-major order flattening $\mathit{flat} : I^d \to I^1$ where $\mathit{shape}(I^d) = \langle s_0, s_1, \ldots, s_{d-1}\rangle$ was given by

$$\mathit{flat}(i_0, i_1, \ldots, i_{d-1}) = i_0 s_1 \cdots s_{d-1} + i_1 s_2 \cdots s_{d-1} + \cdots + i_{d-1}$$

and the size of the array is $s_0 \cdots s_{d-1}$. We define the notion of a stride such that $\mathit{stride}(\langle s_0, s_1, \ldots, s_{d-1}\rangle, 0)$ denotes the size of the entire array, $\mathit{stride}(\langle s_0, s_1, \ldots, s_{d-1}\rangle, 1)$ denotes the size of a 1 increment in the first index, $\mathit{stride}(\langle s_0, s_1, \ldots, s_{d-1}\rangle, 1)$ denotes the size of a 1 increment in the second index and so on. We may define it as follows:

$$\begin{array}{lclcl}
\mathit{stride}(\langle s_0, s_1, \ldots, s_{d-1}\rangle, 0) & & & = & s_0 \cdots s_{d-1} \\
\mathit{stride}(\langle s_0, s_1, \ldots, s_{d-1}\rangle, 1) & = & \mathit{flat}(1, 0, \ldots, 0) & = & s_1 \cdots s_{d-1} \\
\mathit{stride}(\langle s_0, s_1, \ldots, s_{d-1}\rangle, 2) & = & \mathit{flat}(0, 1, \ldots, 0) & = & s_2 \cdots s_{d-1} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\mathit{stride}(\langle s_0, s_1, \ldots, s_{d-1}\rangle, d) & = & \mathit{flat}(0, 0, \ldots, 1) & = & 1
\end{array}$$

For example for an array of shape $\langle s_0, s_1, s_2\rangle = \langle 5, 5, 2\rangle$ giving us indices $\langle i_0, i_1, i_2\rangle$ we have $\mathit{stride}(\langle 5, 5, 2\rangle, 1) = 5 \cdot 2 = 10$ which means that an increment by 1 in i_0 strides by 10 indices in the array(as it is actually layed out) whereas $\mathit{stride}(\langle 5,5,2\rangle, 2) = 2$ means that an increment by 1 in i_1 strides by 2 indices in the array. We can consider this example experssed in C:

```c
int a[5][5][2] = {...};
for(int i0=0;i0<5;i0++) {
  for(int i1=0;i1<5;i1++) {
    for(int i2=0;i2<2;i2++) {
      a[i0][i1][i2] = ...;
    }
  }
}
```

Here our loop ordering ensures that we are always incrementing by 1 in the array's layout. We could say we have the loop ordering i_0, i_1, i_2 with i_2 being the fastest varying index. If we permuted the loop ordering to i_0, i_2, i_1(exchanging the two innermost for-loops) we would be striding by 2 in the fastest varying index. It would mean taking us from an access pattern of $\langle 0,1,2,3,4,5,6,7,9,10\ldots\rangle$ to $\langle 0,2,4,6,8,1,3,5,7,9,10\ldots\rangle$

To summarize our definition of stride, given a shape $\langle s_0, s_2, \ldots, s_{d-1}\rangle$ and k such that $0 \leq k \leq d$:

$$\mathit{stride}(\langle s_0, s_1, \ldots, s_{d-1}\rangle, k) = s_k \cdots s_{d-1} = \prod_{x=k}^{d-1} s_x$$

and for $1 \leq k \leq d$ we have:

$$\mathit{stride}(\langle s_0, s_1, \ldots, s_{d-1}\rangle, k) = \mathit{flat}(i_0, \ldots, i_{k-1}, \ldots, i_{d-1})$$

where $i_x = 1$ when $x = k-1$ and otherwhise $i_x = 0$.

Seeing as our defintion of *flat* is:

$$\mathit{flat}(i_0, \ldots, i_{d-1}) = \sum_{x=0}^{d-1}(i_x \prod_{y=x+1}^{d-1} s_y)$$

We may equivalently define it in terms of $\mathit{stride}(\langle s_0, \ldots, s_{d-1}, k\rangle)$ as:

$$\mathit{flat}(i_0, \ldots, i_{d-1}) = \sum_{x=0}^{d-1}(i_x \cdot \mathit{stride}(\langle s_0, \ldots, s_{d-1}\rangle, x+1))$$

3.7 Selecting orderings

We have defined *flat* : $I^d \to I^1$ for *shape*$(I^d) = \langle s_0, s_1, \ldots, s_{d-1}\rangle$ as:

$$\mathit{flat}(i_0, \ldots, i_{d-1}) = \sum_{x=0}^{d-1}(i_x \prod_{y=x+1}^{d-1} s_y)$$

Where indices $\langle i_0, \ldots, i_{d-1}\rangle \in I^d$ are mapped to $i \in I^1$ and we say elements of I^d are grouped together in I^1 by the ordering on dimensions, from last to first, of $\langle d-1, d-2, \ldots, 0\rangle$. That means if we take all indices $\langle i_0, \ldots, i_{d-1}\rangle \in I^d$ and arrange them in a sequence sorted lexicographically we get one where $\langle i_0, \ldots, i_{d-1}\rangle \in I^d$ is placed in the sequence corresponding to where it is mapped to in $n \in I^d$ by *flat*. That is $\langle i_0, \ldots, i_{d-1}\rangle_n \Leftrightarrow \mathit{flat}(i_0, \ldots, i_{d-1}) = n$. To get a different ordering we may use a bijective function $p : \{0, \ldots, d-1\} \to \{0, \ldots, d-1\}$ or simpler put a permutation to change the ordering to $\langle p(d-1), p(d-2), \ldots, p(0)\rangle$ and get:

$$\mathit{flat}_p(i_0, \ldots, i_{d-1}) = \mathit{flat}(i_{p(0)}, \ldots, i_{p(d-1)})$$

where the permutation must also be applied to the shape used in *flat*

$$p'(\mathit{shape}(I^d)) = \langle s_{p(0)}, s_{p(1)}, \ldots, s_{p(d-1)}\rangle$$

For clarity a complete definition would then be:

$$\mathit{flat}_p(i_0, \ldots, i_{d-1}) = \sum_{x=0}^{d-1}(i_{p(x)} \prod_{y=x+1}^{d-1} s_{p(y)})$$

We may in fact regard *flat* as always being subject to such a permutation, but when the permutation is the identity permutation $id(x) = x$ then that doesn't warrant its mention as $\mathit{flat}_{id} = \mathit{flat}$.

For instance to get a column-major ordering from our row-major ordering by default *flat* that corresponds to specifying the permutation $r(x) = d-1-x$ which reverses the order and using the function flat_r.

Naturally *stride* must change according to the permutation:

$$\mathit{stride}_p(\langle s_0, s_1, \ldots, s_{d-1}\rangle, k) = \mathit{stride}(\langle s_{p(0)}, s_{p(1)}, \ldots, s_{p(d-1)}\rangle, k) = \prod_{x=k}^{d-1} s_{p(x)}$$

Similarly as with *flat* we may regard it as always being subject to a permutation $\mathit{stride}_{id} = \mathit{stride}$ that only warrants mentioning when differing from the identity permutation.

We may define flat_p in terms of stride_p almost as before:

$$\mathit{flat}_p(i_0, \ldots, i_{d-1}) = \sum_{x=0}^{d-1} (i_{p(x)} \cdot \mathit{stride}_p(\langle s_0, \ldots, s_{d-1} \rangle, x+1))$$

3.7.1 Ordering selection example

The permutation concept can be confusing so let us take a moment to make an example. Consider that we want to index a 4-dimensional array of shape $\langle s_0, s_1, s_2, s_3 \rangle$. We then have our indexing of:

$$\mathit{flat}_{id}(i_0, i_1, i_2, i_3) = i_0 s_1 s_2 s_3 + i_1 s_2 s_3 + i_2 s_3 + i_3$$

Where the identity permutation *id* can be described by $id \equiv \langle 0, 1, 2, 3 \rangle$ as short for $id = \{0 \mapsto 0, \ldots, 3 \mapsto 3\}$. Selecting a different permutation such as for example $p \equiv \langle 2, 3, 0, 1 \rangle$ that is $p = \{0 \mapsto 2, 1 \mapsto 3, 2 \mapsto 0, 3 \mapsto 1\}$ will give us the indexing:

$$\mathit{flat}_p(i_0, i_1, i_2, i_3) = i_2 s_3 s_0 s_1 + i_3 s_0 s_1 + i_0 s_1 + i_1$$

Whereas before the indices $\langle i_0, i_1, i_2, i_3 \rangle$ were ordered by $\langle 3, 2, 1, 0 \rangle$ they are now ordered by $\langle 1, 0, 3, 2 \rangle$.

3.8 Summary and insights

We have studied the abstraction of multidimensional arrays in C and Fortran, how they map to a flat memory layout and certain implications of different access patterns and layouts with respect to hardware.

We conclude by introducing a function γ [15, p.11] equivalent to the one defined in 3.1 (for $0 < d$) and summarize the mapping of multidimensional indices to one-dimensional indices.

Let $\langle i_0, i_1, \ldots, i_{d-1} \rangle \in I^d$ where $\mathit{lower}(I^d) = \langle 0, \ldots, 0 \rangle$ and $\mathit{shape}(I^d) = \langle s_0, s_1, \ldots, s_{d-1} \rangle$.

Then $\gamma : I^d \times \{shape(I^d)\} \rightarrow I^1$ is defined where $shape(I^1) = \langle s_0, s_1, \ldots, s_{d-1}\rangle$ by:

$$\begin{gathered}\gamma(\langle\rangle, \langle\rangle) = 0 \\ \gamma(\langle i_0\rangle, \langle s_0\rangle) = i_0 \\ \gamma(\langle i_0, \ldots, i_{d-2}, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-2}, s_{d-1}\rangle) = \\ i_{d-1} + s_{d-1} \cdot \gamma(\langle i_0, \ldots, i_{d-2}\rangle, \langle s_0, \ldots, s_{d-2}\rangle)\end{gathered}$$

Note the addition of an empty index and an empty shape for $\gamma(\langle\rangle, \langle\rangle)$. This is an artifact of how this function is used where it was introduced. A scalar value such as $1, 2, 3, \ldots$ can be seen as an array of zero dimensions with a single value index. For $0 < d$ we have:

$$\gamma(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) = \mathit{flat}(i_0, \ldots, i_{d-1})$$

Remark: In an actual implementation an "unfolding" version of γ may be more desireable the shown "folding" one:

$$\begin{gathered}\gamma(\langle\rangle, \langle\rangle) = 0 \\ \gamma(\langle i_0, i_1, \ldots, i_{d-1}\rangle, \langle s_0, s_1, \ldots, s_{d-1}\rangle) = \\ \mathit{unfold}(\langle i_1, \ldots, i_{d-1}\rangle, \langle s_1, \ldots, s_{d-1}\rangle, i_0)\end{gathered}$$

$$\begin{gathered}\mathit{unfold}(\langle i_0, i_1, \ldots, i_{d-1}\rangle, \langle s_0, s_1 \ldots, s_{d-1}\rangle, r) = \\ \mathit{unfold}(\langle i_1, \ldots, i_{d-1}\rangle, \langle s_1, \ldots, s_{d-1}\rangle, i_0 + s_0 \cdot r) \\ \mathit{unfold}(\langle\rangle, \langle\rangle, r) = r\end{gathered}$$

In γ the order given in 3.3 yields a row-major ordering whereas in 3.4 yields a column-major ordering. Regardless of order, $\mathit{stride}(s, k)$ with s ordered as used in γ yields the stride in the flat array.

$$\gamma(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) \tag{3.3}$$

$$\gamma(\langle i_{d-1}, \ldots, i_0\rangle, \langle s_{d-1}, \ldots, s_0\rangle) \tag{3.4}$$

In general we can select a different ordering with γ_p where p is a permutation $p : \{0, \ldots, d-1\} \rightarrow \{0, \ldots, d-1\}$ such that:

$$\gamma_p(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) = \gamma(\langle i_{p(0)}, \ldots, i_{p(d-1)}\rangle, \langle s_{p(0)}, \ldots, s_{p(d-1)}\rangle)$$

In fact we may always regard γ as using a permutation namely the identity permutation $id(x) = x$ such that $\gamma_{id} = \gamma$.

We note that for $0 < d$ we have:

$$\begin{aligned}\gamma_p(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) &= \mathit{flat}_p(i_0, \ldots, i_{d-1}) \\ &= \sum_{x=0}^{d-1}(i_{p(x)} \prod_{y=x+1}^{d-1} s_{p(y)})\end{aligned}$$

In fact we may lift the restriction that $0 < d$ for the latter equality to that $0 \leq d$ as we have:

$$\gamma(\langle\rangle, \langle\rangle) = 0 = \sum_{x=0}^{0-1}(i_{p(x)} \prod_{y=x+1}^{0-1} s_{p(y)})$$

That is we have an empty sum and so it fits with the definition of γ and we obtain:

$$\gamma_p(\langle i_0, \ldots, i_{d-1)}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) = \sum_{x=0}^{d-1}(i_{p(x)} \prod_{y=x+1}^{d-1} s_{p(y)})$$

3.8.1 Relating gamma and strides

Given that for $0 < d$ we have $\gamma(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) = \mathit{flat}(i_0, \ldots, i_{d-1})$ Recall that *flat* could be defined in terms of *stride* as:

$$\mathit{flat}(i_0, \ldots, i_{d-1}) = \sum_{x=0}^{d-1}(i_x \cdot \mathit{stride}(\langle s_0, \ldots, s_{d-1}\rangle, x+1))$$

and thus similarly we may define γ in terms of *stride* as:

$$\gamma(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) = \sum_{x=0}^{d-1}(i_x \cdot \mathit{stride}(\langle s_0, \ldots, s_{d-1}\rangle, x+1))$$

similarly for a permutation p as before we may derive

$$\gamma_p(\langle i_0, \ldots, i_{d-1)}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) = \sum_{x=0}^{d-1}(i_{p(x)} \cdot \mathit{stride}_p(\langle s_0, \ldots, s_{d-1}\rangle, x+1))$$

3.8.2 Sections of an array

Consider defining where indices start(or are mapped to) in the flat index space when they are of the form $\langle i_0, \ldots, i_{k-1}, i_k, \ldots, i_{d-1}\rangle$ where $0 \leq k \leq d$, such that we consider $i_0, \ldots, i_{k-1}$ as fixed(or as constants) and $i_k, \ldots, i_{d-1}$ as varying(or as variables).

We first note:

$$\begin{aligned}
\gamma(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) &= \sum_{x=0}^{d-1}(i_x \prod_{y=x+1}^{d-1} s_y) \\
&= \sum_{x=0}^{k-1}(i_x \prod_{y=x+1}^{d-1} s_y) + \sum_{x=k}^{d-1}(i_x \prod_{y=x+1}^{d-1} s_y) \\
&= \sum_{x=0}^{k-1}(i_x \prod_{y=x+1}^{k-1} s_y)\prod_{y=k}^{d-1} s_y + \sum_{x=k}^{d-1}(i_x \prod_{y=x+1}^{d-1} s_y)
\end{aligned}$$

We can then define *start* and relate it to the greater γ expression we just derived and also our definition of *stride*:

$$\begin{aligned}
\mathit{start}(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle, k) &= \gamma(\langle i_0, \ldots, i_{k-1}\rangle, \langle s_0, \ldots, s_{k-1}\rangle) \\
&= \sum_{x=0}^{k-1}(i_x \prod_{y=x+1}^{k-1} s_y)
\end{aligned}$$

$$\mathit{stride}(\langle s_0, \ldots, s_{d-1}\rangle, k) = \prod_{y=k}^{d-1} s_y$$

$$\gamma(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle) = \sum_{x=0}^{k-1}(i_x \prod_{y=x+1}^{k-1} s_y)\prod_{y=k}^{d-1} s_y + \sum_{x=k}^{d-1}(i_x \prod_{y=x+1}^{d-1} s_y)$$

Thus we see we can express γ with the partitioning of fixed $i_{x<k}$ and varying $i_{x>k}$ indices where $0 \leq k \leq d$ as:

$$\begin{gathered}
\gamma(\langle i_0, \ldots, i_{k-1}, i_k, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{k-1}, s_k, \ldots, s_{d-1}\rangle\rangle) = \\
\mathit{start}(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle, k)\mathit{stride}(\langle s_0, \ldots, s_{d-1}\rangle, k) \\
+ \gamma(\langle i_k, \ldots, i_{d-1}\rangle, \langle s_k, \ldots, s_{d-1}\rangle)
\end{gathered}$$

The practical benefit of this result, which is also obtained in [17], is that for indices:

$$\langle \underbrace{i_0, \ldots i_{k-1},}_{\text{fixed}} \underbrace{i_k \ldots, i_{d-1}}_{\mathit{varying}} \rangle$$

we may compute the fixed part, corresponding to

$$\mathit{fixed} = \mathit{start}(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle, k)\mathit{stride}(\langle s_0, \ldots, s_{d-1}\rangle, k)$$

and compute all the complete indices we would get from adding the varying part, which are constrained by the shape $\langle s_k, \ldots, s_{d-1}\rangle$, by the ordering the parameters to the varying part as listed:

$$\mathit{indices} = \{\mathit{fixed} + \mathit{varying} \mid \mathit{varying} \in \{0, 1, \ldots, (s_k \cdots s_{d-1} - 1)\}\}$$

With this kind of relationship we are able to express subarrays; the fixed part selects a subarray and then the varying part offsets into it. Notice that by having no fixed part the flat indices become the range from 0 to $s_k \cdots s_{d-1} - 1$ and that these are the flat indices for all multidimensional indices $\langle i_0, \ldots, i_{d-1}\rangle$ to γ with a fixed shape of $\langle s_0, \ldots, s_{d-1}\rangle$. This can be regarded as a proof that γ with a fixed shape is a bijection between the set of multidimensional indices denoted by the fixed shape and the set of flat indices(from 0 to $s_0 \cdots s_{d-1} - 1$) denoted by the product of the fixed shape.

3.8.2.1 Sections and permutations

It can be shown that the central relationship here, that is:

$$\begin{gathered}\gamma(\langle i_0, \ldots, i_{k-1}, i_k, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{k-1}, s_k, \ldots, s_{d-1}\rangle\rangle) = \\ \mathit{start}(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle, k)\mathit{stride}(\langle s_0, \ldots, s_{d-1}\rangle, k) \\ + \gamma(\langle i_k, \ldots, i_{d-1}\rangle, \langle s_k, \ldots, s_{d-1}\rangle)\end{gathered}$$

Still holds given a permutation p and a definition of start_p such that:

$$\begin{gathered}\mathit{start}_p(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle, k) = \\ \mathit{start}(\langle i_{p(0)}, \ldots, i_{p(d-1)}\rangle, \langle s_{p(0)}, \ldots, s_{p(d-1)}\rangle, k)\end{gathered}$$

We then obtain the following:

$$\begin{gathered}\gamma_p(\langle i_0, \ldots, i_{k-1}, i_k, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{k-1}, s_k, \ldots, s_{d-1}\rangle\rangle) = \\ \mathit{start}_p(\langle i_0, \ldots, i_{d-1}\rangle, \langle s_0, \ldots, s_{d-1}\rangle, k)\mathit{stride}_p(\langle s_0, \ldots, s_{d-1}\rangle, k) \\ + \gamma_p(\langle i_k, \ldots, i_{d-1}\rangle, \langle s_k, \ldots, s_{d-1}\rangle)\end{gathered}$$

CHAPTER 4

A Mathemathics of Arrays

4.1 Introduction

In 1988 Lenore Mullin published her phd. thesis A Mathemathics of Arrays (MOA) [15] where she builds a theory of arrays as algebraic objects. She cites Kenneth E. Iverson, the creator of APL, as having a profound impact on her research.

In this chapter we will present select ideas from her thesis and her view of arrays. Certain concepts will already be familiar by now such as the idea of a shape and how it generates an index space. Mullin's style of specification is in gradually extending operations; she may introduce an operation for a certain kind of array and then alter that operation in general for arrays of another kind.

We proceed in the same fashion as Mullin and gradually build her MOA vocabulary albeit simplified and with slight deviations to avoid certain complexities while attempting to faithfully convey her ideas.

Remark: We have departed from Mullin's use of equivalences in describing arrays of certain classifications. These equivalences however should be apparent and we will give arrays of certain classifications names to reflect that.

4.2 Preliminaries: Vectors

As the term vector will be used in this chapter the terminology and notation should be clarified. We are not considering vectors in the typical mathematical sense, but rather one that is more familiar to a programmer; an ordered, by index, collection of elements. Other names commonly used is a list or array. We pick vector as it is the terminology already in use by Mullin(as well as a popular choice in programming).

We can consider a vector $\vec{x}$ of some non-negative integer size $\tau\,\vec{x} \in \mathbb{Z}^{\geq}$[1] possibly containing elements from some set E, referred to as vectors of E, as the pair $\vec{x} = (f, E)$ such that:

- For $\tau\,\vec{x} = 0$ the vector $\vec{x}$ is called an empty vector and $f = \{\}$.
- For $\tau\,\vec{x} \geq 0$ the vector $\vec{x}$ is called a non-empty vector and

  $$f : \{0, \dots, \tau\,\vec{x} - 1\} \to E$$

  is such that f maps an index $i \in \{0, \dots, \tau\,\vec{x} - 1\}$ to an element $e \in E$

  $$f = \{0 \mapsto e_0, \dots, \tau\,\vec{x} - 1 \mapsto e_{\tau\,\vec{x}-1} | e_i \in E\}$$

We use the notation $\vec{x} = \langle\rangle$ for the empty vector, i.e. when $\tau\,\vec{x} = 0$ and when $\tau\,\vec{x} \geq 0$ the notation $\vec{x} = \langle x_0, \dots, x_{\tau\,\vec{x}-1}\rangle$ where this may still be the empty vector, but if it is not an empty vector we may use it as a function and index elements $\vec{x}[i] = f(i)$. When mentioning several vectors we implicitly assume they contain elements from the same set E. We regard the the set of all vectors given some set of elements E to be:

$$\boldsymbol{V}(E) = \{\vec{x} \mid \tau\,\vec{x} \in \mathbb{Z}^{\geq} \wedge \text{for all } i \text{ s.t. } 0 \leq i < \tau\,\vec{x} : \vec{x}[i] \in E\}$$

As we may occasionally wish to use a nesting of vectors, e.g. $\boldsymbol{V}(\boldsymbol{V}(\boldsymbol{V}(E)))$ for vectors of vectors of vectors of E, we will use the notation $\boldsymbol{V}^1(E) = \boldsymbol{V}(E)$ and $\boldsymbol{V}^n(E) = \boldsymbol{V}(\boldsymbol{V}^{(n-1)}(E))$. This means that $\boldsymbol{V}^3(E) = \boldsymbol{V}(\boldsymbol{V}(\boldsymbol{V}(E)))$. To be able to refer to the set of all nested vectors with elements from E we introduce:

$$\boldsymbol{V}^*(E) = \bigcup_{i=1}^{\infty} = \boldsymbol{V}^i(E)$$

[1] $\mathbb{Z}^{\geq} = \{0, 1, 2, \dots\}$

4.2.1 Vector operations: Iota, Take, Drop and Concatenation

For some examples we will use the somewhat useful function $\iota : \mathbb{Z}_{\geq 0} \rightarrow \boldsymbol{V}(\mathbb{Z}_{\geq 0})$:

$$\iota\, n = \langle x_0, \ldots, x_{n-1} \rangle \text{ where } x_i = i$$

That is $\iota\, 0 = \langle\rangle$ and $\iota\, 5 = \langle 0, 1, 2, 3, 4 \rangle$. Another manner of specifying this is by listing the cases. For $n = 0$ we get an empty vector, otherwise we get a vector with elements equal to their index:

$$\begin{array}{ll} \iota\, n = \langle\rangle & \text{when } n = 0 \\ (\iota\, n)[i] = i & \text{otherwise, for } i \text{ s.t. } 0 \leq i < n \end{array}$$

Given some vector $\vec{x} = \langle x_0, \ldots, x_{\tau\,\vec{x}-1} \rangle$ we define the vector $i \uparrow \vec{x}$ produced by "taking" the i first elements from $\vec{x}$ and placing them in order in a vector. We also define the vector $i \downarrow \vec{x}$ produced by "dropping" the i first elements of $\vec{x}$ and placing the remaining in order in a vector. These functions are of the kind $\uparrow : \mathbb{Z}_{\geq 0} \times \boldsymbol{V}(E) \rightarrow \boldsymbol{V}(E)$ and $\downarrow : \mathbb{Z}_{\geq 0} \times \boldsymbol{V}(E) \rightarrow \boldsymbol{V}(E)$.

First we define $\mathit{min} : \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0} \rightarrow \mathbb{Z}_{\geq 0}$ in the following manner:

$$\mathit{min}(a, b) = \begin{cases} a & \text{when } a \leq b \\ b & \text{otherwise} \end{cases}$$

We then define the vectors resulting from taking or dropping as:

$$\tau\,(i \uparrow \vec{x}) = \mathit{min}(i, \tau\,\vec{x})$$
$$\tau\,(i \downarrow \vec{x}) = \tau\,\vec{x} - \mathit{min}(i, \tau\,\vec{x})$$

The elements of these vectors are defined by:

$$\vec{x} = \langle \underbrace{x_0, \ldots, x_{i-1}}_{i\uparrow\vec{x}}, \underbrace{x_i, \ldots, x_{\tau\,\vec{x}-1}}_{i\downarrow\vec{x}} \rangle$$

Or more precisely:

$$\begin{array}{lll} (i \uparrow \vec{x}) = \langle\rangle & \text{when} & \tau\,(i \uparrow \vec{x}) = 0 \\ (i \uparrow \vec{x})[j] = \vec{x}[j] & \text{otherwise, where} & 0 \leq j < \tau\,(i \uparrow \vec{x}) \\ (i \downarrow \vec{x}) = \langle\rangle & \text{when} & \tau\,(i \downarrow \vec{x}) = 0 \\ (i \downarrow \vec{x})[j] = \vec{x}[j+i] & \text{otherwise, where} & 0 \leq j < \tau\,(i \downarrow \vec{x}) \end{array}$$

A final and familiar operation on vectors is concatenation. Let $\vec{x}$ and $\vec{y}$ be vectors. We define their concatenation $\vec{x} \triangleright \vec{y}$, where we think of concatenation as a function $\triangleright : \boldsymbol{V}(E) \times \boldsymbol{V}(E) \to \boldsymbol{V}(E)$, as a vector such that:

$$\tau\,(\vec{x} \triangleright \vec{y}) = \tau\,\vec{x} + \tau\,\vec{y}$$

for $\vec{x} = \langle\rangle$ or $\vec{y} = \langle\rangle$:

$$\begin{aligned} \langle\rangle \triangleright \vec{y} &= \vec{y} \\ \vec{x} \triangleright \langle\rangle &= \vec{x} \end{aligned}$$

otherwise for $\vec{x} \neq \langle\rangle$ and $\vec{y} \neq \langle\rangle$:

$$(\vec{x} \triangleright \vec{y})[i] = \begin{cases} \vec{x}[i] & \text{when} \quad 0 \leq i < \tau\,\vec{x} \\ \vec{y}[i - \tau\,\vec{x}] & \text{when} \quad \tau\,\vec{x} \leq i < \tau\,\vec{x} + \tau\,\vec{y} \end{cases}$$

Note a few trivially derivable identities from the definition of take($\uparrow$), drop($\downarrow$) and concatenation assuming $0 \leq k \leq \tau\,\vec{x}$ then:

$$\begin{aligned} k \uparrow (\vec{x} \triangleright \vec{y}) &= k \uparrow \vec{x} \\ k \downarrow (\vec{x} \triangleright \vec{y}) &= (k \downarrow \vec{x}) \triangleright \vec{y} \\ (k \uparrow \vec{x}) \triangleright (k \downarrow \vec{x}) &= \vec{x} \end{aligned}$$

With these identities we may for example obtain a short proof of the associativity of concatenation:

$$\begin{aligned} \tau\,\vec{x} \uparrow ((\vec{x} \triangleright \vec{y}) \triangleright \vec{z}) &= \tau\,\vec{x} \uparrow (\vec{x} \triangleright \vec{y}) = \vec{x} \\ \tau\,\vec{x} \downarrow ((\vec{x} \triangleright \vec{y}) \triangleright \vec{z}) &= (\tau\,\vec{x} \downarrow (\vec{x} \triangleright \vec{y})) \triangleright \vec{z} = \vec{y} \triangleright \vec{z} \\ (\vec{x} \triangleright \vec{y}) \triangleright \vec{z} &= (\tau\,\vec{x} \uparrow ((\vec{x} \triangleright \vec{y}) \triangleright \vec{z})) \triangleright (\tau\,\vec{x} \downarrow ((\vec{x} \triangleright \vec{y}) \triangleright \vec{z})) = \vec{x} \triangleright (\vec{y} \triangleright \vec{z}) \end{aligned}$$

4.3 MOA fundamentals

Mullin describes an array, denoted by ξ, as having dimensions which is a non-negative integer $\delta\,\xi \in \mathbb{Z}_{\geq 0}$. Without further qualifications ξ means an array of unknown dimensions. Aside from an explicit statement $\delta\,\xi = n$ we may superscript the array with its dimensions ξ^n to denote the same.

Whatever the dimensionality is for a particular array ξ it has a shape denoted by $\rho\,\xi = \langle s_0, \ldots, s_{\delta\,\xi - 1}\rangle$, a possibly empty vector $\langle\rangle$, of as many non-negative integers as the array's dimensions, that is $\rho\,\xi \in \boldsymbol{V}(\mathbb{Z}_{\geq 0})$ and $\tau\,(\rho\,\xi) = \delta\,\xi$. We classify arrays according to its dimensionality as follows:

- A 0-dimensional array ξ^0 is a scalar array and its shape is an empty vector:
$$\rho\,\xi^0 = \langle\rangle$$
- A 1-dimensional array ξ^1 is a vector array and its shape is a vector with a single element:
$$\rho\,\xi^1 = \langle s_0\rangle$$
- A $\delta\,\xi$-dimensional array $\xi^{\delta\,\xi}$ where $1 < \delta\,\xi$ is a multidimensional array and its shape is a non-empty vector of non-negative integers:
$$\rho\,\xi^{\delta\,\xi} = \langle s_0, \ldots, s_{(\delta\,\xi)-1}\rangle$$

In either case the shape defines what valid indices for an array are and by extension how many indices there are. For an array with shape $\rho\,\xi = \langle s_0, \ldots, s_{\delta\,\xi-1}\rangle$ a valid index $\vec{i} = \langle i_0, \ldots, i_{\delta\,\xi-1}\rangle$ is one such that $\tau\,\vec{i} = \delta\,\xi$ and:

$$0 \leq^\star \vec{i} <^\star \rho\,\xi = (0 \leq^\star \vec{i}) \wedge (\vec{i} <^\star \rho\,\xi)$$

Where this shorthand is defined given some relation $x\ R\ y$ as:

$$x\ R^\star\ \langle y_0, \ldots, y_{n-1}\rangle = \bigwedge_{i=0}^{n-1} x\ R\ y_i$$
$$\langle x_0, \ldots, x_{n-1}\rangle\ R^\star\ \langle y_0, \ldots, y_{n-1}\rangle = \bigwedge_{i=0}^{n-1} x_i\ R\ y_i$$

Note that we consider the empty logical conjunction here(i.e. when $n = 0$) to hold. This has the implication that for a scalar array, having the shape $\rho\,\xi^0 = \langle\rangle$, the empty vector is a valid index as $0 \leq^\star \langle\rangle <^\star \langle\rangle$ holds.

A peculiarity following from the constraints of the shape of an array is that if there is any element in the shape of an array that is zero then no index will satisfy the constraints and be a valid index for an array. Such arrays are considered unique empty arrays. For example for an array of shape $\langle s_0, s_1, s_2\rangle$ there are 7 ways to distribute 0 and thus 7 different unique empty arrays for arrays of dimensionality 3, we will use the notation $\Theta^{\delta\,\xi}$ to denote such an empty array.

An array can be regarded as an extension of the vector. That is we can regard an array as the pair $\xi = (f_{\rho\,\xi}, E)$ where $f_{\rho\,\xi}$ maps an index to an element of some set E or the function:

$$f_{\rho\,\xi} : \{\vec{i} \in \boldsymbol{V}(\mathbb{Z}_{\geq 0}) \mid \tau\,\vec{i} = \delta\,\xi \wedge 0 \leq^\star \vec{i} <^\star \rho\,\xi\} \to E$$

The total amount of elements there are in an array correspond to how many valid indices there are for the array. We extend the notation we had for vectors to arrays such that:

$$\tau\,\xi = |\{\vec{i} \in \boldsymbol{V}(\mathbb{Z}_{\geq 0}) \mid \tau\,\vec{i} = \delta\,\xi \wedge 0 \leq^{\star} \vec{i} <^{\star} \rho\,\xi\}|$$

which we discern is:

$$\tau\,\xi = \pi\,(\rho\,\xi) = \pi\,\langle s_0, \ldots, s_{(\delta\,\xi)-1}\rangle = s_0 \cdots s_{\delta\,\xi-1}$$

We consider the empty product to be one, corresponding to the fact that for a scalar array there is exactly one($\tau\,\xi^0 = 1$) indexable element in the array and as such a scalar array can never be empty.

We now have a vocabulary of scalar arrays, vector arrays and multidimensional arrays as:

$$\begin{aligned}
\xi^0 &= (\{\langle\rangle \mapsto e\}, E)\\
\xi^1 &= (\{\langle i_0\rangle \mapsto e_0, \ldots, \langle i_{\tau\,\vec{x}-1}\rangle \mapsto e_{\tau\,\vec{x}-1}\}, E)\\
\xi^{\delta\,\xi} &= (\{\vec{i}_0 \mapsto e_0, \ldots, \vec{i}_{\tau\,\vec{x}-1} \mapsto e_{\tau\,\xi-1}\}, E)
\end{aligned}$$

We define an indexing notation for the element $\vec{i}$ is mapped to given that $\vec{i}$ is a valid index for $\xi = (f, E)$ such that:

$$\xi[\,\vec{i}\,] = f(\,\vec{i}\,)$$

We will need to refer to the set of all arrays so we define it. The set of all shapes is $\boldsymbol{V}(\mathbb{Z}_{\geq 0})$ and with that we can define the set of all arrays as:

$$\boldsymbol{A}(E) = \{(f_s, E) \mid \text{for all } s \in \boldsymbol{V}(\mathbb{Z}_{\geq 0})\}$$

Partitionings according to our classifications of arrays as either scalar arrays, vector arrays, and multidimensional arrays are:

$$\begin{aligned}
\boldsymbol{A}_s(E) &= \{\xi \in \boldsymbol{A}(E) \mid \delta\,\xi = 0\}\\
\boldsymbol{A}_v(E) &= \{\xi \in \boldsymbol{A}(E) \mid \delta\,\xi = 1\}\\
\boldsymbol{A}_m(E) &= \{\xi \in \boldsymbol{A}(E) \mid \delta\,\xi > 1\}
\end{aligned}$$

We want to make a certain mapping for certain classifications of arrays. We introduce a family of functions g_s, g_v, g_m each on a different subset of all arrays:

$g_s : \boldsymbol{A}_s(E) \leftrightarrow E$	a scalar array's scalar projection
$g_v : \boldsymbol{A}_v(E) \leftrightarrow \boldsymbol{V}(E)$	a vector array's vector projection
$g_m : \boldsymbol{A}_m(E) \nrightarrow \boldsymbol{V}^*(E)$	a multidimensional array's nested vector projection

such that scalar arrays are mapped to elements of E and vector arrays are mapped to vectors of E:

$$\begin{aligned} g_s(\xi^0) &= \xi^0[\langle\rangle] \\ g_v(\xi^1) &= \langle \xi^1[\langle 0\rangle], \ldots, \xi^1[\langle \tau\, \xi^1 - 1\rangle]\rangle \end{aligned}$$

and multidimensional arrays are mapped to nested vectors $g_m(\xi^n) \in \boldsymbol{V}^n(E)$:

$$g_m(\xi^n)[i_0]\cdots[i_{n-1}] = \xi^n[\langle i_0, \ldots, i_{n-1}\rangle] \text{ for } 0 \leq^\star \langle i_0, \ldots, i_{n-1}\rangle <^\star \rho\, \xi^n$$

This projection is partial as any empty array cannot be uniquely mapped. Naturally this could be fixed by changing the projection by mapping such an array to its shape and expand the the target set of the projection $g_m : \boldsymbol{A}_m(E) \to \boldsymbol{V}^*(E) \cup \boldsymbol{V}(\mathbb{Z}_{\geq 0})$. We leave this as a remark and note that the purpose of this projection here is first and foremost to "show" an array in a terse manner.

Note that the scalar and vector projections are bijective. Think of these different projections as different ways of viewing arrays. Mullin defines the array in terms of equivalences. For instance she describes a scalar as equivalent to an array of one element indexable by only one index type with one value. A vector is equivalent to an array indexable by one index type with a range of values. Our basically equivalent definition instead avoids this use of equivalences from the beginning and we instead emphasize the array as an entity unto itself with bijective projections to other entities in certain circumstances.

For convenience we define functions over arrays from these projections. For the nested vector projection:

$$\underline{[\![\xi]\!]} = g_m(\xi)$$

For the scalar and vector projection:

$$[\![\xi]\!] = \begin{cases} g_s(\xi) & \text{when} \quad \xi \in \boldsymbol{A}_s(E) \\ g_v(\xi) & \text{when} \quad \xi \in \boldsymbol{A}_v(E) \end{cases}$$

As the used projections were bijective we may also define its inverse:

$$[\![x]\!]^{-1} = \begin{cases} g_s^{-1}(x) & \text{when} \quad x \in E \\ g_v^{-1}(x) & \text{when} \quad x \in \boldsymbol{V}(E) \end{cases}$$

This means that if we have an array ξ where $\rho\,\xi = \langle 2, 2\rangle$ consisting of the elements:

$$\{\xi[\langle 0,0\rangle] = 0, \xi[\langle 0,1\rangle] = 1, \xi[\langle 1,0\rangle] = 2, \xi[\langle 1,1\rangle] = 3\}$$

then $\underline{[\![\xi]\!]} = \langle\langle 0,1\rangle, \langle 2,3\rangle\rangle$ and for example:

$$\underline{[\![\xi]\!]}[1][0] = \langle\langle 0,1\rangle, \langle 2,3\rangle\rangle[1][0] = \langle 2,3\rangle[0] = 2 = \xi[\langle 1,0\rangle]$$

Note the difference between a vector and a vector array. Whereas a vector is of the form $\vec{x} = \langle e_0, e_1\rangle = (\{0 \mapsto e_0, 1 \mapsto e_1\}, \{e_0, e_1\})$ a vector array is of the form $x = (\{\langle 0\rangle \mapsto e_0, \langle 1\rangle \mapsto e_1\}, \{e_0, e_1\})$ and we have $[\![x]\!] = \langle e_0, e_1\rangle$. Considering a scalar e and the scalar array $y = (\{\langle\rangle \mapsto e,\}, \{e\})$ we have $[\![y]\!] = e$.

The covered notation of our interpretation of MOA and certain additional useful identities is summarized following table:

MOA	**Description**
ξ	array
Θ	empty array
ξ^0	scalar array
ξ^1	vector array
ξ^n	array of n dimensions
Θ^n	empty array of n dimensions
$\delta\,\xi \in \mathbb{Z}_{\geq 0}$	dimensions of an array
$\xi[\,\vec{i}\,]$	element at index of array
$[\![\xi^0]\!] = \xi^0[\langle\rangle]$	scalar projection of scalar array
$[\![\xi^1]\!] = \langle \xi^1[\langle 0\rangle], \ldots, \xi^1[\langle \tau\,\xi^1 - 1\rangle]\rangle$	vector projection of vector array
$\underline{[\![\xi^n]\!]}$	nested vector projection of array
$\rho\,\xi^0 = \langle\rangle$	shape of a scalar array
$\rho\,\xi^1 = \langle \tau\,\xi^1\rangle$	shape of a vector array
$\rho\,\xi = \langle s_0, \ldots, s_{(\delta\,\xi)-1}\rangle$	shape of an array
$\pi\,\langle s_0, \ldots, s_{(\delta\,\xi)-1}\rangle = s_0 \cdots s_{\delta\,\xi - 1}$	product of a shape's elements
$\tau\,\xi = \pi\,(\rho\,\xi)$	total amount of elements in an array

4.4 Defining operations

Now we gradually expand the notion of arrays and operations on arrays. We first introduce an abstract indexing operation. Then we describe it in relationship to a certain ordering on the elements, a layout, which in programming terms can be used to actually implement such an array. Operations are always described in terms of the abstract indexing notion and separates description of array operations from layout.

4.4.1 Psi - Array selection

We define a function that given an index vector and an array selects an array:

$$\psi : \{(\vec{i}, \xi) \in \boldsymbol{V}(\mathbb{Z}_{\geq 0}) \times \boldsymbol{A}(E) \mid \tau\,\vec{i} \leq \delta\,\xi \wedge 0 \leq^\star \vec{i} <^\star \tau\,\vec{i} \uparrow \rho\xi\} \rightarrow \boldsymbol{A}(E)$$

In other words the array selection function maps an index and array to an array given that the size of the index is less than or equal to the dimensions of the array and the index is constrained by the shape produced by taking the $\tau\,\vec{i}$ first elements of the shape of the array.

4.4.1.1 Resulting shape

For any array selection the array selected has a shape corresponding to the removal of the $\tau\,\vec{i}$ first elements from the shape:

$$\rho\,(\vec{i}\ \psi\ \xi) = \tau\,\vec{i} \downarrow \rho\xi$$

and correspondingly the change in dimensions, which is entirely dependent on the shape, is $\delta\,(\vec{i}\ \psi\ \xi) = \tau\,(\rho\,(\vec{i}\ \psi\ \xi)) = \delta\,\xi - \tau\,\vec{i}$. Note the shape for three different cases and its implications:

- For $\tau\,\vec{i} = 0$ we have $\rho\,(\langle\rangle\ \psi\ \xi) = 0 \downarrow \rho\xi = \rho\xi$
- For $\tau\,\vec{i} = \delta\,\xi$ we have $\rho\,(\vec{i}\ \psi\ \xi) = \delta\,\xi \downarrow \rho\xi = \langle\rangle$
- For $0 < \tau\,\vec{i} < \delta\,\xi$ we have $\rho\,(\vec{i}\ \psi\ \xi) = \tau\,\vec{i} \downarrow \rho\xi$ and:
  - Any $\vec{j}$ s.t. $0 <^\star \vec{j} \leq^\star \tau\,\vec{i} \downarrow \rho\xi$ is a valid index for $\vec{i}\ \psi\ \xi$
  - Any such $\vec{j}$ where also $\tau\,\vec{j} = \delta\,\xi - \tau\,\vec{i}$ is s.t. $\tau\,(\vec{i} \triangleright \vec{j}) = \delta\,\xi$ and $0 \leq^\star \vec{i} \triangleright \vec{j} <^\star \rho\xi$. The latter, recalling that $(\tau\,\vec{i} \uparrow \rho\xi) \triangleright (\tau\,\vec{i} \downarrow \rho\xi) = \rho\xi$, follows from that:

$$(0 \leq^\star \vec{i} <^\star \tau\,\vec{i} \uparrow \rho\xi) \wedge (0 \leq^\star \vec{j} <^\star \tau\,\vec{i} \downarrow \rho\xi) \Rightarrow 0 \leq^\star \vec{i} \triangleright \vec{j} <^\star \rho\xi$$

4.4.1.2 Selected array

Here we define the array selected for $\vec{i}\ \psi\ \xi$ by cases:

- For $\tau\,\vec{i} = 0$ we have an empty index and the selection is:

$$\langle\rangle\ \psi\ \xi = \xi$$

- For $\tau \vec{i} = \delta\, \xi$ we have a full index and the selection is:

$$\vec{i}\ \psi\ \xi \text{ s.t. } [\![\vec{i}\ \psi\ \xi]\!] = \xi[\,\vec{i}\,]$$

- For $0 < \tau \vec{i} < \delta\, \xi$ we have a partial index and the selection is:

$$\vec{i}\ \psi\ \xi$$

  Where all $\vec{j}$ s.t. they are a full and valid index for $\vec{i}\ \psi\ \xi$, that is $\tau \vec{j} = \delta\, \xi - \tau \vec{i}$ and $0 \leq^{\star} \vec{j} <^{\star} \tau \vec{i} \downarrow \rho\, \xi$, must select in ξ to produce the array a full selection would yield:

$$\vec{j}\ \psi\ (\vec{i}\ \psi\ \xi) = (\vec{i} \triangleright \vec{j})\ \psi\ \xi$$

4.4.1.3 As a partial right monoid action

Recall our coverage of a right monoid action 2.2. Given a monoid $(M, +, e)$, that is some set M, a binary and associative operation $+ : M \times M \to M$ and an identity element $e \in M$. The monoid is a right monoid action on S if there is an operation $* : M \times S \to S$ where:

- For all $m_1, m_2 \in M$ and all $s \in S$ we have $(m_1 + m_2) * s = m_2 * (m_1 * s)$.
- For all $s \in S$ we have $e * s = s$

We can introduce a weakening for the case of partiality [8]. In that case let the operation be partial $* : M \times S \nrightarrow S$ where:

- If for $m_1, m_2 \in M$ and $s \in S$ it is the case that $(m_1 + m_2) * s$ is defined and $m_2 * (m_1 * s)$ is defined then that implies $(m_1 + m_2) * s = m_2 * (m_1 * s)$.
- For all $s \in S$ it is the case that $e * s$ is defined and $e * s = s$

Clearly $\boldsymbol{V}(\mathbb{Z}_{\geq 0}), \triangleright, \langle\rangle)$ is a monoid. If $\vec{i}, (\vec{i} \triangleright \vec{j}) \in \boldsymbol{V}(\mathbb{Z}_{\geq 0})$ are valid indices of ξ that is $\vec{j}\ \psi\ \xi$ and $(\vec{i} \triangleright \vec{j})\ \psi\ \xi$ are defined then that implies $\vec{j}\ \psi\ (\vec{i}\ \psi\ \xi) = (\vec{i} \triangleright \vec{j})\ \psi\ \xi$. Furthermore for all $\xi \in \boldsymbol{A}(E)$ we have $\langle\rangle\ \psi\ \xi = \xi$. Thus the monoid $\boldsymbol{V}(\mathbb{Z}_{\geq 0}), \triangleright, \langle\rangle)$ is a right monoid action on ψ. This may shorten the specification of ψ given that there is already knowledge of monoids and partial right monoid actions.

4.4.2 Ravel - flattening

The function $rav : \boldsymbol{A}(E) \to \boldsymbol{A}(E)$ maps an array ξ to a vector array ξ^1 such that $\tau\,(rav\, \xi) = \tau\, \xi^1$. If ξ has indices $\vec{i}_0, \ldots, \vec{i}_{\tau\,\xi - 1}$ the elements at these indices in ξ, i.e. $\xi[\vec{i_0}], \ldots, \xi[\vec{i}_{\tau\,\xi-1}]$, all occur in some order in $rav\,\xi$ indexed by $\langle 0\rangle, \ldots, \langle \tau\,\xi - 1\rangle$.

Before we proceed note that for a scalar array ξ^0 there is only one arrangement:

$$[\![rav\,\xi^0]\!] = \langle [\![\xi^0]\!] \rangle$$

For a vector array ξ^1 we want to pick the arrangement:

$$[\![rav\,\xi^1]\!] = [\![\xi^1]\!]$$

Let rav be defined in terms of a bijective function dependent on the shape of the array $\rho\,\xi$:

$$f_{\rho\,\xi} : \{\vec{i}_0, \ldots, \vec{i}_{\tau\,\xi-1}\} \leftrightarrow \{\langle 0\rangle, \ldots, \langle \tau\,\xi - 1\rangle\}$$

that picks an arrangement satisfying the covered cases for ξ^0, ξ^1 and in general for ξ:

$$(rav\,\xi)[f_{\rho\,\xi}(\,\vec{i}\,)] = \xi[\,\vec{i}\,]$$

There are $(\tau\,\xi)!$ such functions; as many as there are ways to permute an ordering of $\tau\,\xi$ elements. We choose the function $\gamma(\vec{i}, \rho\,\xi)$ from 3.8 such that our arrangement becomes:

$$(rav\,\xi)[\langle \gamma(\,\vec{i}\,,\ \rho\,\xi\,)\rangle] = \xi[\,\vec{i}\,]$$

Or stated equivalently:

$$[\![rav\,\xi]\!][\gamma(\,\vec{i}\,,\ \rho\,\xi\,)] = \xi[\,\vec{i}\,]$$

Recall that for an array ξ and a valid index $\vec{i}$ such that $\tau\,\vec{i} = \delta\,\xi$ we had the identity $[\![\vec{i}\ \psi\ \xi]\!] = \xi[\,\vec{i}\,]$. Thus we may, for such cases, derive:

$$[\![\vec{i}\ \psi\ \xi]\!] = (rav\,\xi)[\langle \gamma(\,\vec{i}\,,\ \rho\,\xi\,)\rangle] = [\![rav\,\xi]\!][\gamma(\,\vec{i}\,,\ \rho\,\xi\,)]$$

From an implementors perspective $[\![rav\,\xi]\!][\gamma(\vec{i}, \rho\,\xi)]$ is an interpretation of a layout on a vector that realizes a higher dimensional array abstraction. The abstract notion of an array ξ and array selection ψ which aren't tied to any arrangement of elements gives us a tool to reason about what should happen to arrays when we apply operations on them. Indeed this is how Mullin proceeds to define operations. First however we consider our choices in regards to layout.

4.4.3 Gamma and change of layout

Given an array ξ such that $\rho\,\xi = \vec{s}$ and $\vec{i}$ is a valid index for ξ the function $\gamma(\vec{i}, \vec{s})$ is introduced in an equivalent manner to the one in given in 3.8:

$$\gamma(\vec{i}, \vec{s}) = \begin{cases} 0 & \tau\,\vec{i} = \tau\,\vec{s} = 0 \\ \vec{i}\,[0] & \tau\,\vec{i} = \tau\,\vec{s} = 1 \\ \vec{i}\,[n-1] + \vec{s}\,[n-1]\gamma((n-1) \uparrow \vec{i}, (n-1) \uparrow \vec{s}) & \tau\,\vec{i} = \tau\,\vec{s} = n > 1 \end{cases}$$

Recall the reformulations of γ in 3.8.2. We define those here as well in terms of our MOA vocabulary. Given k such that $0 \leq k \leq \delta\,\xi$ and a valid index $\vec{i}$ for ξ we have:

$$\gamma(\vec{i}, \vec{s}\,) = \sum_{k=0}^{\tau\,\vec{s}-1} \vec{i}[k]\pi\,(k+1 \downarrow \vec{s})$$

$$\mathit{stride}(\vec{s}, k) = \pi\,(k \downarrow \vec{s})$$

$$\mathit{start}(\vec{i}, \vec{s}, k) = \gamma(k \uparrow \vec{i}, k \uparrow \vec{s}\,)$$

$$\gamma((k \uparrow \vec{i}\,) \triangleright (k \downarrow \vec{i}\,), \vec{s}) = \mathit{start}(\vec{i}, \vec{s}, k)\mathit{stride}(\vec{s}, k) + \gamma(k \downarrow \vec{i}, k \downarrow \vec{s}\,)$$

This is a specific layout and as mentioned in 3.8 we can select a different one by a permutation $p(\langle x_0, \ldots, x_{n-1}\rangle) = \langle x_{p'(0)}, \ldots, x_{p'(n-1)}\rangle$ where $p' : \{0, \ldots, n-1\} \leftrightarrow \{0, \ldots, n-1\}$ is some bijective function. We could in fact always regard γ as using some permutation and consider the one we have provided here as the identity permutation, i.e. the one that does nothing. From this viewpoint we consider the layout we have provided γ_{id} where $id(x) = x$ and can regard the layout selected for an array as γ_{id} if the γ shown here is used. In general we define the layout given by some permutation p as:

$$\gamma_p(\vec{i}, \overrightarrow{\rho\,\xi}) = \gamma(p(\vec{i}), p(\overrightarrow{\rho\,\xi}))$$

If we consider layout to be a propery of the array, given some permutation p as mentioned, and denote this by naming or marking the array with this permutation ξ_p then indexing is defined by:

$$[\![\vec{i}\ \psi\ \xi_p]\!] = [\![\mathit{rav}\,\xi_p]\!][\gamma(p(\vec{i}), p(\rho\,\xi))]$$

Suppose we have an array ξ_{p_1}, i.e. we have named it the permuation p_1 indicating its layout, and we wish to change the array's layout according to a different permutation p_2 to get the array ξ_{p_2} which is identical in value at all indices. What we wish to do then is to iterate through $\mathit{rav}\,\xi_{p_1}$ and rearrange it to the differently layed out $\mathit{rav}\,\xi_{p_2}$. This is trivial to implement if we decide to simply allocate new memory and fill a new array. In that case we simply perform the assignment of ξ_{p_2} such that $[\![\vec{i}\ \psi\ \xi_{p_2}]\!] = [\![\vec{i}\ \psi\ \xi_{p_1}]\!]$ holds for all valid indices. If we however disallow reallocation we must find the permutation *swap* such that $[\![rav\,\xi_{p_1}]\!][i] = [\![rav\,\xi_{p_2}]\!][swap(i)]$ for all valid indices and, as the name would imply, imperatively rearrange the array by swapping elements. An example of such a permutation which takes you from row-major layout to column-major layout is to simply reverse the flat array or equivalently the permuting of its elements by the swapping implied by $swap(i) = n - i - 1$ where n is the length of the array.

In the rest of this treatment we will ignore layout selection or change and assume a fixed layout γ_{id} and simply refer to it as γ. The point of the ψ array selection abstraction is to hide the concern of layout and thus describe the change in elements at indices of arrays, over operations, in terms of layout independent indexing.

4.4.4 The Psi correspondence theorem

Consider some array $\xi \in \boldsymbol{A}(E)$ with a shape $\rho\,\xi$. Pick a subarray with the partial and valid index $\vec{v}$, i.e. $\tau\,\vec{v} \le \delta\,\xi$ and $0 \le^\star \vec{v} <^\star (\tau\,\vec{v}) \uparrow \rho\,\xi$, that is $\vec{v}\ \psi\ \xi$.

Then we have an array with shape $\rho\,(\vec{v}\ \psi\ \xi) = (\tau\,\vec{v}) \downarrow \rho\,\xi$. Let $\vec{u}$ be a valid index for this array, i.e $\tau\,\vec{u} = \delta\,\xi - \tau\,\vec{v}$ and $0 \le^\star \vec{u} <^\star (\tau\,\vec{v}) \downarrow \rho\,\xi$, so we arrive at the scalar array $\vec{u}\ \psi\ (\vec{v}\ \psi\ \xi)$ which we may write as:

$$\vec{u}\ \psi\ (\vec{v}\ \psi\ \xi) = (\vec{v} \triangleright \vec{u})\ \psi\ \xi$$

Now we wish to compute the element at every valid index $\vec{u}$ of $\vec{v}\ \psi\ \xi$. Note how the shape of the array is the concatenation:

$$\rho\xi = ((\tau\,\vec{v}) \uparrow \rho\,\xi) \triangleright ((\tau\,\vec{v}) \downarrow \rho\,\xi)$$

and recall we have:

$$\begin{aligned}[\![\vec{u}\ \psi\ (\vec{v}\ \psi\ \xi)]\!] &= [\![(\vec{v} \triangleright \vec{u})\ \psi\ \xi]\!] \\ &= [\![rav\,\xi]\!][\gamma(\vec{v} \triangleright \vec{u}, \rho\,\xi)] \\ &= [\![rav\,\xi]\!][\gamma(\vec{v} \triangleright \vec{u}, ((\tau\,\vec{v}) \uparrow \rho\,\xi) \triangleright ((\tau\,\vec{v}) \downarrow \rho\,\xi))]\end{aligned}$$

With parameters to γ in this form we see that we may apply the reformulated

γ that applies *start* and *stride*:

$$
\begin{aligned}
\mathit{stride}(\rho\,\xi, k) &= \pi\,(k \downarrow \rho\,\xi)\\
\mathit{start}(\vec{i}, \rho\,\xi, k) &= \gamma(k \uparrow \vec{i}, k \uparrow \rho\,\xi)\\
\gamma((k \uparrow \vec{i}\,) \triangleright (k \downarrow \vec{i}\,), \rho\,\xi) &= \mathit{start}(\vec{i}, \rho\,\xi, k)\mathit{stride}(\rho\,\xi, k) + \gamma(k \downarrow \vec{i}, k \downarrow \rho\,\xi)
\end{aligned}
$$

Thus we get(with $k = \tau\,\vec{v}$, $\vec{i} = \vec{v} \triangleright \vec{u}$):

$$
[\![\vec{u}\ \psi\ (\vec{v}\ \psi\ \xi)]\!] = [\![\mathit{rav}\,\xi]\!][\gamma(\vec{v}, \tau\,\vec{v} \uparrow \rho\,\xi)\pi\,(\tau\,\vec{v} \downarrow \rho\,\xi) + \gamma(\vec{u}, \tau\,\vec{v} \downarrow \rho\,\xi)]
$$

Now consider that we want to access every index that constitutes a valid and full index in the subarray $\vec{v}\ \psi\ \xi$. That means all $\vec{u}$ such that $0 \leq^{\star} \vec{u} <^{\star} \tau\,\vec{v} \downarrow \rho\,\xi$. However we do not care in which order. We know $\gamma(\vec{u}, \tau\,\vec{v} \downarrow \rho\,\xi)$ will give us indices k in the range $0 \leq k < \pi\,(\tau\,\vec{v} \downarrow \rho\,\xi)$ We may write this range as the vector $\iota\,(\pi\,(\tau\,\vec{v} \downarrow \rho\,\xi))$.

We now extend our vocabulary of vectors of non-negative integers slightly such that $\vec{x} \bullet \vec{y} = \vec{z}$ means $\vec{z}[i] = \vec{x}[i] \bullet \vec{y}[i]$ for all i s.t. $0 \leq i < k$ where we require that $\bullet$ is some binary arithmetic operation and $\tau\,\vec{z} = \tau\,\vec{x} = \tau\,\vec{y} = k$.

Then we may complete the description of computing all full indices $\vec{u}$ for $\vec{v}\ \psi\ \xi$ in its flattening as:

$$
\begin{aligned}
[\![\vec{u}\ \psi\ (\vec{v}\ \psi\ \xi)]\!] &= [\![\mathit{rav}\,\xi]\!][\vec{i}\,[x]]\\
\vec{i} &= \gamma(\vec{v}, \tau\,\vec{v} \uparrow \rho\,\xi)\pi\,(\tau\,\vec{v} \downarrow \rho\,\xi) + \iota\,(\pi\,(\tau\,\vec{v} \downarrow \rho\,\xi))
\end{aligned}
$$

Where $0 \leq x < \pi\,(\tau\,\vec{v} \downarrow \rho\,\xi)$. This idea is mostly due to the Psi correspondence theorem [17, p. 506]. This is a substantial improvement over repeated γ computations to perform operations over a subarray.

4.4.5 Reshaping arrays

We define reshaping in terms of the partial function:

$$
\overset{\bullet}{\rho} : \boldsymbol{V}(\mathbb{Z}_{\geq 0}) \times \boldsymbol{A}(E) \nrightarrow \boldsymbol{A}(E)
$$

Which maps a shape and an array to an array with the associated shape.

$$
\rho\,(\vec{s}\ \overset{\bullet}{\rho}\ \xi) = \vec{s}
$$

4.4.5.1 Reshaping arrays

Given an array ξ and a shape $\vec{s}$ the function $\vec{s} \overset{\bullet}{\rho} \xi$ is defined whenever $0 \leq^\star \vec{s}$ and $\xi \neq \Theta$ we don't allow reshaping an empty array as we wouldn't be able to assign elements to any indices.

- For a valid index $0 \leq^\star \vec{i} <^\star \vec{s}$ its element at that index is defined as:

  $$[\![\vec{i} \ \psi \ (\vec{s} \overset{\bullet}{\rho} \xi)]\!] = [\![rav\, \xi]\!][\gamma(\vec{i}, \vec{s}) \ mod \ (\tau\, \xi)]$$

  In other words indexing that would fall outside of $[\![rav\, \xi]\!]$ instead wraps around in it.

- Otherwise if there are no valid indices as one or more elements of the shape $\vec{s}$ are zero, we are reshaping to an empty array:

  $$(\vec{s} \overset{\bullet}{\rho} \xi) = \Theta^{\tau\, \vec{s}}$$

Example: Suppose we want to reshape an array ξ which is such that $[\![\xi]\!] = \iota\, 3 = \langle 0, 1, 2\rangle$ according to a shape $\langle 2, 3\rangle$. This array is $\langle 2, 3\rangle \overset{\bullet}{\rho} \xi$ and we index it as follows:

$$\begin{aligned}[\![\langle i_0, i_1\rangle \ \psi \ (\langle 2,3\rangle \overset{\bullet}{\rho} \xi^1)]\!] &= [\![rav\, \xi]\!][\gamma(\langle i_0, i_1\rangle, \langle 2,3\rangle) \ mod \ (\tau\, \xi)] \\ &= (\iota\, 3)[i_1 + 3 \cdot i_0 \ mod\ 3]\end{aligned}$$

for every index $\vec{i} = \langle i_0, i_1\rangle <^\star \langle 2, 3\rangle$:

$$\begin{aligned}
[\![\langle 0,0\rangle \ \psi \ (\langle 2,3\rangle \overset{\bullet}{\rho} \xi)]\!] &= (\iota\, 3)[\langle 0 + 3 \cdot 0 \ mod\ 3\rangle] = (\iota\, 3)[0] = 0 \\
[\![\langle 0,1\rangle \ \psi \ (\langle 2,3\rangle \overset{\bullet}{\rho} \xi)]\!] &= (\iota\, 3)[\langle 1 + 3 \cdot 0 \ mod\ 3\rangle] = (\iota\, 3)[1] = 1 \\
[\![\langle 0,2\rangle \ \psi \ (\langle 2,3\rangle \overset{\bullet}{\rho} \xi)]\!] &= (\iota\, 3)[\langle 2 + 3 \cdot 0 \ mod\ 3\rangle] = (\iota\, 3)[2] = 2 \\
[\![\langle 1,0\rangle \ \psi \ (\langle 2,3\rangle \overset{\bullet}{\rho} \xi)]\!] &= (\iota\, 3)[\langle 0 + 3 \cdot 1 \ mod\ 3\rangle] = (\iota\, 3)[0] = 0 \\
[\![\langle 1,1\rangle \ \psi \ (\langle 2,3\rangle \overset{\bullet}{\rho} \xi)]\!] &= (\iota\, 3)[\langle 1 + 3 \cdot 1 \ mod\ 3\rangle] = (\iota\, 3)[1] = 1 \\
[\![\langle 1,2\rangle \ \psi \ (\langle 2,3\rangle \overset{\bullet}{\rho} \xi)]\!] &= (\iota\, 3)[\langle 2 + 3 \cdot 1 \ mod\ 3\rangle] = (\iota\, 3)[2] = 2
\end{aligned}$$

In terms the array's nested vector projection we see we have reshaped $[\![\xi]\!] = \iota\, 3 = \langle 0, 1, 2\rangle$ into $\underline{[\![\langle 2,3\rangle \overset{\bullet}{\rho} (\iota\, 3)]\!]} = \langle\langle 0,1,2\rangle, \langle 0,1,2\rangle\rangle$. which can be layed out flatly as $[\![rav\,(\langle 2,3\rangle \overset{\bullet}{\rho} \xi)]\!] = \langle 0,1,2,0,1,2\rangle$.

4.4.5.2 Reshaping by cases

We inspect the values of $\vec{s} \overset{\bullet}{\rho} (rav\, \xi)$ by cases. For $\vec{s}$ such that:

- $\pi\, \vec{s} = 0$, at least one element in the shape vector is zero:

$$\vec{s} \overset{\bullet}{\rho} (rav\, \xi) = \Theta^{\delta\, \xi}$$

- $\pi\, \vec{s} \neq 0$ or equivalently $0 <^{\star} \vec{s}$ then either:
  - $\vec{s} = \langle\rangle$ and we are reshaping to a scalar array, which we may index as follows:

$$\begin{aligned} [\![\langle\rangle \ \psi \ (\langle\rangle \overset{\bullet}{\rho} (rav\, \xi))]\!] &= [\![rav\, \xi]\!][\gamma(\langle\rangle, \langle\rangle) \ mod \ (\tau\, (rav\, \xi))] \\ &= [\![rav\, \xi]\!][0 \ mod \ 1] \\ &= [\![rav\, \xi]\!][0] \end{aligned}$$

  - $\vec{s} \neq \langle\rangle$ and we are reshaping to an array that is not a scalar array

$$\vec{s} \overset{\bullet}{\rho} \xi = \vec{s} \overset{\bullet}{\rho} (rav\, \xi)$$

From these it follows[15, p. 14] that for all arrays ξ reshape is valid for:

$$\xi = (\rho\, \xi) \overset{\bullet}{\rho} (rav\, \xi) = (\rho\, \xi) \overset{\bullet}{\rho} \xi$$

Example: Now consider that we want to represent a 5×5 matrix of the kind:

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{bmatrix}$$

Let us interpret this as desiring a nested vector mapping for the array of:

$$\begin{aligned} \underline{[\![m_{5\times5}]\!]} = \langle &\langle 0, 1, 2, 3, 4\rangle, \\ &\langle 5, 6, 7, 8, 9\rangle, \\ &\langle 10, 11, 12, 13, 14\rangle, \\ &\langle 15, 16, 17, 18, 19\rangle, \\ &\langle 20, 21, 22, 23, 24\rangle\rangle \end{aligned}$$

Recall that $\iota\, 25 = \langle 0, 1, \ldots, 24\rangle$ and assume we have $[\![m_{25}]\!] = \iota\, 25$ and the shape we want is $\langle 5, 5\rangle$ whereas the shape of m_{25} is $\langle 25\rangle$. The desired array can be described as $m_{5\times5} = \langle 5, 5\rangle \overset{\bullet}{\rho}\ m_{25}$. Let's check that 13 is at $\langle 2, 3\rangle$ (i.e. that for $\langle i_0, i_1\rangle$ we have i_0 selecting row and i_1 selecting column) as expected:

$$\begin{aligned}
[\![\langle 2,3\rangle\ \psi\ m_{5\times5}]\!] &= [\![\langle 2,3\rangle\ \psi\ (\langle 5,5\rangle \overset{\bullet}{\rho}\ m_{25})]\!] \\
&= [\![rav\, m_{25}]\!][\gamma(\langle 2,3\rangle, \langle 5,5\rangle)\ mod\ \tau\, m_{25}] \\
&= (\iota\, 25)[3 + 5\cdot 2\ mod\ 25] = (\iota\, 25)[13] = 13
\end{aligned}$$

4.4.5.3 Reshaping to insert one

There is a special case of reshaping that is worth some extra mention. Suppose we wish to reshape some array ξ to have a shape where 1 is either prepended to it, appended to it or inserted in it. That is we want to have a shape $\vec{s} = (k \uparrow \rho\,\xi) \triangleright \langle 1\rangle \triangleright (k \downarrow \rho\,\xi)$ which would yield indices of the form $\vec{j} = (k \uparrow \vec{i}\,) \triangleright \langle 0\rangle \triangleright (k \downarrow \vec{i}\,)$ where $0 \le k \le \delta\,\xi$ and $\vec{i}$ is a valid full index for ξ. Reshaping ξ to the shape $\vec{s}$ and indexing it with $\vec{j}$ yields:

$$[\![\vec{j}\ \psi\ (\vec{s} \overset{\bullet}{\rho}\ \xi)]\!] = [\![rav\,\xi]\!][\gamma(\vec{j}\,, \vec{s}\,)\ mod\ \tau\,\xi]$$

We know $\vec{j}$ will never exceed the size of $\tau\,\xi$ in $\gamma(\vec{j}\,, \vec{s}\,)$ as $\pi\,\vec{s} = \tau\,\xi$ and thus we may state:

$$[\![\vec{j}\ \psi\ (\vec{s} \overset{\bullet}{\rho}\ \xi)]\!] = [\![rav\,\xi]\!][\gamma(\vec{j}\,, \vec{s}\,)]$$

where by definition the elements of ξ are arranged in $rav\,\xi$ as follows:

$$[\![\vec{i}\ \psi\ \xi]\!] = [\![rav\,\xi]\!][\gamma(\vec{i}\,, \rho\,\xi)]$$

We assert that $\gamma(\vec{j}\,, \vec{s}) = \gamma(\vec{i}\,, \rho\,\xi)$ for some chosen k where $0 \le k \le \delta\,\xi$

$$\gamma(\vec{j}\,, \vec{s}) = \gamma((k \uparrow \vec{i}\,) \triangleright \langle 0\rangle \triangleright (k \downarrow \vec{i}\,), (k \uparrow \rho\,\xi) \triangleright \langle 1\rangle \triangleright (k \downarrow \rho\,\xi))$$

for this problem we will require some slight reformulations, recall that:

$$\gamma(\vec{i}, \vec{s}\,) = \sum_{x=0}^{\tau\,\vec{s}-1} \vec{i}\,[x]\pi\,(x + 1 \downarrow \vec{s}\,)$$

We could split out the product of the sum and represent the sum as a sum over the pointwise product, i.e. $(\vec{x}\cdot\vec{y}\,)[i] = \vec{x}\,[i]\vec{y}\,[i]$, between a vector of strides and the index vector. Let $strides(\vec{s}\,)$ denote the vector of size $\tau\,\vec{s}$ such that:

$$strides(\vec{s}\,)[i] = stride(\vec{s}, i + 1) = \pi\,(i + 1 \downarrow \vec{s}\,)$$

We need to map products over vectors so let $e\vec{x} = \vec{y}$ where $\vec{y}[i] = e\vec{x}[i]$. Naturally such maps distributes over concatenation $e(\vec{x} \triangleright \vec{y}) = e\vec{x} \triangleright e\vec{y}$. Then some crucial relationships of *strides* are:

$$\begin{gathered} \mathit{strides}(\langle s\rangle \triangleright \vec{s}\,) = \langle \pi\,\vec{s}\,\rangle \triangleright \mathit{strides}(\vec{s}\,) \\ \mathit{strides}(\vec{x} \triangleright \vec{y}\,) = (\pi\,\vec{y}\,)\mathit{strides}(\vec{x}\,) \triangleright \mathit{strides}(\vec{y}) \end{gathered}$$

where the first identity follows trivially from the definition of *strides* while the second can be derived by induction with the first as the basis. The identities also follow from that the vector $\mathit{strides}(\vec{s})$ is the reversed prefix product of $\vec{s}$ with s_0 replaced with 1 and then reversed. That is we take $\langle s_0, s_1, s_2, s_3\rangle$, get $\langle 1, s_3, s_2, s_1\rangle$, get the prefix product $\langle 1, s_3, s_2s_3, s_1s_2s_3\rangle$ and then the reversal yields $\langle s_1s_2s_3, s_2s_3, s_3, 1\rangle$ which would be the result of $\mathit{strides}(\langle s_0, s_1, s_2, s_3\rangle)$. This is perhaps better understood with the recurrence:

$$\mathit{strides}(\vec{s}\,)[i] = \begin{cases} 1 & \text{when } i = \tau\,\vec{s} - 1 \\ \vec{s}\,[i+1]\mathit{strides}(\vec{s}\,)[i+1] & \text{when } i < \tau\,\vec{s} - 1 \end{cases}$$

The identities for *strides* allow us to derive the following:

$$\begin{gathered} \vec{x} = k \uparrow \rho\,\xi \\ \vec{y} = k \downarrow \rho\,\xi \end{gathered}$$

$$\begin{aligned} \mathit{strides}((\vec{x} \triangleright \langle 1\rangle) \triangleright \vec{y}) &= (\pi\,\vec{y}\,)\mathit{strides}(\vec{x} \triangleright \langle 1\rangle) \triangleright \mathit{strides}(\vec{y}\,) \\ &= (\pi\,\vec{y}\,)(\ (\pi\,\langle 1\rangle)\mathit{strides}(\vec{x}) \triangleright \mathit{strides}(\langle 1\rangle)\ ) \triangleright \mathit{strides}(\vec{y}\,) \\ &= (\pi\,\vec{y}\,)(\mathit{strides}(\vec{x}\,) \triangleright \mathit{strides}(\langle 1\rangle)) \triangleright \mathit{strides}(\vec{y}\,) \\ &= (\pi\,\vec{y}\,)\mathit{strides}(\vec{x}\,) \triangleright (\pi\,\vec{y}\,)\mathit{strides}(\langle 1\rangle) \triangleright \mathit{strides}(\vec{y}\,) \\ &= (\pi\,\vec{y}\,)\mathit{strides}(\vec{x}\,) \triangleright \langle \pi\,\vec{y}\,\rangle \triangleright \mathit{strides}(\vec{y}\,) \end{aligned}$$

With *strides* we can reformulate our sum as:

$$\sum_{x=0}^{\tau\,\vec{s}-1} \vec{i}\,[x]\pi\,(x+1 \downarrow \vec{s}) = \sum_{x=0}^{\tau\,\vec{s}-1} (\vec{i} \cdot \mathit{strides}(\vec{s}\,))[x]$$

and equivalently we now have:

$$\gamma(\vec{i}\,, \vec{s}\,) = \sum \vec{i} \cdot \mathit{strides}(\vec{s}\,)$$

Notice we remove the ranges as they are implicit properties of the vector. We shall rely on certain trivial identities that we omit proof of:

$$\begin{gathered} \sum \vec{x} \triangleright \vec{y} = \sum \vec{x} + \sum \vec{y} \\ \tau\,\vec{x} = \tau\,\vec{u} \wedge \tau\,\vec{y} = \tau\,\vec{v} \Rightarrow (\vec{x} \triangleright \vec{y}) \cdot (\vec{u} \triangleright \vec{v}) = (\vec{x} \cdot \vec{u}) \triangleright (\vec{y} \cdot \vec{v}) \\ \tau\,\vec{x} = \tau\,\vec{y} \Rightarrow k \uparrow (\vec{x} \cdot \vec{y}) = (k \uparrow \vec{x}) \cdot (k \uparrow \vec{y}) \\ \tau\,\vec{x} = \tau\,\vec{y} \Rightarrow k \downarrow (\vec{x} \cdot \vec{y}) = (k \downarrow \vec{x}) \cdot (k \downarrow \vec{y}) \end{gathered}$$

Now returning to our original problem we have:

$$\begin{aligned}\gamma(\vec{j}\,,\vec{s}) &= \gamma((k \uparrow \vec{i}\,) \triangleright \langle 0\rangle \triangleright (k \downarrow \vec{i}\,), (k \uparrow \rho\,\xi) \triangleright \langle 1\rangle \triangleright (k \downarrow \rho\,\xi)) \\ &= \sum \vec{j} \cdot \mathit{strides}(\vec{s}\,)\end{aligned}$$

we list $\vec{j}$ and the formerly derived $\mathit{strides}(\vec{s}\,)$

$$\begin{array}{rcccccc}\vec{j} & = & (k \uparrow \vec{i}) & \triangleright & \langle 0\rangle & \triangleright & (k \downarrow \vec{i}) \\ \mathit{strides}(\vec{s}\,) & = & (\pi\,\vec{y})\mathit{strides}(\vec{x}) & \triangleright & \langle \pi\,\vec{y}\rangle & \triangleright & \mathit{strides}(\vec{y})\end{array}$$

and as they are the same size pairwise we may write:

$$\begin{aligned}\vec{j} \cdot \mathit{strides}(\vec{s}\,) &= ((k \uparrow \vec{i}) \cdot (\pi\,\vec{y})\mathit{strides}(\vec{x})) \triangleright (\langle 0\rangle \cdot \langle \pi\,\vec{y}\rangle) \triangleright ((k \downarrow \vec{i}) \cdot \mathit{strides}(\vec{y})) \\ &= ((k \uparrow \vec{i}) \cdot (\pi\,\vec{y})\mathit{strides}(\vec{x})) \triangleright \langle 0\rangle \triangleright ((k \downarrow \vec{i}) \cdot \mathit{strides}(\vec{y}))\end{aligned}$$

which leads us to the sum, recalling that $\vec{x} = k \uparrow \rho\,\xi$ and $\vec{y} = k \downarrow \rho\,\xi$:

$$\begin{aligned}\gamma(\vec{j}, \vec{s}) &= \sum \vec{j} \cdot \mathit{strides}(\vec{s}\,) \\ &= \sum (k \uparrow \vec{i}) \cdot (\pi\,\vec{y})\mathit{strides}(\vec{x}) + \sum (k \downarrow \vec{i}) \cdot \mathit{strides}(\vec{y}) \\ &= \sum (k \uparrow \vec{i}) \cdot (\pi\,\vec{y})\mathit{strides}(\vec{x}) \triangleright (k \downarrow \vec{i}) \cdot \mathit{strides}(\vec{y}) \\ &= \sum ((k \uparrow \vec{i}) \triangleright (k \downarrow \vec{i})) \cdot ((\pi\,\vec{y})\mathit{strides}(\vec{x}) \triangleright \mathit{strides}(\vec{y})) \\ &= \sum \vec{i} \cdot \mathit{strides}(\vec{x} \triangleright \vec{y}) = \sum \vec{i} \cdot \mathit{strides}(\rho\,\xi) = \gamma(\vec{i}, \rho\,\xi)\end{aligned}$$

So in summary: When we perform a reshape of ξ to have a shape of $\vec{s} = (k \uparrow \rho\,\xi) \triangleright \langle 1\rangle \triangleright (k \downarrow \rho\,\xi)$ for k such that $0 \le k \le \delta\,\xi$ then for a valid index $\vec{i}$ of ξ we have a valid index $(k \uparrow \vec{i}) \triangleright \langle 0\rangle \triangleright (k \downarrow \vec{i})$ of $\vec{s}\ \overset{\bullet}{\rho}\ \xi$ and its values are defined by:

$$[\![((k \uparrow \vec{i}) \triangleright \langle 0\rangle \triangleright (k \downarrow \vec{i}))\ \psi\ \vec{s}\ \overset{\bullet}{\rho}\ \xi]\!] = [\![\mathit{rav}\,\xi]\!][\gamma(\vec{i}, \rho\,\xi)]$$

The motivation for expanding a bit on this particular case is, as we shall soon get to, that we will define concatenation along the first dimension for arrays with the requirement that two arrays can only be concatenated together if the tail of their shapes are identical. So an array of shape $\langle 3, 5, 5\rangle$ can only be concatenated with one of shape $\langle x, 5, 5\rangle$. However it is natural to want to concatenate together an array of shape $\langle 3, 5, 5\rangle$ with one of $\langle 5, 5\rangle$. The result here shows us we can do that at no cost by reshaping the array of shape $\langle 5, 5\rangle$ to one of shape $\langle 1, 5, 5\rangle$. Not dealing with this concatenation in the definition of concatenation makes the requirements and definition of it simpler.

4.4.6 Extending take, drop and concatenate to arrays

There is a natural extension of take, drop and concatenate for vectors to arrays, specifically along the first dimension, that is consistent with our definitions for vectors. These are also largely consisten with Mullin's definitions. We specifically exclude the arrays that are scalar array, so we define the set of arrays that exclude them:

$$\boldsymbol{A}(E)_{>0} = \{\xi \in \boldsymbol{A}(E) \mid \delta\,\xi \neq 0\}$$

All the identities that held for take, drop and concatenate for vectors also hold for arrays.

4.4.6.1 Take

Given an array ξ with shape $\rho\,\xi$ we define the function:

$$\uparrow_* : \mathbb{Z}_{\geq 0} \times \boldsymbol{A}(E)_{>0} \to \boldsymbol{A}(E)$$

Such that:

$$\rho\,(i \uparrow_* \xi) = \langle min(i, (\rho\,\xi)[0])\rangle \triangleright (1 \downarrow \rho\,\xi)$$

Where when $min(i, (\rho\,\xi)[0]) = (\rho\,\xi)[0]$ then it is the original array:

$$i \uparrow_* \xi = \xi$$

Otherwise given a valid index $\vec{j}$, i.e $0 \leq^\star \vec{j} <^\star \rho\,(i \uparrow_* \xi)$, it is an array with values:

$$[\![\vec{j}\ \psi\ (i \uparrow_* \xi)]\!] = [\![\vec{j}\ \psi\ \xi]\!]$$

4.4.6.2 Drop

Given an array ξ with shape $\rho\,\xi$ we define the function:

$$\downarrow_* : \mathbb{Z}_{\geq 0} \times \boldsymbol{A}(E)_{>0} \to \boldsymbol{A}(E)$$

such that:

$$\rho\,(i \downarrow_* \xi) = \langle(\rho\,\xi)[0] - min(i, (\rho\,\xi)[0])\rangle \triangleright (1 \downarrow \rho\,\xi)$$

Where when $min(i, (\rho\,\xi)[0]) = (\rho\,\xi)[0]$ then it is an empty array:

$$i \downarrow_* \xi = \Theta^{\delta\,\xi}$$

otherwise, given a valid index $\vec{j}$, i.e $0 \leq^\star \vec{i} <^\star \rho\,(i \downarrow_* \xi)$, it is an array with values:

$$[\![\vec{j}\ \psi\ (i \downarrow_* \xi)]\!] = [\![\langle j + (\rho\,\xi)[0]\rangle\ \psi\ \xi]\!]$$

4.4.6.3 Concatenate

Given arrays ξ_1 and ξ_2 with we define their concatenation $\xi_1 \rhd_* \xi_2$ such that:

$$\rhd_* : \{(\xi_1, \xi_2) \in \boldsymbol{A}(E)_{>0} \times \boldsymbol{A}(E)_{>0} \mid (1 \downarrow \rho\,\xi_1) = (1 \downarrow \rho\,\xi_2)\} \to \boldsymbol{A}(E)$$

That is concatenation is defined for arrays of matching elements in every but the first element of their shapes or when either array is empty.

The shape of a concatenation of arrays is defined as:

$$\rho\,(\xi_1 \rhd_* \xi_2) = \langle(\rho\,\xi_1)[0] + (\rho\,\xi_2)[0]\rangle \rhd (1 \downarrow \rho\,\xi_1)$$

and its values are defined for valid indices $\vec{i}$ where $0 \leq^\star \vec{i} <^\star \rho\,(\xi_1 \rhd_* \xi_2)$ as:

$$[\![\vec{i}\ \psi\ (\xi_1 \rhd_* \xi_2)]\!] = \begin{cases} [\![\vec{i}\ \psi\ \xi_1]\!] & \text{when} \quad \vec{i}[0] < (\rho\,\xi_1)[0] \\ [\![\vec{j}\ \psi\ \xi_2]\!] & \text{when} \quad (\rho\,\xi_1)[0] \leq \vec{i}[0] < (\rho\,\xi_1)[0] + (\rho\,\xi_2)[0] \end{cases}$$

where $\vec{j} = \langle\vec{i}[0] - (\rho\,\xi_1)[0]\rangle \rhd (1 \downarrow \vec{i})$ which is a valid index for ξ_2.

Note for a valid index such that $\vec{i} = \langle i\rangle \rhd (1 \downarrow \vec{i})$ we have the identity following from $(1 \downarrow \vec{i})\ \psi\ (\langle i\rangle\ \psi\ (\xi_1 \rhd_* \xi_2)) = (\langle i\rangle \rhd (1 \downarrow \vec{i}))\ \psi\ (\xi_1 \rhd_* \xi_2)$ that:

$$\langle i\rangle\,\psi\,(\xi_1 \rhd_* \xi_2) = \begin{cases} \langle i\rangle\ \psi\ \xi_1 & \text{when} \quad i < (\rho\,\xi_1)[0] \\ \langle i - (\rho\,\xi_1)[0]\rangle\ \psi\ \xi_2 & \text{when} \quad (\rho\,\xi_1)[0] \leq i < (\rho\,\xi_1)[0] + (\rho\,\xi_2)[0] \end{cases}$$

4.4.7 Rotation

Here we describe an operation rotate right $\circlearrowright\ : \mathbb{Z}_{\geq 0} \times \boldsymbol{A}(E) \nrightarrow \boldsymbol{A}(E)$ and rotate left $\circlearrowleft\ : \mathbb{Z}_{\geq 0} \times \boldsymbol{A}(E) \nrightarrow \boldsymbol{A}(E)$ sometimes called shift, that reorders the elements in an array:

For $i \in \mathbb{Z}_{\geq 0}$ and ξ such that $\delta\,\xi \geq 1$ we define:

$$\rho\,(i \circlearrowright \xi) = \rho\,(i \circlearrowleft \xi) = \rho\,\xi$$

If $\langle j\rangle$ is a valid, possibly partial, index for $i \circlearrowright \xi$ or $i \circlearrowleft \xi$, i.e. $0 \leq j < (\rho\,\xi)[0]$, the array at that index is defined as:

$$\langle j\rangle\ \psi\ (i \circlearrowright \xi) = \langle(j - i)\ mod\ (\rho\,\xi)[0]\rangle\ \psi\ \xi$$
$$\langle j\rangle\ \psi\ (i \circlearrowleft \xi) = \langle(j + i)\ mod\ (\rho\,\xi)[0]\rangle\ \psi\ \xi$$

In case of a partial index we are reordering indexing along the first dimension. For vector arrays it is exactly:

$$\langle j\rangle\ \psi\ (i \circlearrowright \xi^1) = \langle(j - i)\ mod\ \tau\,\xi^1\rangle\ \psi\ \xi^1$$
$$\langle j\rangle\ \psi\ (i \circlearrowleft \xi^1) = \langle(j + i)\ mod\ \tau\,\xi^1\rangle\ \psi\ \xi^1$$

4.4.7.1 Rotation example

Consider the array from before $m_{5\times5}$. The rotation $1 \circlearrowright m_{5\times5}$ makes us index it as follows:

$$\begin{aligned}\langle j\rangle \;\psi\; (\langle i\rangle \;\psi\; (1 \circlearrowright m_{5\times5})) &= \langle j\rangle \;\psi\; (\langle i-1 \;mod\; 5\rangle \;\psi\; m_{5\times5}) \\ &= \langle i-1 \;mod\; 5, j\rangle \;\psi\; m_{5\times5}\end{aligned}$$

Which will result in the "matrix" changing as follows:

$$\underline{[\![m_{5\times5}]\!]} = \begin{array}{l}\langle\langle 0,1,2,3,4\rangle,\\ \langle 5,6,7,8,9\rangle,\\ \langle 10,11,12,13,14\rangle,\\ \langle 15,16,17,18,19\rangle,\\ \langle 20,21,22,23,24\rangle\rangle\end{array} \qquad \underline{[\![1 \circlearrowright m_{5\times5}]\!]} = \begin{array}{l}\langle\langle 20,21,22,23,24\rangle\\ \langle 0,1,2,3,4\rangle\\ \langle 5,6,7,8,9\rangle,\\ \langle 10,11,12,13,14\rangle,\\ \langle 15,16,17,18,19\rangle\rangle\end{array}$$

There is another natural way of defining rotation[2] in terms of taking and dropping. We define it correctly for one complete rotation and give it the name shuffle and the direction of the shuffle. Let $l = (\rho\,\xi)[0]$, we then have:

$$\begin{aligned}\mathit{shuffleright}(i,\xi) &= (l-i \downarrow_* \xi) \rhd_* (l-i \uparrow_* \xi)\\ \mathit{shuffleleft}(i,\xi) &= (i \downarrow_* \xi) \rhd_* (i \uparrow_* \xi)\end{aligned}$$

In order to make it correct for any rotation we may simply perform the modulo for i and obtain:

$$\begin{aligned}\mathit{shuffleright}(i \;mod\; l,\xi) &= i \circlearrowright \xi\\ \mathit{shuffleleft}(i \;mod\; l,\xi) &= i \circlearrowleft \xi\end{aligned}$$

4.4.8 The point-wise and scalar extension

Given $\xi_1 \in \boldsymbol{A}(E)$ and $\xi_2 \in \boldsymbol{A}(E)$ such that $\rho\,\xi_1 = \rho\,\xi_2$. If we have a binary function

$$op : E \times E \to E$$

Then we point-wise extend it to apply over arrays such that we get an array of unchanged shape:

$$\rho\,(\xi_1 \;op_p\; \xi_2) = \rho\,\xi_1 = \rho\,\xi_2$$

For valid indices $\vec{i}$ where $\tau\,\vec{i} = \delta\,(\rho\,(\xi_1 \;op_p\; \xi_2))$ we define:

$$[\![\vec{i} \;\psi\; (\xi_1 \;op_p\; \xi_2)]\!] = [\![\vec{i} \;\psi\; \xi_1]\!] \;op\; [\![\vec{i} \;\psi\; \xi_2]\!]$$

[2]credits due to Benjamin Chetioui

For $\xi \in \boldsymbol{A}(E)$ if we have a unary function $uop : E \to E$ we point-wise extend it to apply over arrays such that we get an array of unchanged shape:

$$\rho\,(uop_p\,\xi) = \rho\,\xi$$

For valid indices $\vec{i}$ where $\tau\,\vec{i} = \delta\,\xi$ we define:

$$[\![\vec{i}\ \psi\ (uop_p\,\xi)]\!] = uop\,[\![\vec{i}\ \psi\ \xi]\!]$$

We define the scalar extension for an array $\xi \in \boldsymbol{A}(E)$ of a binary operation $op : E \times E \to E$ and $\xi^0 \in \boldsymbol{A}(E)$ as a shape preserving operation:

$$\rho\,(\xi^0\ op_s\ \xi) = \rho\,(\xi\ op_s\ \xi^0) = \rho\,\xi$$

For valid indices $\vec{i}$ where $\tau\,\vec{i} = \delta\,\xi$ we define:

$$[\![\vec{i}\ \psi\ (\xi^0\ op_s\ \xi)]\!] = [\![\xi^0]\!]\ op\ [\![\vec{i}\ \psi\ \xi]\!]$$
$$[\![\vec{i}\ \psi\ (\xi\ op_s\ \xi^0)]\!] = [\![\vec{i}\ \psi\ \xi]\!]\ op\ [\![\xi^0]\!]$$

4.4.9 Reduction and scan

A useful operation over vectors in a programming or mathematics setting is to compute the sum or product over one. A generalization of this is the reduction operation, here along the first dimension.

Given an array $\xi \in \boldsymbol{A}(E)$ and a binary operation $op : E \times E \to E$ such that $id(op)$ picks an identity element for op. That is for any $e \in E$ we have $op(e, id(op)) = e = op(id(op), e)$ we define the reduction operation $reduce_{op} : \boldsymbol{A}(E) \nrightarrow \boldsymbol{A}(E)$ in terms of the point-wise extension of op. Note the partiality; reduction is undefined for a scalar array.

First we decide that the shape of a reduction is the removal of an element in the case of a vector array or the removal of a dimension otherwise:

$$\rho\,(reduce_{op}(\xi^1)) = \langle(\rho\,\xi^1)[0] - min(1, (\rho\,\xi^1)[0])\rangle$$
$$\rho\,(reduce_{op}(\xi)) = 1 \downarrow \rho\,\xi$$

We then decide the resulting array. Given ξ such that $\delta\,\xi \geq 1$ we have for an empty array:

$$reduce_{op}(\Theta) = (1 \downarrow \rho\,\Theta)\ \overset{\bullet}{\rho}\ ([\![id(op)]\!]^{-1})$$

This means we assign, by a reshape, the identity element for the operation at every element of an array with the required shape. For a scalar array we define it as:

$$reduce_{op}(\xi^0) = \xi^0$$

For a vector array we define it as:

$$\mathit{reduce}_{op}(\xi^1) = (\langle 0\rangle \ \psi \ \xi^1) \ op_s \ \mathit{reduce}_{op}(1 \downarrow_* \xi^1)$$

Otherwise for any other array we define it as:

$$\mathit{reduce}_{op}(\xi) = (\langle 0\rangle \ \psi \ \xi) \ op_p \ \mathit{reduce}_{op}(1 \downarrow_* \xi)$$

Is this definition sound? We need to inspect the shapes to determine this. We see that it has a proper definition in the case of the empty array and a vector array. In the case of any other array ξ we know the shape of $\langle 0\rangle \ \psi \ \xi$ will be $\rho\,(\langle 0\rangle \ \psi \ \xi) = 1 \downarrow \rho\,\xi$. Recalling the definition for drop for arrays we have:

$$\rho\,(1 \downarrow_* \xi) = \langle(\rho\,\xi)[0] - \mathit{min}(1, (\rho\,\xi)[0])\rangle \triangleright (1 \downarrow \rho\,\xi)$$

and we see $\rho\,(\mathit{reduce}_{op}(1 \downarrow_* \xi)) = 1 \downarrow \rho\,(1 \downarrow_* \xi) = 1 \downarrow \rho\,\xi$ which satisfies the requirements for the point-wise extension; the shapes are the same for both parameters of the point-wise extended operation.

Example: Consider a vector array such that $\rho\,\xi = \langle 10\rangle$ and $[\![\langle i\rangle \ \psi \ \xi]\!] = i$, that is $[\![\xi]\!] = \iota\,10$. We compute $\mathit{reduce}_+(\xi)$ were $\mathit{id}(+) = 0$.

First observe that $\langle 0\rangle \ \psi \ (i \downarrow_* \xi^1) = \langle i\rangle \ \psi \ \xi^1$ and $1 \downarrow_* (i \downarrow_* \xi^1) = (i+1) \downarrow_* \xi^1$ as long as the dropping does not empty the array. Thus a repeated expansion gives us:

$$\mathit{reduce}_+(\xi) = (\langle 0\rangle \ \psi \ \xi) +_s \ldots +_s (\langle 9\rangle \ \psi \ \xi) +_s \mathit{reduce}_+(10 \downarrow_* \xi)$$

and the last term is

$$\mathit{reduce}_+(10 \downarrow_* \xi) = \mathit{reduce}_+(\Theta) = \langle\rangle \ \overset{\bullet}{\rho} \ [\![\mathit{id}(+)]\!]^{-1} = \xi^0$$

where $[\![\xi^0]\!] = \mathit{id}(+) = 0$ and we get:

$$\begin{aligned}[\![\mathit{reduce}_+(\xi)]\!] &= [\![\langle 0\rangle \ \psi \ \xi]\!] +_s \ldots +_s [\![\langle 9\rangle \ \psi \ \xi]\!] +_s [\![\mathit{reduce}_+(10 \downarrow_* \xi)]\!] \\ &= 0 + \ldots + 9 + 0 = 45\end{aligned}$$

Another useful construct is to keep the intermediary reductions. That is for a sum $0+1+2+3+4$ we would want a sequence of $0, 0+1, 0+1+2, 0+1+2+3, 0+1+2+3+4$. Mullin has opted to call this a scan, but it is also called a prefix sum.

Let $\mathit{scan} : \boldsymbol{A}(E) \nrightarrow \boldsymbol{A}(E)$ be a function that produces the array of all intermediary reductions steps taken by reduce. Note the partiality; scan being defined in terms of reduction is also undefined for a scalar array. Its shape is:

$$\rho\,(scan_{op}(\xi)) = \rho\,\xi$$

It is defined for empty arrays as:

$$scan_{op}(\Theta) = \Theta$$

For a vector array as:

$$scan_{op}(\xi^1) = scan_{op}((\tau\,\xi - 1) \uparrow_* \xi) \triangleright_* reduce_{op}(\xi)$$

Otherwise for any other array as:

$$scan_{op}(\xi) = scan_{op}((\delta\,\xi - 1) \uparrow_* \xi) \triangleright_* reduce_{op}(\xi)$$

4.5 On evaluation of expressions

It is important to stress that the intent in specifying these operations is not to indicate an efficient implementation. The symbolic expansion and evaluation of these expressions would likely be a horribly slow way of performing such reductions. Rather the motivation is that we can specify these operations and manipulate them as expressions that compose and have reduction steps towards forms which we can generate code for. There are many considerations to be had in finding the right way to implement an expression. For instance do we move data when we rotate or do we reinterpret the index space? If we are working with a small array and reading from it billions of times perhaps it's best to perform the data move to remove the reindexing and modulo operations. However if the size of the array is in the billions it is likely a very bad to perform the data move and wise to stick with a reindexing approach.

4.6 Summary and related work

We have shown the basic building blocks of MOA and its view of arrays. There is a substantial body of work in Mullin's thesis which simply cannot be covered here. Certain aspects we have seen before like reduction but also other operations are covered in her thesis[15]. Most array operations(e.g. concatenation, take, drop and rotation) are extended to work along any dimension. The ψ indexing is extended to allow for array left arguments such that it selects several indices(so it can be provided an array of valid indices to select an array of values). Previous work has shown[18] it is feasible to describe a reduction order on MOA expressions that will yield a form that is possible to realize as performant array code. Compilation of performant array code from MOA expressions is also explored in[17] which suggest using a technique that has come to be called dimension lifting of arrays in which an array is partitioned in subarrays to model a spread of a computation over an array such that the subarrays can be sent off to be computed independently(spread over threads, over a network of machines, etc.). In certain cases where the Psi correspondence theorem may be applied(when multidimensional indices do not end up being used in the computation itself and must be kept) such subarrays may be entirely freed from the branching overhead of nested loopings and we believe these kinds of forms are particularly admissible for vectorization. We will later provide some data points towards this end, but we will not make a strong claim as to whether this is the case as more investigation is needed.

CHAPTER 5

Modeling A Mathemathics of Arrays in Magnolia

5.1 Introduction

In this chapter we will go through parts of a specification of MOA in Magnolia. We will only use the specification or interface facility of Magnolia and focus on the use of the concept construct in Magnolia. In Magnolia the concept construct can be seen as similar to the interface construct in Java or the typeclass in Haskell that both specify what operations types must support or the contract that must be upheld for an implmentation of the concept. However in Magnolia specifying axioms is also part of such a specification. This has two obvious uses; to generate tests for any implemetation of the concept to check that axioms hold and to rewrite expressions according to the assertions of axioms.

We will take a tutorial-like approach as we walk through parts of our MOA specifications.

5.2 Fundamentals

We begin by specifying that a semigroup is a type and an associative operation on that type.

```
concept Semigroup = {
  type S;
  function op( a:S, b:S ):S;
  axiom associativeAxiom( a:S, b:S, c:S ) {
    assert op(op(a,b),c) == op(a,op(b,c));
  }
};
```

Here we read the declaration **type** s; as declaring that any implementation implementing this concept must provide a type named S. Furthermore a function op, and a function in the pure mathemathical sense, must be provided such that it takes two values of type S and results in a type S when applied. The axiom declaration **axiom** associativeAxiom(a:S, b:S, c:S) can be interpreted as a collection of assertions that must hold for all values of parameters passed to it. Here as mentioned before we may choose to use this as a way of rejecting an implementation of the concept as non-conforming to the specification. Another interpretation is to view axioms asserting truths such that in any expression, in an implementation conforming to the semigroup concept, we may substitute op(op(a,b),c) with op(a,op(b,c)) and get an equivalent implementation.

There is a re-use or mixin mechanism in Magnolia concepts, which we will use to specify a monoid. A monoid is a semigroup that has an identity element.

```
concept Monoid = {
  use Semigroup [S => M];
  function identity():M;
  axiom identityAxiom( a:M ){
    assert op(identity(),a) == a;
    assert op(a,identity()) == a;
  }
};
```

Here **use** Semigroup [S =>M]; does two things. It will perform the renaming [S =>M], i.e. replace the symbol name S with the symbol name M, on every declaration in Semigroup. Then the inclusion of **use** ... is to perform the union of the declarations in Semigroup [S =>M] and Monoid. Thus the complete expansion after the union is:

```
concept Monoid = {
  type M;
  function op ( a:M, b:M ):M;
  function identity():M;
  axiom associativeAxiom( a:M, b:M, c:M ) {
    assert op(op(a,b),c) == op(a,op(b,c));
  }
  axiom identityAxiom( a:M ){
    assert op(identity(),a) == a;
    assert op(a,identity()) == a;
  }
};
```

Where any ordering of declarations are all equivalent. In our MOA specification we found that the psi indexing was a partial right action:

```
concept PartialRightAction = {
  use Monoid;
  type S;
  //predicate such that defined(m,s)
  //holds when action(m,s) is defined
  predicate defined( m:M, s:S );
  function action( m:M, s:S ):S guard defined(m,s);
  axiom ActionAxiom( m1:M, m2:M, s:S )
  guard defined(m1,s) && defined(m2,s) {
    assert action( op(m1,m2), s ) ==
    action( m2, action(m1,s) );
  }
  axiom identityAxiom( s:S ) {
    assert action( identity(), s ) == s;
  }
};
```

Here we see two new constructs introduced; the predicate and the guard. A predicate operates as expected; it either holds or does not hold depending on its parameters. A guard expression following a function has the effect that the predicate expression that follows the guard statement must hold in order for the function to be applied. A guard on an axiom similarly state that the axiom applies when the guard expression holds. For functions the guard expression may be seen as dealing with partiality. For axioms we may consider an unguarded axiom such as:

```
axiom ActionAxiom( m1:M, m2:M, s:S ) {...}
```

as for all values m1:M, m2:M and s:S the assertions in {...} hold whereas

```
axiom ActionAxiom( m1:M, m2:M, s:S )
guard defined(m1,s) && defined(m2,s) {...}
```

is to be interpreted as for all m1:M, m2:M and s:S where defined(m1,s) and defined(m2,s) hold then the assertions in {...} hold.

5.3 Building our building blocks

Now we have couple of specifications we can combine and use, but we need some more. In Magnolia there is a large amount of concepts in its standard library, but we will not rely on them in an effort to have this be a stand-alone effort. This naturally has the effect of certain types being underspecified, but they shall suffice for our use. First is a non-negative integer specification that should be substituted with a fully specified non-negative integer specification.

```
concept Int = {
  predicate _<_( a:Int, b:Int );
  predicate _<=_( a:Int, b:Int );

  //Gives us zero and addition
  use Monoid [M => Int, identity => zero, op => _+_ ];

  //Gives us one and multiplication
  use Monoid [M => Int, identity => one, op => _*_ ];

  function _-_( a:Int, b:Int ):Int guard b <= a;
  function min( a:Int, b:Int ):Int;

  axiom minAxiom( a:Int, b:Int ) {
    assert (a <= b) => min(a,b) == a;
    assert (b < a) => min(a,b) == b;
  }

  // l < u && l <= i < u, sometimes written i in [l,u)
  predicate inRange( i:Int, l:Int, u:Int );

  // l <= u && l <= i <= u, sometimes written i in [l,u]
  predicate inRangeInclusive( i:Int, l:Int, u:Int );

  axiom inRangeAxiom
  ( i:Int, lowerInclusive:Int, upper:Int )
  guard lowerInclusive < upper {
    assert inRange( i,lowerInclusive, upper ) <=>
    ( lowerInclusive <= i ) && ( i < upper );
  }
  axiom inRangeInclusiveAxiom
  ( i:Int, lowerInclusive:Int, upperInclusive:Int )
  guard lowerInclusive <= upperInclusive {
    assert inRangeInclusive(i,lowerInclusive,upperInclusive)
    <=> ( lowerInclusive <= i ) && ( i <= upperInclusive );
  }
};
```

We have intentionally underspecified the Int type as describing non-negative integers is not the main focus here. We turn our focus to the vector which is fundamental in further specification.

```
concept Vector = {
  use Int;
  //The Vector type
  type V;
  //The element type
  type E;

  function singleton( e:E ):V;
  function size( v:V ):Int;
  function get( i:Int, v:V ):E guard i < size(v);
  function take( i:Int, v:V ):V;
  function drop( i:Int, v:V):V;
  // Here cat is short for conCATenate
  use Monoid[M => V, op => cat, identity => empty];
```

That is the function and type declaration part of our needed vector type. We consider them self-explanatory however someone not as biased would need to read documentation to understand these operations. However documentation can have wildly varying formalisms for explaining what is going on. In Magnolia with axioms we have one consistent way, we proceed to specify our functions:

```
  axiom emptySizeAxiom() {
    assert size( empty() ) == zero();
  }
  axiom singletonSizeAxiom(e:E) {
    assert size( singleton(e) ) == one();
  }
  axiom getsingletonAxiom( e:E ) {
    assert get( zero(), singleton(e)) == e;
  }
```

```
axiom VectorEmptyeqAxiom() {
  assert empty() == empty();
}
axiom VectorNonemptyeq(i:Int, l:V, r:V)
guard size(l) != zero() && size(l) == size(r) &&
      inrange(i,zero(),size(l)) {
  assert get(i,l) == get(i,r);
}
axiom takeSizeAxiom( i:Int, v:V ) {
  assert size(take(i,v)) == min(i,size(v));
}
axiom dropSizeAxiom( i:Int, v:V ) {
  assert size(drop(i,v)) == size(v)-min(i,size(v));
}
axiom takeEmptyAxiom( i:Int, v:V )
guard size(take(i,v)) == zero() {
  assert take(i,v) == empty();
}
axiom dropEmptyAxiom( i:Int, v:V )
guard size(drop(i,v)) == zero() {
  assert drop(i,v) == empty();
}
axiom takeNonEmptyAxiom( i:Int, j:Int, v:V )
guard size(take(i,v)) != zero() &&
inRange(j,zero(),size(take(i,v))) {
  var taken = take(i,v);
  assert get(j,taken) == get(j,v);
}
axiom dropNonEmptyAxiom( i:Int, j:Int, v:V )
guard size(drop(i,v)) != zero() &&
inRange(j,zero(),size(drop(i,v))) {
  var dropped = drop(i,v);
  assert get(j,dropped) == get(j+i,v);
}
axiom catSizeAxiom( v1:V, v2:V ) {
  assert size(cat(v1,v2)) == size(v1)+size(v2);
}
axiom catAxiom( i:Int, v1:V, v2:V )
guard v1 != empty() && v2 != empty() &&
inRange(i,zero(),size(cat(v1,v2))) {
  var v1v2 = cat(v1,v2);
  assert inRange(i,zero(),size(v1)) =>
        get(i,v1v2) == get(i,v2);
```

```
    assert inRange(i,size(v1),size(v1)+size(v2)) =>
           get(i,v1v2) == get(i-size(v1),v2);
  }
  //This axiom contains enough to
  //prove the associativity axiom
  axiom takedropcatAxiom( k:Int, v1:V, v2:V )
  guard zero() <= k && k <= size(v1) {
    assert take(k,cat(v1,v2)) == take(k,v1);
    assert drop(k,cat(v1,v2)) == cat(drop(k,v1),v2);
    assert cat(take(k,v1),drop(k.v1)) == v1;
  }
};
```

Recall in MOA we used the notion of lifting a relations such that for example $x <^* \langle i_0, \ldots, i_{n-1}\rangle \Leftrightarrow x < i_0 \wedge \ldots \wedge x < i_{n-1}$ and $\langle i_0, \ldots, i_{n-1}\rangle <^* \langle s_0, \ldots, s_{n-1}\rangle \Leftrightarrow i_0 < s_0 \wedge \ldots \wedge i_{n-1} < s_{n-1}$ we specify that as a lifted relation.

```
concept LiftedRelations = {
  use Vector;

  predicate r( l:E, r:E );
  predicate pointwise( v1:V, v2:V );
  predicate all( e:E, v:V );

  axiom pointwiseEmptyAxiom() {
    assert pointwise(empty(),empty());
  }
  axiom pointwiseNonemptyAxiom( v1:V, v2:V, i:Int )
  guard size(v1) == size(v2) &&
        size(v1) != zero() &&
        inRange(i,zero(),size(v1)) {
    assert pointwise(v1,v2) => r(get(i,v1),get(i,v2));
  }
  axiom allEmptyAxiom( e:E ) {
    assert all(e,empty());
  }
  axiom allNonemptyAxiom( e:E, v:V, i:Int )
  guard size(v) != zero() && inRange(i,zero(),size(v)) {
    assert all(e,v) => r(e,get(i,v));
  }
};
```

Then we have an additional useful and recurring concept called a reduction which can be used to specify sums or products. Another common name for this concept is a right fold. It is the concept that given a binary function and an identity element we can perform a reduction over the vector.

```
concept Reduction = {
  use Vector;

  function binop( l:E, r:E ):E;
  function identity():E;
  function reduce( v:V ):E;

  axiom reductionAxiom( v:V ) {
    assert size(v) == zero() =>
           reduce(v) == identity();

    assert size(v) != zero() =>
           reduce(v) == binop( get(zero(),v),
                               reduce(drop(one(),v)) );
  }
};
```

We also will need to be able to apply a function over vector's elements. We introduce a map that applies a binary function over the vector that maps e to $g(v, e)$ where v is some supplied value and e is the old vector element.

```
concept BMap = {
  use Vector;

  function bop( e1:E, e2:E ):E;
  function bopmap( e:E, V:V ):V;

  axiom bmappedemptyAxiom( e:E ) {
    assert bopmap(e,empty()) == empty();
  }

  axiom bmappedNonemptyAxiom( i:Int, v:V, e:E )
  guard zero() < size(v) && inRange(i,zero(),size(v)) {
    assert size(bopmap(e,v)) == size(v);
    assert get(i,bopmap(e,v)) == bop(e,get(i,v));
  }
};
```

Finally we add the ability to generate a vector from a function. This is typically a very useful operation and will be used to specify the iota vector.

```
// Generating a vector with a function
concept FunctionToVector = {
  use Vector;

  function f( i:Int ):E;
  function make( size:Int ):Vector;

  axiom makeZeroAxiom() {
    assert make(zero()) == empty();
  }
  axiom makeAxiom( i:Int, size:Int )
  guard inRange(i,zero(),size) {
    assert get(i,make(size)) == f(i);
  }
};
```

5.4 Indices and shapes

Now we will use our vector concept to specify a vector of non-negative integers. This is needed for indices and shapes which are just vectors of integers. Here we will use certain additonaly constucts of Magnolia, namely the ability to name a renaming and use it several times.

```
renaming IntVectorR = [V => IntVector, E => Int];
concept IntVector = {
 use Vector [ IntVectorR ];
 // common reductions + and *
 use Reduction [ IntVectorR ]
                 [ reduce => sum,
                   identity => zero,
                   binop => _+_ ];
 use Reduction [ IntVectorR ]
               [ reduce => product,
                 identity => one,
                 binop => _*_ ];
 // common maps + and *
 use BMap [ IntVectorR ]
          [ bop => _+_,
           bopmap => _+_];
 use BMap [ IntVectorR ]
```

```
              [ bop => _*_,
               bopmap => _*_];
 use LiftedRelations [ IntVectorR ]
                       [ r => _<_,
                       pointwise => _<_,
                       all => _<_ ];
 use LiftedRelations [ IntVectorR ]
                       [ r => _<=_,
                       pointwise => _<=_,
                       all => _<=_ ];

};
```

Recall the iota function, which is an empty vector for a parameter of zero or can be generated by the identity function for parameters above zero.

```
concept Iota = {
  use FunctionToVector [ IntVectorR ]
                       [ make => iota,
                         f => id ];
  axiom id( i:Int ) {
    assert id(i) == i;
  }
};
```

It will be useful to specify permutations of the iota vector. In particular this is a means of checking that a gamma function given a shape and its mapping, collected in a vector, of all possible indices for that shape results in a vector that is a permutation of the iota vector for the size of the index space determined by the shape.

```
concept Permutation = {
  use Int;
  use IntVector;
  use Iota;

   // for any element i in the vector i < size(v)
  predicate bounded( v:IntVector );

  axiom boundedEmptyAxiom() {
    assert bounded(empty());
  }
  axiom boundedNonemptyAxiom( i:Int, v:IntVector )
  guard inRange(i,zero(), size(v)) && zero() < size(v) {
    assert bounded(v) => get(i,v) < size(v);
  }


  //This function, taking v such that any element e in v
  //is such that e < size(v), must first perform
  //the assignment b[i] = size(v) for 0 <= i < size(v) and
  //then the assignment b[v[i]] = v[i] for 0 <= i < size(v)
  //thus the postcondition is that for any 0 <= i < size(v)
  //b[i] <= size(v) and b is the result of reorder
  //
  //If any b[i] == size(v) at least two elements in v were
  //equal. If not all were distinct and b is now iota
  function reorder( v:IntVector ):IntVector
  guard bounded(v);

  axiom reorderEmptyAxiom() {
    assert reorder( empty() ) == empty();
  }

  axiom reorderNonEmptyAxiom( i:Int, v:IntVector )
  guard inRange(i,zero(),size(v)) && bounded(v) {
    assert get( get(i,v), reorder(v) ) == get(i,v);
  }
  axiom reorderValuesNonEmptyAxiom( i:Int, v:IntVector )
  guard inRange(i,zero(),size(v)) && bounded(v) {
    assert get(i, reorder(v) ) <= size(v);
  }

  //predicate that holds if v is
  //a permutation of the iota vector
  predicate permutation( v:IntVector );

  axiom permutationAxiom( p:IntVector ) guard bounded(v) {
    assert iota(size(p)) == reorder(p);
  }
};
```

Now we have enough to cover the gamma function and related.

```
concept Gamma = {
  use IntVector;
  use Iota;
  function gamma( i:IntVector, s:IntVector ):Int;
  function stride( s:IntVector, k:Int ):Int;
  function start( i:IntVector, s:IntVector, k:Int ):Int;

  axiom gamma( i:IntVector, s:IntVector )
  guard size(i) == size(s) && i < s {
    var size = size(i);
    assert zero() == size =>
           gamma(i,s) == zero();

    assert one() == size =>
           gamma(i,s) == get(zero(),i);

    assert  one() <  size =>
            gamma(i,s) == get(size-one(),i) +
            get(last,i)*gamma( take(size-one(),i),
                               drop(size-one(),i) );
   }

  axiom strideStartGammaAxiom
  ( k:Int, i:IntVector, s:IntVector )
  guard size(i) == size(s) && i < s &&
  inRangeInclusive(k,zero(),size(s))  {
    assert stride(s,k) == product(drop(k,s));

    assert start(i,s,k) == gamma( take(k,i), take(k,s) );

    assert gamma( cat( take(k,i), drop(k,i) ), s ) ==
           start(i,s,k) * stride(s,k) + gamma( drop(k,i),
                                               drop(k,s) );
  }

  function section( i:IntVector, s:IntVector ):IntVector
  guard size(i) <= size(s) && i < take(size(i),s);

  axiom sectionAxiom( i:IntVector, s:IntVector )
  guard size(i) <= size(s) && i < take(size(i),s) {
    var offsets = iota(product(drop(k,s)));
    var start = start(i,s,size(i));
    assert section(i,s) == start + offsets;
  }
};
```

5.5 MOA in Magnolia

Finally we have arrived at a point where we can specify MOA in Magnolia. Here we will use most of our concepts so far. However to make our specification independent of a fixed bijection between multidimensional indices and unidimensional indices we will only specify that we need a bijection. The strategy here is to specify that our gamma function mapped over all multidimensional indices must result in the iota vector when reordered.

```
concept GammaMap = {
  use IntVector;
  use Vector [ E => IntVector, V => Indices ];

  function gamma( i:IntVector, s:IntVector ):Int;

  //The collection of gamma(i,s) with i from ix
  function gammaMap( ix:Indices, s:IntVector ):IntVector;

  axiom emptyAxiom( s:IntVector ) {
    assert gammaMap(empty(),s) == empty();
  }

  axiom nonEmptyAxiom( i:Int, ix:Indices, s:IntVector )
  guard zero() < size(ix) && inRange(i,zero(),size(is)) {
    assert size(gammaMap(ix,s)) == size(ix);
    assert get(i,gammaMap(ix,s)) == gamma(get(i,ix),s);
  }
};
```

With concepts we can partition MOA into the fundamental definitions and operations. First we specify the fundamental definitions of MOA.

```
concept MOA = {
  use Int;
  use Iota;

  //The array type
  type A;

  //The element type
  type E;

  //The shape and any index of an array is a Vector of Int
  use IntVector;
```

```
//Projection to get the defining properties of an array
function dimensions( a:A ):Int;
function shape( a:A ):IntVector;
function size( a:A ):Int;

axiom sizeAxiom( a:A ) {
  assert size(a) == product(shape(a));
}

predicate scalarArray( a:A );
predicate vectorArray( a:A );
predicate multiArray( a:A );

axiom arrayClassificationAxiom( a:A ) {
  assert zero() == dimensions(a) <=> scalarArray(a);
  assert one()  == dimensions(a) <=> vectorArray(a);
  assert one()  <  dimensions(a) <=> multiArray(a);
}

//Scalar projection of scalar array
function toScalar( a:A ):E guard scalarArray(a);
function fromScalar( e:E ):A;
axiom scalarProjectionAxiom ( e:E ) {
  assert toScalar( fromScalar(e) );
}

//Vector projection of vector array
function toVector( a:A ):Vector guard vectorArray(a);
function fromVector( v:Vector ):A;
axiom projectionAxiom ( v:Vector ) {
  assert toVector( fromVector(v) ) == v;
}
//Array selection with psi
use PartialRightAction [ M       => IntVector,
                         S       => A,
                         op      => cat,
                         action  => psi,
                         defined => validIndex ];

predicate inShape( i:IntVector, s:IntVector );
axiom inShapeAxiom( i:IntVector, s:IntVector ) {
  assert inShape(i,s) <=>
```

```
          size(i) == size(s) &&  i < s;
}

predicate fullIndex( i:IntVector, a:A );
axiom fullIndexAxiom( i:IntVector, a:A ) {
  assert fullIndex(i,a) <=>
          size(i) == dimensions(a) &&
          inShape(i,shape(a));
}

predicate partialIndex( i:IntVector, a:A );
axiom partialIndexAxiom( i:IntVector, a:A ) {
  assert partialIndex(i,a) <=>
          size(i) < dimensions(a) &&
          inShape(i,take(size(i),shape(a)));
}

//We specify the meaning of a valid index
//for which psi is defined
axiom validIndexAxiom( i:IntVector, a:A ) {
  assert validIndex(i,a) <=>
          fullIndex(i,a) || partialIndex(i,a);
}

axiom psiIndexingShapeAxiom( i:IntVector, a:A )
guard validIndex(i,a) {
  assert shape(psi(i,a)) == drop( size(i), shape(a) );
}

function rav( a:A ):A;
axiom ravDimensionsAxiom( a:A ) {
  assert dimensions( rav(a) ) == one();
}
axiom ravScalarAxiom( a:A ) guard scalarArray(a) {
  assert toVector( rav(a) ) == singleton(toScalar(a));
}
axiom ravVectorAxiom( a:A ) guard vectorArray(a) {
  assert toVector( rav(a) ) == toVector(a);
}

//Given a shape produce all full indices in the shape
function indices( s:IntVector ):Indices;
axiom indicesSizeAxiom ( s:IntVector ) {
    assert size( indices(s) ) == product(s);
  }
  axiom zeroIndicesAxiom( s:IntVector )
  guard product(s) == zero() {
    assert indices(s) == empty();
  }
  axiom oneIndicesAxiom( s:IntVector )
  guard product(s) == one() {
    assert get(zero(),indices(s)) == empty();
  }
  axiom indicesUniqueAxiom( i:Int, j:Int, s:IntVector )
  guard one() < product(s) && i != j &&
  inRange(i,zero(),product(s)) &&
  inRange(i,zero(),product(s)) {
    assert get(i,indices(s)) != get(j,indices(s));
  }

  //This will introduce the gamma function
  use GammaMap;
  //We require that it is a bijection
  //between multiindices and uniindices
  axiom bjectiveGammaAxiom ( s:IntVector ) {
    assert iota(product(s)) ==
           reorder(gammaMap(indices(s),s));
  }

  axiom ravAnyAxiom( i:IntVector, a:A )
  guard fullIndex(i,a) {
    assert toScalar( psi(i,a) ) ==
           get( gamma(i,shape(a)), toVector(rav(a)));
  }
};
```

We have now have the fundamental definitions of MOA in a concept. Notice we have not specified any operations. These can be defind piecewise as needed. As an example we will specify the outer product from MOA that corresponds to computing the Kronecker product. It is not indexed identically(as a 2d-matrix) as it preserves sub-matrices. In brief the Kronecker product can be described given matrices A and B:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \; B = \begin{bmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

then the Kronecker product is:

$$A \otimes B = \begin{bmatrix} 1 \times B & 2 \times B \\ 3 \times B & 4 \times B \end{bmatrix}$$

Where we define $\times$ as some binary operation:

$$x \times B = \begin{matrix} x \times 5 & x \times 6 & x \times 7 & x \times 8 \\ x \times 9 & x \times 10 & x \times 11 & x \times 12 \\ x \times 13 & x \times 14 & x \times 15 & x \times 16 \end{matrix}$$

Notice with the matrix formulation it becomes hard to refer to the sub-matrices of the resulting computation. This is information that is preserved in the MOA formulation, this is a topic that is explored further in [16] where the inspiration for this example is from. We now model this in MOA as an example of specification with MOA in Magnolia.

```
concept OuterProduct = {
  use MOA;
  function op( e1:E, e2:E ):E;
  function opM( e:E, a:A ):A;

  axiom opMAxiom( e:E, a:A, i:IntVector )
  guard inShape(i,a) {
    assert shape( opM(e,a) ) == shape(a);
    assert psi( i, opM(e,a) ) == op( e, psi(i,a) );
  }

  function two():Int;
  axiom numbersAxiom() {
    assert two() == one()+one();
  }

  function outer( a:A, b:A ):a
  guard dimensions(a) == two() &&
         dimensions(b) == two();

  axiom outerShapeAxiom( a:A, b:A ) guard
  dimensions(a) == two() && dimensions(b) == two() {
    assert shape( outer(a,b) ) == cat( shape(a),
                                       shape(b) );
  }
```

As can be seen we first describe the operations we need for the product. That opM(which corresponds to $\times$) is op with a provided element applied over an array. We describe needed shapes for the outer operation and describe the resulting shape in terms of the shapes of its parameters. We then describe the change in values of the resulting array of outer.

```
  axiom outerValuesAxiom( a:A, b:A, ijkl:IntVector )
  guard dimensions(a) == two() && dimensions(b) == two() &&
        inShape( ijkl , shape( outer(a,b) ) ) {
    var ij = take(two(),ijkl);
    var kl = drop(two(),ijkl);
    assert psi(ijkl , outer(a,b)) ==
            psi(kl, opM( psi(ij ,a), b));
  }
};
```

This completes our specification of an outer product and is an example of MOA put to use. The value produced here is that we have a layout independent description of an operation for which we can supply any given bijective gamma function and experiment with different memory layouts of the operation. Every operation in MOA is described in this fashion.

5.6 Summary

We have described the fundamental parts of MOA in Magnolia based on the simplified MOA described in the former chapter. We have focused on a limited subset of what is available in Magnolia and we have opted to specify from scratch. Many concepts are already in Magnolia, but it is not as instructive to simply use them here. In a specification meant for inclusion in Magnolia's libraries we would rely on many of its specifications, among them likely a better integer specification, which in this treatment is likely underspecified. The specification of operations so far are concerned with the "what" and the "how" at the logical level where arrays are described in terms of shapes and values with respect to layout independent abstract indexing, or in other words arrays described in terms of their Denotational Normal Form (DNF). When a layout is selected and abstract indexing is translated to concrete indexing we have description of arrays and operations in their Operational Normal Form (ONF). We have now considered the former(DNF) to completion and will move on to the latter(ONF), but we first take a break from MOA to consider other array abstractions and libraries.

CHAPTER 6

Other array abstractions and libraries

6.1 introduction

As we saw in the array programming chapter there are important performance considerations when a programmer is given an array type and is free to choose the iteration order over its index space. One might argue that the basic constructs offered in C and the sligthly more involved constructs in Fortran devolve into little more than a thin veil over flat access to memory for such considerations. Certainly what is to be expressed in terms of computation cannot be the only concern; the order of computation must be an important concern for performant computations.

In this chapter we will review array libraries that raise the level of abstraction for array programming. These can be considered as Domain Specific Language (DSL)s specifically for array programming. These abstracted views of arrays can be quite involved and we cannot delve into all facets of them, but we pick some and attempt to convey their conceptual essence, theoretical underpinnings and usefulness in abstraction.

6.2 Petalisp and array operations

Petalisp [7], is a Common Lisp library that attempts to generate high performance parallel array computations with a Just In Time Compiler (JIT) for its array operations.

It is conceptually simple with only a single data structure which is the strided array and four operations on it. We will attempt to avoid language details and instead focus on the underlying concepts of the library. The focus will be to explain the primitives given and consider their expressiveness.

6.2.1 The primitives of Petalisp

An array in Petalisp need not have a contiguous index space. That is an index space of $\{0, 2, 4, 6\}$ is perfectly valid and would be expressed by $\mathit{range}(0, 6, 2)$. Multidimensional indices are expressed through several such ranges. We begin by defining the concept of a range and then the Petalisp Array:

- A range is a space $R \subset \mathbb{Z}$ where $x_l, x_u, s \in \mathbb{Z}$ and $x_l < x_u$ given by:

  $$\mathit{range}(x_l, x_u, s) = \{x \in \mathbb{Z} \mid x_l \leq x \leq x_u \wedge \exists(k \in \mathbb{Z})[x = x_l + ks]\}$$

  from which we summarize that:

  - x_l is the inclusive lower bound
  - x_u is the inclusive upper bound
  - s is the stride[1]

  such that:

  $$\begin{aligned}\mathit{range}(x_l, x_u, s) &= \{x_l + ks \mid k \in \{0, 1, \dots, n\} \wedge x_l \leq x_l + ks \leq x_u\} \\ &= \{x_l = x_l + 0s, x_l + 1s, \dots, x_l + ns = x_u\}\end{aligned}$$

  remark: As x_u is inclusive $\mathit{range}(0, 5, 2)$ is an invalid range whereas $\mathit{range}(0, 4, 2)$ is a valid range. The size of the range n is not given but can be computed from the constraints as $x_u = x_l + ns$ yields $n = \frac{x_u - x_l}{s}$.

- A strided array of n dimensions is a function $f : R_1 \times \cdots \times R_n \to O$ where O is a set of objects and R_i are defined by ranges. We will refer to the domain of f and its index space interchangably.

[1]take care not to confuse this use of the word stride with former use of the word stride

Petalisp uses the terminology of an array having a shape and a list of ranges is Petalisp's representation for the shape of an array [7, p. 13]. That is if an array has a domain determined by $\mathit{range}(l_1, u_1, s_1) \times \cdots \times \mathit{range}(l_n, u_n, s_n)$ then its shape is the list or tuple $\langle\langle l_1, u_1, s_1\rangle, \ldots, \langle l_n, u_n, s_n\rangle\rangle$.

We shall use Petalisp's definition of a shape for our review of Petalisp, but however note that we could have stuck to the definition of a shape we already have some familiarity with by letting an index space defined by:

$$\mathit{range}(l_1, u_1, s_1) \times \cdots \times \mathit{range}(l_n, u_n, s_n)$$

have the shape:

$$\langle \frac{u_1 - l_l}{s_1} + 1, \ldots, \frac{u_n - l_n}{s_n} + 1 \rangle$$

and instead speak of arrays with equivalent domains or index spaces, determined by a list of ranges, as in the case of equivalent shapess only equally sized index spaces can be inferred.

For the domain of strided arrays we shorten $R_1 \times \cdots \times R_n$ to $R_{1,\ldots,n}$ onwards. Again note the possibility of expressing discontigous index spaces. Let $s = \langle\langle 0, 10, 5\rangle, \langle 0, 3, 1\rangle\rangle$. The index space for a strided array with shape s defining its domain is given by $\{0, 5, 10\} \times \{0, 1, 2, 3\}$. As seen here the first index position for example "strides" from 0 to 5.

Petalisp mainly consists of four operations on these strided arrays:

- **Application:** $c = \mathit{apply}(g, a, b)$ denotes an array where g is binary function over objects $g : O \times O \to O$ and a, b are arrays of identical shape such that if i is a valid index then $c(i) = f(a(i), b(i))$.

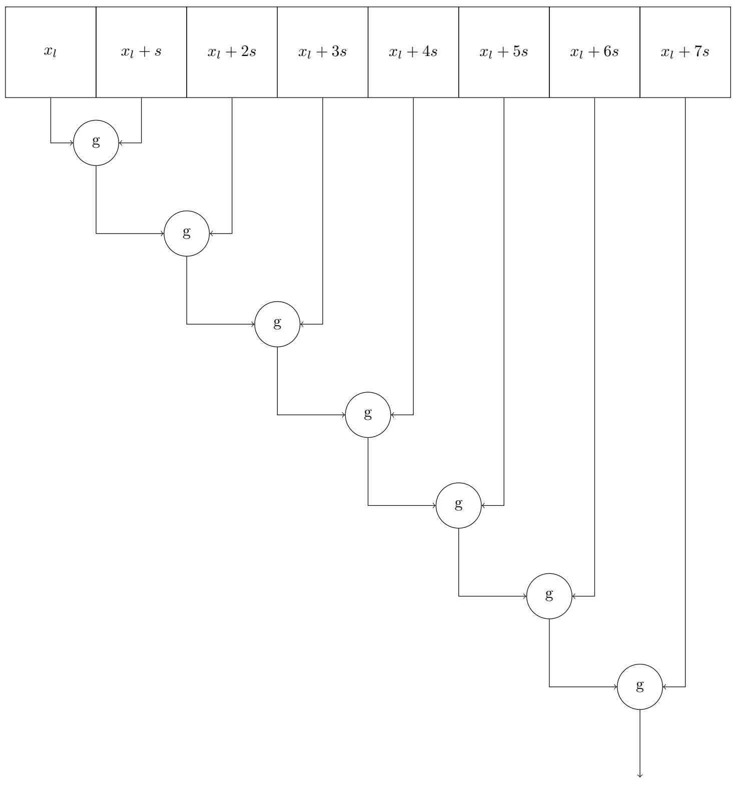
  Example: The elementwise sum of two arrays $\mathit{apply}(+, a, b)$.

- **Reduction:** $b = \mathit{reduce}(g, a)$ denotes either an object O when a is 1-dimensional or an $n - 1$ dimensional array of index space $R_{1,\ldots,n-1}$ that is the result of reducing an n dimensional array a of index space $R_{1,\ldots,n}$ along the last axis with a binary function $g : O \times O \to O$.

  We define reduction for an array a with index space $R_{1,\ldots,n}$ by cases $n = 1$ and $n > 1$:

  Remark: The reduction operator is largely modelled after the Common Lisp reduction function. We may deviate slightly from its exact details in that we specify that the binary operator g must be commutative and associated with an identity element s.t. $\mathit{identity}(g) = e$ and $g(e, k) = g(k, e) = k$.

Figure 6.1: reduce with a function g



For the case when $n = 1$ and a has domain defined by $range(x_l, u_l, s)$, the result is an object:

$$reduce(g, a) = foldr(identity(g), x_l)$$
$$foldr(r, i) = \begin{cases} foldr(g(r, i), i + s) & i \leq x_u \\ r & \text{otherwise} \end{cases}$$

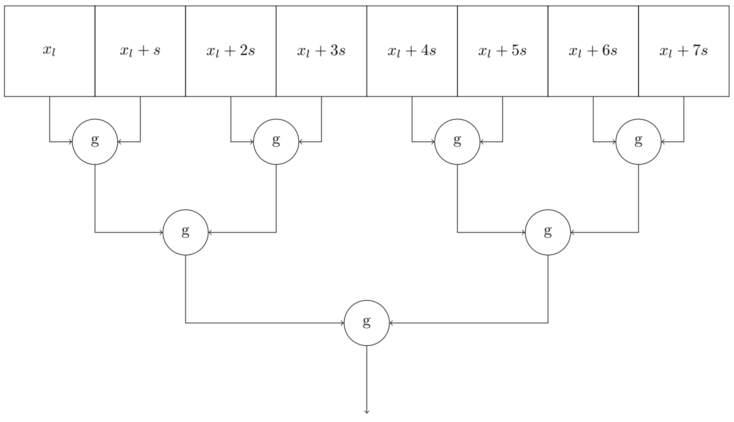
see figure 6.1 if this is unclear. Note that if g is associative the operation can be reordered as seen in figure 6.2.

For the case when $n > 1$ and a has domain $R_{1,\ldots,n}$ where $R_n = range(x_l, x_u, s)$, the result is an array with domain $R_{1,\ldots,n-1}$ such that:

$$(reduce(g, a))(i_1, \ldots, i_{n-1}) = reduce(g, \{k(x) = a(i_1, \ldots, i_{n-1}, x) \mid x \in R_n\})$$

Example: Consider an array a with a shape of $\langle\langle 1, 3, 1\rangle, \langle 5, 7, 1\rangle\rangle$, i.e.

Figure 6.2: reduce with an associative function g



a has domain $\{1, 2, 3\} \times \{5, 6, 7\}$

$$\begin{aligned} a = \{&\langle 1,5\rangle \mapsto 1, \langle 2,5\rangle \mapsto 2, \langle 3,5\rangle \mapsto 3,\\ &\langle 1,6\rangle \mapsto 4, \langle 2,6\rangle \mapsto 5, \langle 3,6\rangle \mapsto 6,\\ &\langle 1,7\rangle \mapsto 7, \langle 2,7\rangle \mapsto 8, \langle 3,7\rangle \mapsto 9\ \} \end{aligned}$$

then we have for $R_2 = \mathit{range}(5, 7, 1)$:

$$\begin{aligned} \mathit{reduce}(+, a) = \{&1 \mapsto \mathit{reduce}(+, \{k(x) = a(1, x)|x \in R_2\}),\\ &2 \mapsto \mathit{reduce}(+, \{k(x) = a(2, x)|x \in R_2\}),\\ &3 \mapsto \mathit{reduce}(+, \{k(x) = a(3, x)|x \in R_2\})\ \} \end{aligned}$$

which results in

$$\begin{aligned} \mathit{reduce}(+, a)(i) = \{&1 \mapsto 0 + 1 + 4 + 7 = 12,\\ &2 \mapsto 0 + 2 + 5 + 8 = 15,\\ &3 \mapsto 0 + 3 + 6 + 9 = 18\ \} \end{aligned}$$

We may regard this example as the sum of the rows in the 3×3 matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

For an array representing a cube of elements the reduction with addition would be a bottom plane where each point in that plane is the sum of the elements intersecting a normal line starting there in the cube.

- **Fusion:** Given two arrays a, b then $\mathit{fuse}(a, b)$ is defined when the arrays have an index space, where a's is $R^a_{1,\ldots,n}$ and b's is $R^b_{1,\ldots,n}$, such that given any $R^a_i = \mathit{range}(a_l, a_u, s)$ and $R^b_i = \mathit{range}(b_l, b_u, s)$ then $b_l = a_u + s$, i.e. a range of b at dimension i starts one s stride from where a range of a at dimension i ends.

  The index space of $\mathit{fuse}(a, b)$ is defined as $R^{\mathit{fuse}(a,b)}_{1,\ldots,n}$ such that $R^{\mathit{fuse}(a,b)}_i = \mathit{range}(a_l, b_u, s)$ where $R^a_i = \mathit{range}(a_l, a_u, s)$ and $R^b_i = \mathit{range}(b_l, b_u, s)$. The values of $\mathit{fuse}(a, b)$ is defined as:

$$\mathit{fuse}(a, b)(i_1, \ldots, i_n) = \begin{cases} a(i_1, \ldots, i_n) \text{ when } \langle i_1, \ldots, i_n \rangle \in R^a_{1,\ldots,n} \\ b(i_1, \ldots, i_n) \text{ when } \langle i_1, \ldots, i_n \rangle \in R^b_{1,\ldots,n} \end{cases}$$

  In effect $\mathit{fuse}(a, b) = a \cup b$. It's clearly the case that fusion is associative(by associativity of union) and thus we can collapse big fusions together and write $\mathit{fuse}(a_1, a_2, \ldots, a_n) = a_1 \cup \ldots \cup a_n$.

  Example: Let the index space of $a(x, y) = x$ be defined by the shape $\langle\langle 1, 3, 1\rangle, \langle 1, 3, 1\rangle\rangle$ and of $b(x, y) = x$ by the shape $\langle\langle 4, 6, 1\rangle, \langle 4, 6, 1\rangle\rangle$. The condition to fuse these two arrays together are met and thus $\mathit{fuse}(a, b)$ is equivalent to the array defined by

$$\mathit{fuse}(a, b)(x, y) = \begin{cases} a(x, y) \text{ when } \langle x, y \rangle \in \mathit{range}(1, 3, 1) \times \mathit{range}(1, 3, 1) \\ b(x, y) \text{ when } \langle x, y \rangle \in \mathit{range}(4, 6, 1) \times \mathit{range}(4, 6, 1) \end{cases}$$

  which we can consider to represent the matrix

$$\begin{bmatrix} 1 & 1 & 1 & 4 & 4 & 4 \\ 2 & 2 & 2 & 5 & 5 & 5 \\ 3 & 3 & 3 & 6 & 6 & 6 \end{bmatrix}$$

  The requirement that one fuse together arrays that don't overlap is likely because it allows for either a very simple implementation of copying non-overlapping elements from two arrays to a new array or as we shall consider further that we may create a structure that satisfies an array-like interface, but that really dispatches indexing to a referenced array. By making arrays resulting from fusion(s) outwardly keep a singular range while internally maintaining its subdivisions into constituent ranges, selecting which referenced array an index must be from is possible in an efficient manner. Notice that in order to select which underlying array an index must be from it is sufficient to discriminate into which range a single index component falls; a consequence of the non-overlap.

Consider a fused array that is the result of many fusions, i.e.

$$\mathit{fuse}(a_1, \ldots, a_m)$$

where $\mathit{fuse}(a_1, \ldots, a_m)$ has an index space of $R_{1,\ldots,n}$ as do any a_i. Along the kth dimension $\mathit{fuse}(a_1, \ldots, a_m)$ has the range $R_k = \mathit{range}(x_l, x_u, s)$ and this range can be subdivided into each of the ranges for the arrays a_i. Consider the upper bounds for each of these ranges $\langle u_1, \ldots, u_m\rangle$ s.t. u_i is the upper bound for a_i. Recall that every bound u_i is expressible as $u_i = x_l + k_i s$ so the sequence $\langle u_1, u_2, \ldots, u_m$ may be considered as $\langle k_1, k_2, \ldots, k_m\rangle$ where $k_i = \frac{u_i - x_l}{s}$.

Determining which range an index's kth component $x = x_l + k_x s$ belongs in, given that it does belong in the fused range, can be determined by finding where k_x belongs in the sequence $\langle k_1, k_2, \ldots, k_m\rangle$, i.e. if it belongs below k_1 then it is an index into a_1, if it below in k_2 but below k_2 then it is an index into a_2 and so on. This can computed efficiently by $\mathit{subrange}(k_1, k_2, \ldots, k_{m-1})$ which is detailed in appendix D.

There also is a variation of fusion $\mathit{fuse}^*(a, b)$ in Petalisp that relaxes the requirement of non-overlap. In that case dispatch to an array is done by first checking the rightmost argument or in a long chain the innermost and rightmost.

- **Reference:** Given an array a with index space $R_{1,\ldots,n}$, a shape s and either an affine-linear transformation $T : R_{1,\ldots,n} \to R'_{1,\ldots,m}$ or a non-invertible broadcasting function $F : R'_{1,\ldots,m} \to R'_{1,\ldots,n}$ we define that $\mathit{reshape}(a, s, T)$ denotes an array b with index space $R'_{1,\ldots,m}$ determined by the shape s and its values determined by the affine-linear transformation:

$$b(i_1, \ldots, i_m) = a(j_1, \ldots, j_n) \text{ where } T(j_1, \ldots, j_n) = \langle i_1, \ldots, i_m\rangle.$$

or the case of a non-invertible broadcasting function $\mathit{reshape}(a, s, F)$

$$b(i_1, \ldots, i_m) = a(j_1, \ldots, j_n) \text{ where } F(i_1, \ldots, i_n) = \langle j_1, \ldots, j_m\rangle.$$

An affine-linear transformation can be summarized as a mapping that preserves lines, more details are given in C. Petalisp is able to compute the inverse $T^{-1} : R_{1,\ldots,m} \to R_{1,\ldots,n}$.

Example: Given the array $a(x) = x$ for $1 \leq x \leq 5$ and the affine-linear transformation $T(x) = x - 5$ then $\mathit{reshape}(a, \langle 6, 10, 1\rangle), T)$ is the array $b(x) = a(T(x))$ for $6 \leq x \leq 10$

Remark: Consider nested reshapings:

$$b = \mathit{reshape}(\mathit{reshape}(a, s_1, T_1), s_2, T_2)$$

Then the values $b(j_1, \dots, j_m)$ are given by $a(i_1, \dots, i_n)$ where[2]

$$T_1^{-1} \circ T_2^{-1}(j_1, \dots, j_m) = \langle i_1, \dots, i_n \rangle$$

and we have the identity

$$\mathit{reshape}(\mathit{reshape}(a, s_1, T_1), s_2, T_2) = \mathit{reshape}(a, s_2, T_2 \circ T_1)$$

In the case that we have a non-invertible broadcasting function

$$\mathit{reshape}(\mathit{reshape}(a, s_1, T), s_2, F)$$

We can still find its values by indices given by $T^{-1} \circ F$, but can no longer collapse the reshaping.

The manner in which one actually creates new arrays is by using *reshape* and *fuse*. A special variant of *reshape* that breaks the rules that we will call *reshape** also exists that lifts the requirement that the transformation is affine-linear and can be used on anything that can be acted upon with respect to indexing as a strided array; e.g. a Common Lisp function or a Common Lisp vector.

A final deviation is a variation of *apply* that we will call *apply** which lifts the requirement that all arrays must have identical shape. In the case that $\mathit{apply}^*(g, a, b)$ is such that the arrays have different shapes then an attempt is made to find a common index space for the arrays with two reshapings i.e. $a' = \mathit{reshape}(a, s, F_1)$ and $b' = \mathit{reshape}(b, s, F_2)$ such that $\mathit{apply}^*(g, a, b) = \mathit{apply}(g, a', b')$ and where the broadcasting functions repeat values along dimensions.

In particular if a has shape $\langle\langle 1, N, 1\rangle, \langle 1, 1, 1\rangle\rangle$ and b has shape $\langle\langle 1, 1, 1\rangle, \langle 1, M, 1\rangle\rangle$ they will be found to have a common index space of $\langle\langle 1, N, 1\rangle, \langle 1, M, 1\rangle\rangle$ when passed to *apply**. We will have $\mathit{apply}^*(g, a, b) = \mathit{apply}(g, a', b')$ where $a'(x, y) = a(x, 1)$ and $b'(x, y) = b(1, y)$. Repetition of values are thus substituted for missing values. This conceptually allows one to regard certain

[2]note $\circ$ means application order composition i.e. $(f \circ g)(x) = f(g(x))$

arrays, when using this operation, as dense representation arrays where *apply* otherwise would have required. We do not cover the particularities of this inference beyond the repetition of one value.

These operations in Petalisp only serve to create an expression that can be evaluated. The Petalisp strategy is to build an expression graph and at runtime analyse the dataflow of what is needed for a requested value and evaluate only as much computation as is needed to produce this value.

6.2.2 Matrix multiplication in Petalisp

Consider describing matrix multiplication with these primitives. We assume we have an $M \times N$ matrix represented by the strided array A_{MN} with the index space defined by a shape $\langle\langle 1, M, 1\rangle, \langle 1, N, 1\rangle\rangle$ and a $N \times K$ matrix represented by the strided array B_{NK} with index space defined by the shape $\langle\langle 1, N, 1\rangle, \langle 1, K, 1\rangle\rangle$.

In case the result we are after is not clear, we are looking for an $M \times K$ matrix which we will call C_{MK}, such that its values are as follows:

$$C_{MK}(m, k) = \sum_{i=1}^{n} A_{MN}(m, i) B_{NK}(i, k)$$

We consider first reshaping A_{MN} with the shape s_{M1N} to obtain A_{M1N} and B_{NK} with the shape s_{1KN} to obtain B_{1KN} where:

$$\begin{aligned} s_{M1N} &= \langle\langle 1, M, 1\rangle, \langle 1, 1, 1\rangle, \langle 1, N, 1\rangle\rangle \\ s_{1kN} &= \langle\langle 1, 1, 1\rangle, \langle 1, K, 1\rangle, \langle 1, N, 1\rangle\rangle \end{aligned}$$

The affine-linear transformations needed are

$$T_1(m, n) = \langle m, 1, n\rangle \text{ and } T_1^{-1}(m, k, n) = \langle m, n\rangle$$
$$T_2(n, k) = \langle 1, k, n\rangle \text{ and } T_2^{-1}(m, k, n) = \langle n, k\rangle$$

We may then define the reshapings as:

$$\begin{aligned} A_{M1N} &= \mathit{reshape}(A_{MN}, s_{M1N}, T_1) \\ B_{1KN} &= \mathit{reshape}(B_{NK}, s_{1KN}, T_2) \end{aligned}$$

We now consider the result of *apply** with multiplication on these:

$$C_{MKN} = \mathit{apply}^*(\times, A_{M1N}, B_{1KN})$$

As the shapes do not match an attempt is made to find a common index space and $s_{MKN} = \langle\langle 1, M, 1\rangle, \langle 1, K, 1\rangle, \langle 1, N, 1\rangle\rangle$ is such a candidate as both arrays' values can be broadcast into arrays such that the requirements of *apply* are met:

$$\begin{aligned} C_{MKN} &= \mathit{apply}^*(\times, A_{M1N}, B_{1KN}) \\ &= \mathit{apply}(\times, \mathit{reshape}(A_{M1N}, s_{MKN}, F_1), \mathit{reshape}(B_{1KN}, s_{MKN}, F_2)) \end{aligned}$$

where the broadcasting functions F_1, F_2 are determined by the *apply** operator to be:

$$\begin{aligned} F_1(m, k, n) &= \langle m, 1, n\rangle \\ F_2(m, k, n) &= \langle 1, k, n\rangle \end{aligned}$$

The indices for values of these nested reshapings, keeping in mind that A_{M1N} and B_{1kN} are themselves reshapings, are now determined by:

$$\begin{aligned} (T_1 \circ F_1)(m, k, n) = T_1^{-1}(m, 1, n) = \langle m, n\rangle \text{ where } F_1(m, k, n) = \langle m, 1, n\rangle \\ (T_2 \circ F_2)(m, k, n) = T_2^{-1}(1, k, n) = \langle n, k\rangle \text{ where } F_2(m, k, n) = \langle 1, k, n\rangle \end{aligned}$$

Thus if we name these reshapings and consider their values we have:

$$\begin{aligned} A_{MKN} &= \mathit{reshape}(A_{M1N}, s_{MKN}, F_1) \\ A_{MKN}(m, k, n) &= A_{MN}(m, n) \\ B_{MKN} &= \mathit{reshape}(B_{1KN}, s_{MKN}, F_2) \\ B_{MKN}(m, k, n) &= B_{NK}(n, k) \end{aligned}$$

Thus we can consider C_{MKN} to have the values:

$$\begin{aligned} C_{MKN} &= \mathit{apply}(\times, A_{MKN}, B_{MKN}) \\ C_{MKN}(m, k, n) &= A_{MKN}(m, k, n) \times B_{MKN}(m, k, n) \\ C_{MKN}(m, k, n) &= A_{MN}(m, n) \times B_{NK}(n, k) \end{aligned}$$

If we reduce C_{MKN} with additon to get C_{MK}

$$C_{MK} = \mathit{reduce}(+, C_{MKN})$$

then by definition of *reduce* our result C_{MK} will have values:

$$\begin{aligned} C_{MK}(m, k) &= (\mathit{reduce}(+, C_{MKN}))(m, k) \\ &= \mathit{reduce}(+, \{k(x) = C_{MKN}(m, k, x) \mid x \in \{1, \ldots, N\}\}) \\ &= \mathit{reduce}(+, \{k(x) = A_{MN}(m, x) \times B_{KN}(x, k) \mid x \in \{1, \ldots, N\}\}) \end{aligned}$$

Which is clearly the result we were looking for. The example we've just gone through is much more concisely expressed Petalisp's implementation language Common Lisp as presented in [7]:

```
(defun matrix-multiplication (A B)
  (β #'+
  (α #'*
    (-> A (τ (m n) (n m 1)))
    (-> B (τ (n k) (n 1 k))))))
```

however it may be more difficult to follow, in particular the subtleties of how $\alpha(\mathit{apply}^*)$ performs broadcasting to match arrays. There isn't a direct correspondance to the primitives we've presented either; for instance the reshape(−>) here can work out the shape from its parameters and τ is either the affine-linear transformation or a broadcasting.

6.2.3 Abstraction gains

After building an expression nothing is yet evaluated. Were we to use our matrix multiplication expression in a larger expression that only used a single row of the matrix then in the compilation of the larger expression only the single row would be evaluated. In a benchmark [7, p. 16] Petalisp is competitive with a popular numerical and array programming library NumPy; an indication that there may be merit to the direction Petalisp has taken.

The abstraction of application and reduction as primitives lend themselves to the automatic introduction of parallelism. The fuse abstraction is used to abstract arrays that are spread in memory as a singular array. However it can also conceivably abstract over arrays that are spread over a network.

The abstraction of the array type in terms of a function that requires a certain index space also nets an important gain over regular arrays. Where before we would either split our program into array access and functions or waste space and fill an array with a value that could very well be determined from its index , e.g. $f(x) = x$ or some other pattern, we can now simply program against the function yielding the values we would fill in as an array. In combination with reshape we can have a sparse representation of a matrix.

For example consider representing a sparse matrix with zeroes everywhere except for along its diagonal. That could be represented by having a function

$$f(x, y) = \begin{cases} 1 & \text{when } x = y \\ 0 & \text{otherwise} \end{cases}$$

and a shape $\langle\langle 1, 5, 1\rangle, \langle 1, 5, 1\rangle\rangle$ to which we wish to overlay the values of a flat array $g = \{1 \mapsto a, \ldots, 5 \mapsto e\}$ along the diagonal giving us:

$$\begin{bmatrix} a & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & e \end{bmatrix}$$

We then have a few options. One is we can reshape the flat array to an array with the same shape with a the transform $T(x, y) = x$ and use apply with multiplication on it and f to obtain the array. Each access then to a zeroed spot would require a multiplication, which presumably could be optimized away as a compilation step. Other options involve fusing together and reshaping several smaller arrays.

In general the more we can abstract with a small set of primitives the more we get for free as optimization becomes a separate concern.

6.3 Lift

Lift [23] is a compiler for a DSL, the Lift Intermediate Language(IL), aiming at representing array computations with a functional set of primitive operations and a basic array type that closely match easily constructed OpenCL code, in other words primitive operations and types have been carefully chosen for ease of compilation. The purely functional semantics of the DSL and that the means of constructing computations is nested or composed pure functions results in a lot of intermediate results, this however is offset by automatic transformation into "fused" functions that eliminate these intermediaries.

The Lift IL is actually two DSLs one of which is a high-level DSL which compiles to a low-level DSL [23, p. 82] which very closely corresponds to OpenCL primitives. There is quite some difference between the memory model for OpenCL and the typical single linear memory model one might be used to in typical programming paradigms. In particular OpenCL exposes a hierarchy of separate memory areas that must be effectively utilized when using OpenCL to offload highly parallelizable work to a Graphics Processing Unit (GPU) where these separate memory areas actually have direct hardware implementations. The low-level DSL further exposes parameters required to take these memory areas into consideration.

We consider the high-level DSL and refer to it as Lift IL. The Lift IL has been described through a denotational semantics in "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code" [24] for a dependently typed λ-calculus describing an array type and operations on it along with rewrite rules on the operations. The rewrite rules are split into two categories: so-called algorithmic ones and OpenCL specific ones. For the dependently typed λ-calculus and in later publications on Lift operations are described along the lines of equations and transformations are derived or proved correct with equational reasoning.

The result of compiling the low-level Lift IL is OpenCL which in turn can be compiled to run on a GPU. We are mostly interested in what Lift describes as arrays, its expressiveness and its algorithmic rewrite rules so we will take a more informal approach over very rigorous examination of its denotational semantics. We do not present the actual DSL but rather a terse metalanguage close to the one used to describe the high-level Lift IL in most publications.

6.3.1 The primitives of Lift

Here present a semi-formal metalanguage to describe the primitives of Lift. A function is declared $f : (T_1, \ldots, T_k) \to T$ and has the usual set-theoretic interpretation of $f : T_1 \times \ldots \times T_k \to T$ where $T, T_1 \ldots, T_k$ are sets interpreted as types. We allow functions to have explicitly named parameters in their declaration $g : (n_1 : T_1, \ldots, n_k : T_k) \to T \equiv g : (T_1, \ldots, T_k) \to T$ and we use a shorthand for partial application by creating an unordered list of bindings $\alpha = \{n_i = x_i \mid 1 \leq i \leq k\}$ and if $n_p, \ldots, n_{p+|\alpha|}$ are the parameters not bound in α but retaining their order in g then $g\alpha : (n_p : T_p, \ldots, n_{p+|\alpha|} : T_{p+|\alpha|})$ and $g\alpha(n_p, \ldots, n_{p+|\alpha|}) = g(n_1, \ldots, n_k)$.

A type $Int = \{0, 1, 2 \ldots\}$ is assumed to exist, which is a compile-time integer constant for which basic integer arithmethic can be performed at compile-time.

If T or $T_1, T_2 \ldots$ are types then:

- $[T]_n$ is the type of an array of n elements of type T, indexed by values from 0 inclusive to $n - 1$ inclusive of type Int. Values of this type are written $[t_0, \ldots, t_{n-1}]$ where t_i is of type T and i is its index.
- $\langle T_0, T_2, \ldots, T_{n-1} \rangle$ is the type of an n-tuple, indexed by values from 0 inclusive to $n - 1$ inclusive of type Int. Values of this type are written $\langle t_1, \ldots, t_2 \rangle$ where the value t_i has type T_i and i is its index.

If $f : T_2 \to T_3$ and $g : T_1 \to T_2$ are functions then $f \circ g : T_1 \to T_3$ is the function $(f \circ g)(x) = f(g(x))$. Sometimes it is more natural to reverse the order of composition which we allow with $g; f = f \circ g$ s.t. $(g; f)(x) = f(g(x))$ and we allow a reversed application s.t. $x; g; f = (g; f)(x)$. In long chains we will number compositions as follows $x; g;_1 g;_2 ;_3 g$. With this vocabulary we can present a selection of operations. It is assumed that the user can provide pure functions and flat arrays (i.e. $[T]_n$ where T cannot be an array) to be used with these operators:

- Map a function over an array

$$\begin{aligned}&\mathit{map} : (f : T \to U, a : [T]_n) \to [U]_n \\ &\mathit{map}(f, [t_0, \ldots, t_{n-1}]) = [f(t_0), \ldots, f(t_{n-1})]\end{aligned}$$

- Reduce an array with a binary function

$$\begin{aligned}&\mathit{reduce} : (z : U, f : (u : U, t : T) \to U, a : [T]_n) \to [U]_1 \\ &\mathit{reduce}(z, f, [t_0, \ldots, t_{n-1}]) = [z; f\{t = t_0\};_1 \cdots ;_{n-1} f\{t = t_{n-1}\}]\end{aligned}$$

- Iterate a function g on an array m times

$$\begin{aligned}&\mathit{iterate} : (m : \mathit{Int}, g : [T]_n \to [T]_n, a : [T]_n,) \to [T]_n \\ &\mathit{iterate}(m, g, a) = a; \mathit{map}\{f = g\};_1 \cdots ;_m \mathit{map}\{f = g\}\end{aligned}$$

- Zip two arrays together as an array of tuples

$$\begin{aligned}&\mathit{zip} : (l : [T]_n, r : [U]_n) \to [\langle T, U\rangle]_n \\ &\mathit{zip}([t_0, \ldots, t_{n-1}], [u_0, \ldots, u_{n-1}]) = [\langle t_0, u_0\rangle, \ldots, \langle t_{n-1}, u_{n-1}\rangle]\end{aligned}$$

- Join an array of arrays together

$$\begin{aligned}&\mathit{join} : ([[T]_n]_m,) \to [T]_{n\times m} \\ &\mathit{join}([[t_0, \ldots, t_{0+n-1}], \ldots, [t_{n(m-1)}, \ldots, t_{n(m-1)+n-1}]] = \\ &\qquad [t_0, \ldots, t_{0+n-1}, \ldots, t_{n(m-1)}, \ldots, t_{n(m-1)+n-1}]\end{aligned}$$

- Split an array into an arrays of arrays, m must divide n i.e. $n = mk$ where k is a positive integer. Here i ranges over the new indexing i.e. $i \in \{0, \ldots, (n/m) - 1\}$

$$\mathit{split} : (m : \mathit{Int}, a : [T]_n) \to [[T]_m]_{n/m}$$

$$\begin{aligned}\mathit{split} : (m : \mathit{Int}, [t_0, \ldots, t_{n-1}]) = [\,&[t_0, \ldots, t_{0+m-1}], \ldots \\ &, [t_{mi+0}, \ldots, t_{mi+m-1}], \ldots \\ &, [t_{m((n/m)-1)+0}, \ldots, t_{m(n/m)+m-1}]\,]\end{aligned}$$

Note that we can regard *split* as consuming a function $a : \{0, \dots, n-1\} \to T$ and producing a function $g : \{0, \dots, n/m\} \to \{[a(mi+0), \dots, a(mi+m-1)] \mid 0 \leq i \leq n/m - 1\}$

- Accessor functions for arrays and tuples

$$\begin{aligned} at &: (i : Int, [T]_n) \to T \\ at(i, [t_0 &\dots, t_{n-1}]) = t_i \\ get &: (i : Int, \langle T_0, \dots, T_{n-1} \rangle) \to T_i \\ get &: (i, \langle t_0, \dots, t_{n-1} \rangle) = t_i \end{aligned}$$

We can easily discern a relationship between *at* and *split*

$$\begin{aligned} at(x, at(y, split(s, a))) &= at(x, [at(a, sy + 0), \dots, at(sy + s - 1)]) \\ &= at(a, sy + x) \end{aligned}$$

Alternatively we may write this identity in compositional style:

$$a; split\{m = s\}; at\{i = y\}; at\{i = x\} = at(a, sy + x)$$

Which without applying arguments becomes:

$$split\{m = s\}; at\{i = y\}; at\{i = x\} = at\{i = sy + x)\}$$

To construct multidimensional arrays(in the array of arrays fashion) one must start from flat arrays which are then “reinterpreted” through split. Notice that split corresponds to adding a dimension whereas join corresponds to removing a dimension. For example creating a 3 by 3 matrix in this manner corresponds to laying out 9 elements in an array and splitting it by 3 and conversely flattening it again corresponds to joining.

The only representation available for arrays in OpenCL, the eventual compilation target, is the flat array of floating point or integer types. Thus the array of array construct is solely a compile-time construct. As such it is of particular interest how arrays are compiled so we must delve into the intermediary compile-time representation of arrays.

Certain primitive operations can be regarded as only changing how values are accessed. For the ones presented these are zip, split and join. In Lift representing this change is referred to as creating a view(of an array) and is detailed in [23, p. 79] where such primitives applied to an array do not result in any copies of array values but instead a structure representing the change in access to arrays and a reference to the original array. This view is an intermediary compile-time structure that is eliminated after facilitating access to values. For several nestings of view altering primitives a chain of nodes are created that store the change in type and indexing. Constructing an index for an array that is the result of such a chain of primitives then is the process of composing indexing functions generated for these in reverse order of construction. For instance consider an array being created by the chain of primitives:

$$zip(l, r); split\{m = k_1\}; split\{m = k_2\}$$

where the arrays zipped together are of size n. A chain of nodes is created where the first node generated by *zip* references the two which we assume are flat arrays l and k, the first *split* then references the *zip* and the second *split* references the first *split*. Upon any value being read from the resulting array this construction may be seen as *zip* wrapping two arrays and providing an indexing function $f : Int \to [\langle L, R\rangle]_n$ where $[L]_n$ and $[R]_n$ are the types of l, r respectively. The first *split* provides a new indexing function $g : Int \to [[\langle L, R\rangle]_{k_1}]_{n/k_1}$. The second *split* provides yet another indexing function $h : Int \to [[[\langle L, R\rangle]_{k_1}]_{k_2}]_{n/k_1/k_2}]$. Any indexing then corresponds to expanding these by providing zero(for the array itself) up to three array indices and a tuple index(for a single value). For example we can evaluate indexing for a full indexing by using the nested *at* identity twice:

$$\begin{aligned}
&zip(l, r); split\{m = k_1\}; (split\{m = k_2\}; at\{i = i_1\}; at\{i = i_2\});\\
&at\{i = i_3\}; get\{i = i_4\}\\
=&zip(l, r); (split\{m = k_1\}; at\{i = k_2 i_1 + i_2\}; at\{i = i_3\}); get\{i = i_4\}\\
=&zip(l, r); at\{i = k_1(k_2 i_1 + i_2) + i_3\}; get\{i = i_4\}\\
=&get(i_4, at(k_1(k_2 i_1 + i_2) + i_3, zip(l, r)))
\end{aligned}$$

We omit covering further primitives and consider how Lift describes the various interactions between primitive operations that are admissable to rewrite rules.

6.3.2 Rewriting in Lift

Given that a certain combination of primitives expresses an algorithm one wishes to implement Lift then analyses the expression and attempt to perform substitution into equivalent expressions, according to certain identities encoded in Lift. An example of two such for Lift [24, p. 5] are:

- Decomposing iterate:

$$iterate(x + y, g, a) = iterate(x, g, iterate(y, g, a))$$

A justification is given by equational reasoning and some assumptions making it informal:

$$\begin{aligned}
\mathit{iterate}(x+y,g,a) &= a; \mathit{map}\{f=g\};_1 \cdots ;_{x+y} \mathit{map}\{f=g\} \\
&= a; \mathit{map}\{f=g\};_1 \cdots ;_x ; \mathit{map}\{f=g\} \\
&\quad ;_{x+1} \cdots ;_{x+y} \mathit{map}\{f=g\} \\
&= a; \mathit{iterate}\{m=x, g=g\}; \mathit{iterate}\{m=y, g=g\} \\
&= \mathit{iterate}(x+y,g,a)
\end{aligned}$$

- Composing maps:

$$\mathit{map}(f, \mathit{map}(g, [t_0, \dots, t_{n-1}])) = \mathit{map}(f \circ g, [t_0, \dots, t_{n-1}])$$

A justification follows from function composition:

$$\begin{aligned}
\mathit{map}(f, \mathit{map}(g, [t_0, \dots, t_{n-1}])) &= \mathit{map}(f, [g(t_0), \dots, g(t_{n-1})]) \\
&= [f(g(t_0)), \dots, f(g(t_{n-1}))] \\
&= \mathit{map}(f \circ g, [t_0, \dots, t_{n-1}])
\end{aligned}$$

These are fairly self-evident identities and are encoded in Lift as rewrite rules. The interface for writing more rules is through extending a certain aspect of the implementation of Lift. As an aside, something that might be somewhat alarming in languages that can rely on rewrite rules is perhaps the reliance on, as we have succumbed to here as well albeit for presentation only, is the informal hand-waving about identites that while good for pedagogical purposes so as to not drown in minutiae does not instill trust in things actually being identities. For a proper industry standard tool a formal specification and mechanized verification should be applied.

Instead of going further into language details, we turn to what rewriting strategy is employed in Lift. In this discussion we consider some expression e built from the operations we have listed some of[3] If some rewrite applies $e \to e'$ then e' is a new expression where one substitiution has been applied. Lift considers any rule that can apply to e as generating a search space and

[3] Note that there is a subtstantial amount more primitive operations and identities, especially as Lift's high-level language is translated to the low-level language where parameters pertaining to OpenCL are exposed.

performs several random walks in the search space:

$$\begin{aligned}
&\text{Walk 1: } e \to e_1 \to e_2\\
&\text{Walk 2: } e \to e_3 \to e_4\\
&\text{Walk 3: } e \to e_6 \to e_7\\
&\text{Walk 4: } e \to e_8 \to e_9
\end{aligned}$$

Automatic search space pruning is performed by emitting code for the expression resulting for each walk and measuring its performance in runs, this is referred to as a Monte-Carlo descent[24, p. 8]. The best performing walk's path is fixed and search continues from it until no rules apply. To optimize this process Lift has a macro facility to compose several steps should one notice that a certain pattern of steps is always taken and better expressed as one monolithic step.

6.3.3 Lift choices

The primitives of Lift have been picked with the intent that there exist a high performing implementation of the primitives or they easily decompose to one[6, p. 3] on the targeted hardware. This may suggest why the array construct of Lift is kept relatively simple and the focus on chaining operations.

6.4 The need for array abstractions

In this chapter we have reviewed different approaches to an array vocabulary where the common theme is abstraction to achieve adaptable array code. The adaptation varies; runtime JIT compilation, spread memory hierarchies or substitutable array types by relying on an interface. Specifying identities and enabling transformations based on them is central to compiling these abstractions. A move towards a logical view of the array affords us a partitioning of implementations into an algorithm level and a implementation concern level. We can then potentially avoid rewriting algorithms in detail and instead focus on how to adapt algorithms based on implementation concerns such as distributing computation over multiple CPU or GPU cores or a cluster of machines and varying memory models.

When one depends on a concrete array type, as opposed to an array interface, a usage or data pattern could emerge that would greatly benefit from a change in the underlying implementation. For arrays, should a sparse array be desirable for a certain problem(a data pattern), a hard dependency on primitive arrays(e.g. C arrays) would prevent this optimization even though the algorithm using the array would only require an interface that both types would satisfy.

A more far-reaching concern is that a hard dependency on a certain array type locks the algorithm described to the computational paradigm and memory model exposed by the array type. What we mean by this is that if the method provided to perform computations over arrays is setting and getting values at indices in some sequence in a flat memory model, and there is no real need to express an algorithm in such terms, then that would exclude offloading the same algorithm or execution in, for example, a parallel execution paradigm with a spread hierarchy of memory. In this case we must revert to reimplementing the algorithm in its entirety to adapt it to changing situations.

CHAPTER 7

A Mathemathics of Arrays applied

Here we present a contribution to the “6th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming” in which we apply MOA through modeling key aspects of it in Magnolia and applying it to rewrite an array problem within the vocabulary of MOA from which we will attempt to generate efficient array code. It shall serve as a starting point to answer the question: Can we find a set of rules to, given a MOA description of an array computation, generate efficient code from our Magnolia modeling of MOA expressions.

There were several authors of this paper. The author of this thesis participated in checking the correctness of the MOA derivations and contributed the Magnolia translations of said MOA derivations. Note that the provided paper here does not correspond to the one in the provided DOI, but one where a sign error has been corrected.

There is a follow-up paper in appendix F in which new developments are included.

Finite Difference Methods Fengshui: Alignment through a Mathematics of Arrays

Benjamin Chetioui
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
benjamin.chetioui@uib.no

Lenore Mullin
College of Engineering and Applied Sciences
University at Albany, SUNY
Albany, NY, USA
lmullin@albany.edu

Ole Abusdal
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
ole.abusdal@student.uib.no

Magne Haveraaen
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
https://www.ii.uib.no/~magne/

Jaakko Järvi
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
jaakko.jarvi@uib.no

Sandra Macià
Barcelona Supercomputing Center
(BSC - CNS)
Barcelona, Spain
sandra.macia@bsc.es

Abstract

Numerous scientific-computational domains make use of array data. The core computing of the numerical methods and the algorithms involved is related to multi-dimensional array manipulation. Memory layout and the access patterns of that data are crucial to the optimal performance of the array-based computations. As we move towards exascale computing, writing portable code for efficient data parallel computations is increasingly requiring an abstract productive working environment. To that end, we present the design of a framework for optimizing scientific array-based computations, building a case study for a Partial Differential Equations solver. By embedding the Mathematics of Arrays formalism in the Magnolia programming language, we assemble a software stack capable of abstracting the continuous high-level application layer from the discrete formulation of the collective array-based numerical methods and algorithms and the final detailed low-level code. The case study lays the groundwork for achieving optimized memory layout and efficient computations while preserving a stable abstraction layer independent of underlying algorithms and changes in the architecture.

ARRAY '19, June 22, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6717-2/19/06…$15.00
https://doi.org/10.1145/3315454.3329954

CCS Concepts • **Software and its engineering → Software design engineering**;

Keywords Mathematics of Arrays, Finite Difference Methods, Partial Differential Equations, Psi calculus, Magnolia

ACM Reference Format:
Benjamin Chetioui, Lenore Mullin, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Sandra Macià. 2019. Finite Difference Methods Fengshui: Alignment through a Mathematics of Arrays. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3315454.3329954

1 Introduction

Given an address space, the data layout and the pattern of accessing that data are fundamental for the efficient exploitation of the underlying computer architecture. The access pattern is determined by a numerical algorithm, which may have been tuned to produce a particular pattern. The data layout may have to be adjusted explicitly to a given pattern and the computer hardware architecture. At the same time, high-performance environments are evolving rapidly and are subject to many changes. Moreover, numerical methods and algorithms are traditionally embedded in the application, forcing rewrites at every change. Thus the efficiency and portability of applications are becoming problematic. Under this scenario, software or hardware modifications usually lead to a tedious work of rewriting and tuning throughout which one must ensure correctness and efficiency. To face this scenario, the scientific community suggests a separation of concerns through high-level abstraction layers.

Burrows et al. identified a Multiarray API for Finite Difference Method (FDM) solvers [8]. We investigate the fragment of the Mathematics of Arrays (MoA) formalism [22, 23] that corresponds to this API. MoA gives us the ψ-calculus for optimizing such solvers. We present a full system approach from high level coordinate-free Partial Differential Equations (PDEs) to preparing for the layout of data and code optimization, using the MoA as an intermediate layer and the Magnolia programming language [5] to explore the specifications. In this framework, a clean and natural separation occurs between application code, the optimization algorithm and the underlying hardware architecture, while providing verifiable components. We fully work out a specific test case that demonstrates an automatable way to optimize the data layout and access patterns for a given architecture in the case of FDM solvers for PDE systems. We then proceed to show that our chosen fragment of the rewriting system defined by the ψ-calculus makes up a canonical rewriting subsystem, i.e. one that is both strongly normalizing and confluent.

In the proposed system, algorithms are written against a stable abstraction layer, independent of the underlying numerical methods and changes in the architecture. Tuning for performance is still necessary for the efficient exploitation of different computer architectures, but it takes place below this abstraction layer without disturbing the high-level implementation of the algorithms.

This paper is structured as follows. Section 2 presents the related work, and a concise literature review of the state of the art. Section 3 introduces the general software stack composition and design used for our purposes. Section 4 details the optimizations and transformation rules. The PDE solver test case showcasing the framework is presented in Section 5. Finally, conclusions are given in Section 6.

2 Related work

Whole-array operations were introduced by Ken Iverson [18] in the APL programming language, an implementation of his notation to model an idealized programming language with a universal algebra. Ten years later, shapes were introduced to index these operations by Abrams [1]. Attempts to compile and verify APL proved unsuccessful due to numerous anomalies in the algebra [34]. Specifically, $\iota\sigma$ was equivalent to $\iota\langle\ \sigma\ \rangle$, where σ is a scalar and $\langle\ \sigma\ \rangle$ is a one element vector. Moreover, there was no indexing function nor the ability to obtain all indices from an array's shape. This caused Perlis to conclude the idealized algebra should be a Functional Array Calculator based on the λ-calculus [34]. Even with this, no calculus of indexing was formulated until the introduction of MoA [22]. MoA can serve as a foundation for array/tensor operations and their optimization.

Numerous languages emerged with monolithic or whole-array operations. Some were interpreted (e.g. Matlab and Python), some were compiled (e.g. Fortran90 and TACO [19]) and some were Object Oriented with preprocessing capabilities (e.g. C++ with expression templates [9, 32]). Current tensor (array) frameworks in contemporary languages, such as Tensorflow [33] and Tensor Toolbox [4] provide powerful environments to model tensor computations. None of these frameworks are based on the ψ-calculus.

Existing compilers have various optimizations that can be formulated in the ψ-calculus, e.g. loop fusion (equivalent to distributing indexing of scalar operations in MoA) and loop unrolling (equivalent to collapsing indexing based on the properties of ψ and the ψ-correspondence Theorem (PCT) in MoA [23]). Many of the languages mentioned above implement concepts somewhat corresponding to MoA's concept of shape and its indexing mechanism. It is, however, the properties of the ψ-calculus and its ability to obtain a Denotational Normal Form (DNF) for any computation that make it particularly well-suited for optimization.

Hagedorn et al. [13] pursued the goal of optimizing stencil computations using rewriting rules in LIFT.

3 Background, design and technologies

We present the design of our library-based approach structured by layers. Figure 1 illustrates this abstract generic environment. At the domain abstraction layer,

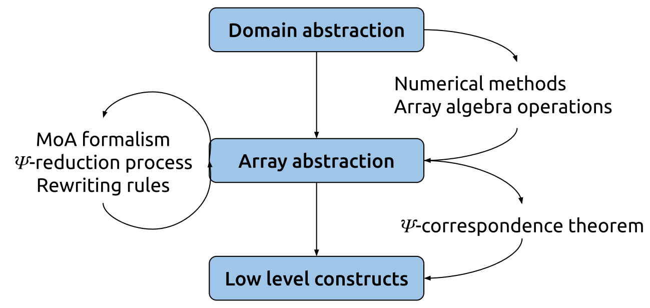


Figure 1. Layer abstraction design; generic environment approach.

code is written in the integrated specification and programming language Magnolia, a language designed to support a high level of abstraction, ease of reasoning, and robustness. At the intermediate level, the MoA formalism describes multi-dimensional arrays. Finally, through the ψ-correspondence theorem, the array abstraction layer is mapped to the final low-level code.

3.1 Magnolia

Magnolia is a programming language geared towards the exploration of algebraic specifications. It is being designed at the Bergen Language Design Laboratory [5]; it is a work in progress and is used to teach the Program Specification class at the University of Bergen, Norway. Magnolia's strength relies in its straightforward way of working with abstract constructs.

Magnolia relies primarily on the *concept* module, which is a list of type and function declarations (commonly called a *signature*) constrained by *axioms*. In Magnolia, an *axiom* defines properties that are assumed to hold; it however differs from the usual axioms in mathematics in that an axiom in Magnolia may define derived properties. Functions and axioms may be given a *guard*, which defines a precondition. The *satisfaction* module serves to augment our knowledge with properties that can be deduced from the premises, typically formatted to indicate that a *concept* models another one.

Magnolia is unusual as a programming language in that it does not have any built-in type or operation, requiring that everything be defined explicitly. Magnolia is transpiled to other languages, and thus, the actual types the programmer intends to use when running their program must be defined in the target language.

3.2 Mathematics of Arrays

MoA [22, 23] is an algebra for representing and describing operations on arrays. The main feature of the MoA formalism is the distinction between the *DNF*, which describes an array by its shape together with a function that defines the value at every index, and the *Operational Normal Form* (ONF), which describes it on the level of memory layout. The MoA's ψ-calculus [23] provides a formalism for index manipulation within an array, as well as techniques to reduce expressions of array operations to the DNF and then transform them to ONF.

The ψ-calculus is based on a generalized array indexing function, ψ, which selects a partition of an array by a multidimensional index. Because all the array operations in the MoA algebra are defined using shapes, represented as a list of sizes, and ψ, the reduction semantics of ψ-calculus allow us to reduce complex array computations to basic indexing/selection operations, which reduces the need for any intermediate values.

By the ψ-correspondence theorem [23], we are able to transform an expression in DNF to its equivalent ONF, which describes the result in terms of loops and controls, *starts*, *strides* and *lengths* dependent on the chosen linear arrangement of the items, e.g. based on hardware requirements.

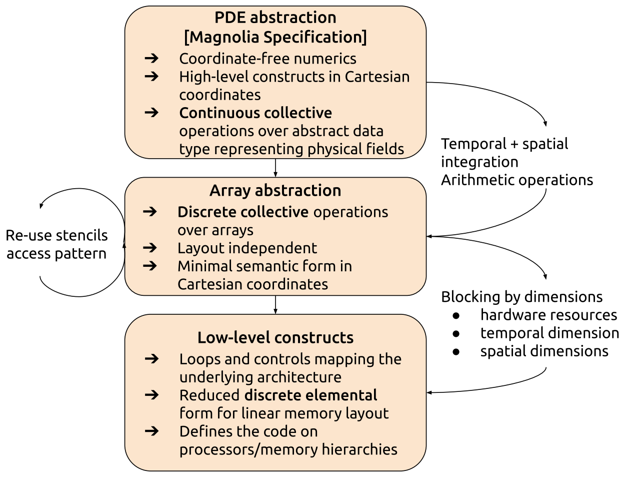

PDE abstraction
[Magnolia Specification]
➔ Coordinate-free numerics
➔ High-level constructs in Cartesian coordinates
➔ Continuous collective operations over abstract data type representing physical fields
Temporal + spatial integration
Arithmetic operations
Array abstraction
➔ Discrete collective operations over arrays
➔ Layout independent
➔ Minimal semantic form in Cartesian coordinates
Re-use stencils access pattern
Blocking by dimensions
● hardware resources
● temporal dimension
● spatial dimensions
Low-level constructs
➔ Loops and controls mapping the underlying architecture
➔ Reduced discrete elemental form for linear memory layout
➔ Defines the code on processors/memory hierarchies

Figure 2. Layer abstraction design; detailed environment designed for a PDE solver.

3.2.1 Motivation behind DNF and ONF

The goal behind the DNF and the ONF is to create an idealized foundation to define most — if not all — domains that use tensors (arrays). Using MoA, all of the transformations to the DNF can be derived from the definition of the ψ function and shapes.

This view has a long history [1] and, when augmented by the λ-calculus [6], provides an idealized semantic core for all arrays [26, 27]. Array computations are very prevalent. A recent Dagstuhl workshop [2, 3] reported the pervasiveness of tensors in the Internet of things, Machine Learning, and Artificial Intelligence (e.g. Kronecker [24]) and Matrix Products [11]. Moreover, they dominate science [12, 21] in general, especially signal processing [25, 28, 30, 31] and communications [29].

3.3 PDE solver framework

Figure 2 illustrates the design structured by layers for the PDE solver framework we describe. The first abstraction layer defines the problem through the domain's concepts. At this level, PDEs are expressed using collective and continuous operations to relate the physical fields involved. Through the functions encapsulating the numerical methods, the high-level continuous abstraction is mapped to a discrete array-based layer. A Magnolia specification of the array algebra defined by the MoA formalism and the ψ-calculus has been developed at this level. This algebra for arithmetic operations and permutations over multi-dimensional arrays defines the problem through collective array operations in a layout independent manner. At this point, array manipulation functions and operations may be defined in the MoA formalism and reduced according to the ψ-reduction process. This process simplifies an expression through transformational and compositional reduction properties: the rewriting rules. From the user's array-abstracted expression we obtain an equivalent optimal and minimal semantic form. Finally, the indexing algebra of the ψ-calculus relates the monolithic operations to elemental operations, defining the code on processors and memory hierarchies through loops and controls. The ψ-correspondence theorem is the theorem defining the mapping from the high-level abstracted array expressions to the operational expressions, i.e. from a form involving Cartesian coordinates into one involving linear arranged memory accesses.

4 MoA transformation rules

4.1 ψ-calculus and reduction to DNF

Multiarrays, or multidimensional arrays, have a shape given by a list of sizes $\langle s_0 \ \ldots \ s_{n-1}\rangle$. For example, a 6 by 8 matrix A has the shape $\langle 6 \ 8\rangle$. The index for a multiarray is given by a multi-index $\langle i_0 \ \ldots \ i_{n-1}\rangle$. For position j of the multi-index, the index i_j is in the range $0 \leq i_j < s_j$. This sets the vocabulary for talking about multiarrays. In the following Magnolia code and in the rest of the paper, we will assume that the following types are declared:

- **type** `MA`, for Multiarrays;
- **type** `MS`, for Multishapes;
- **type** `MI`, for Multi-indexes;
- **type** `Int`, for Integers.

All these types will have (mapped) arithmetic operators. Important functions on a multiarray are:

- the shape function ρ, which returns the shape of a multiarray, e.g. $\rho A = \langle 6 \ 8\rangle$;
- the ψ function, which takes a submulti-index and returns a submultiarray, e.g. $\langle\rangle \psi A = A$ and $\rho(\langle 3\rangle \psi A) = \langle 8\rangle$ is the subarray at position 3;
- the rotate function θ, which rotates the multiarray: $p \ \theta_x \ A$ denotes the rotation of A by offset p along axis x (rotate does not change the shape: $\rho(p \ \theta_x \ A) = \rho A$).

With respect to ψ, rotate works as:

$$\langle \ i_0 \ldots i_x \ \rangle \ \psi \ (p \ \theta_0 \ A) \quad = \quad \langle \ (i_0 + p) \mod s_0 \ \ldots \ i_x \ \rangle \ \psi \ A$$

The rotate operation can be used to calculate, for each element, the sum of the elements in the adjacent columns, $(1 \ \theta_0 \ A) + ((-1) \ \theta_0 \ A)$, which is a multiarray with the same shape as A. Applying ψ to the expression gives the following reduction:

$$\langle i_0\rangle \ \psi \ ((1 \ \theta_0 \ A) + \ ((-1) \ \theta_0 \ A)) = \langle (i_0 + 1) \mod s_0\rangle \ \psi \ A + \langle (i_0 - 1) \mod s_0\rangle \ \psi \ A$$

These above MOA functions can be declared in Magnolia, with axioms stating their properties.

```
/** Extract the shape of an array. */
function rho(a:MA) : MS;
/** Extract subarray of an array. */
function psi(a:MA, mi:MI) : MA;
/** Rotate distance p along axis. */
function rotate(a:MA, axis:Int, p:Int) : MA ;
axiom rotateShape(a:MA, ax:Int, p:Int) {
  var ra = rotate(a,ax,p);
  assert rho(ra) == rho(a);
}
axiom rotatePsi(a:MA, ax:Int, p:Int, mi:MI) {
  var ra = rotate(a,ax,p);
  var ij = pmod(get(mi,ax)+p,get(rho(a),ax));
  var mj = change(mi,ax,ij);
  assert psi(ra,mi) == psi(a,mj);
}
axiom plusPsi(a:MA, b:MA, mi:MI)
  guard rho(a) == rho(b) {
  assert rho(a+b) == rho(a);
  assert psi(a+b,mi) == psi(a,mi) + psi(b,mi);
}
```

Note how we are using ρ and ψ to define operations on multiarrays. The ρ operator keeps track of the resulting shape. The ψ operator takes a partial multi-index and explains the effect of the operation on the subarrays. In this way the ψ operator moves inward in the expression, pushing the computation outwards towards subarrays and eventually to the element level. The concatenation property for ψ-indexing is important for this,

$$\langle \ j \ \rangle \ \psi \ (\langle \ i \ \rangle \ \psi \ A) \equiv \langle \ i \ j \ \rangle \ \psi \ A.$$

```
axiom psiConcatenation(ma:MA, q:MI, r:MI) {
    var psiComp = psi( psi( ma,q ), r );
    var psiCat = psi( ma, cat( q,r ) );
    assert psiComp == psiCat;
  }
```

The rules above, for rotation and arithmetic, show how ψ moves inwards towards the multiarray variables. When this process stops, we have reached the DNF. All other multiarray functions have then been removed and replaced by their ψ definitions. What is left to figure out and what we will tentatively in this paper is how to build the DNF.

Burrows et al [8] made the case that the operations defined above augmented with mapped arithmetic constitute a sufficient basis to work with any FDM solver of PDE systems. It does not matter what language the original expression comes from (Python, Matlab, Fortran, C, etc). With the syntax removed and the tokens expressed as an AST, the DNF denotes the reduced semantic tree and could be returned to the syntax of the originating language, with interpretation or compilation proceeding as usual.

4.2 Transformation rules

The MoA defines many rewriting rules in order to reduce an expression to its DNF. Working with those, we got the insight that the goal of the reduction is to move the call to ψ inwards to apply it as early as possible in order to save computations, and that there are enough rules to allow us to move ψ across any type of operation (Multiarray on Multiarray, scalar on Multiarray).

For the sake of this particular example, we limited ourselves to a subset of the transformation rules in the MoA. We show that this constitutes a rewriting system that is canonical.

Let us first introduce the rules we are using. In the rules, the metavariables $index_i$, u_i and sc_i respectively denote multi-indexes, multiarrays and scalars. The metavariable op is used for mappable binary operations such as $\times$, $+$ and $-$, that take either a scalar and a multiarray or two multiarrays as parameters and return a multiarray.

$$\frac{index\ \psi\ (u_i\ op\ u_j)}{(index\ \psi\ u_i)\ op\ (index\ \psi\ u_j)}\ \text{R1}$$

$$\frac{index\ \psi\ (sc\ op\ u)}{sc\ op\ (index\ \psi\ u)}\ \text{R2}$$

$$\frac{k \geq i \implies \langle\ sc_0\ \dots\ sc_i\ \dots\ sc_k\ \rangle\ \psi\ (sc\ \theta_i\ u)}{\langle\ sc_0\ \dots\ ((sc_i + sc)\ mod\ (\rho\ u)[i])\ \dots\ sc_k\ \rangle\ \psi\ u}\ \text{R3}$$

Proving that a rewriting system is canonical requires proving two properties [20]:

1. the rewriting system must be confluent;
2. the rewriting system must be strongly normalizing (reducible in a finite number of steps).

For a rewriting system, being confluent is the same as having the Church-Rosser property [20], i.e. in the case when reduction rules overlap so that a term can be rewritten in more than one way, the result of applying any of the overlapping rules can be further reduced to the same result. If a term can be derived into two different terms, the pair of the two derived terms is called a critical pair. Proving that a rewriting system is confluent is equivalent to proving that every critical pair of the system yields the same result for both of its terms.

Our rules above of the rewriting system can not generate any critical pair; the system is thus trivially confluent.

Now, we must prove that the rewriting system is strongly normalizing: the system must yield an irreducible expression in a finite number of steps for any expression. To that end, we assign a weight $w \in \mathbb{N}$ to the expression such that w represents the "weight" of the expression tree. We define the weight of the tree as the sum of the weight of each $(index\ \psi)$ node. The weight of each one of these nodes is equal to 3^h, where h is the height of the node.

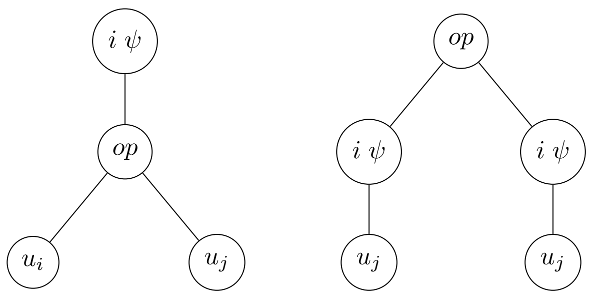


Figure 3. Rule 1 and its application.

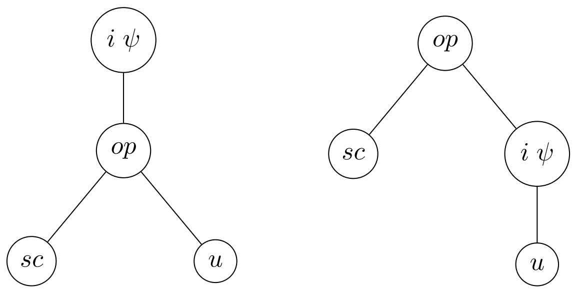


Figure 4. Rule 2 and its application.

Since $\mathbb{N}$ is bounded below by 0, we simply need to prove that the application of each rule results in w strictly decreasing to prove that our rewriting system is strongly normalizing.

For each one of our three rules, we draw a pair of trees representing the corresponding starting expression on the left and the resulting expression from applying the rule on the right. Then, we verify that w strictly decreases from the tree on the left to the tree on the right. We call w_l the weight of the left tree and w_r the weight of the right tree. Figures 3, 4 and 5 illustrate these trees.

In the three figures, we assume that the tree rooted in the $i\psi$ node has height h'. Since the $i\psi$ node has a parameter, it is never a leaf and we have $h' > 0$.

In Figure 3, the starting expression has the weight $w_l = 3^{h'}$. The resulting expression from applying $R1$, however, has the weight $w_r = 2 \times 3^{h'-1} = \frac{2}{3}w_l$, which is less than w_l. In Figure 4, the starting expression has the weight $w_l = 3^{h'}$. The resulting expression from applying $R2$, however, has the weight $w_r = 3^{h'-1} = \frac{1}{3}w_l$, which is less than w_l. In Figure 5, the starting expression has the weight $w_l = 3^{h'}$. The resulting expression from applying $R3$, however, has the weight $w_r = 3^{h'-1} = \frac{1}{3}w_l$, which is less than w_l.

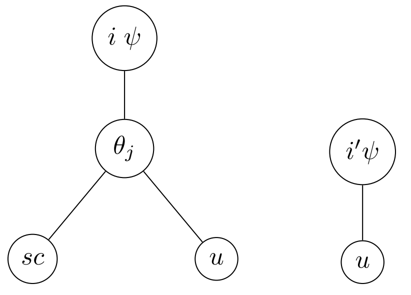


Figure 5. Rule 3 and its application.

Since w strictly decreases with every rewrite, the system is strongly normalizing. Since it is also confluent, it is canonical.

4.3 Adapting to hardware architecture using ONF

Once we have reduced an expression to its DNF, if we know about the layout of the data it uses, we can build its ONF. Assuming a row major layout, let us turn $\langle\ i\ \rangle\ \psi\ ((1\ \theta_0\ A) + ((-1)\ \theta_0\ A))$ into its ONF.
To proceed further, we need to define three functions: γ, ι and rav.

- rav is short for Ravel, which denotes the *flattening* operation, both in APL and in MoA. It takes a multiarray and reshapes it into a vector. We therefore use rav to deal with the representation of the array in the memory of the computer.
- γ takes an index and a shape and returns the corresponding index in the flattened representation of the array[1]. γ is not computable unless a specific memory layout is assumed, which is why this decision has to be taken before building the ONF. One can note that rav and γ are tightly connected in defining flattened array accesses as γ encodes the layout while rav is defined in terms of γ. For FDM, it is important therefore to figure out the right memory layout such that rotations are completed in an efficient fashion.
- ι is a unary function, which takes a natural number n as its parameter and returns a 1-D array containing the range of natural numbers from 0 to n excluded. It is used to build *strides* of indexes needed by the ONF.

With these operations defined, we can proceed. We first apply the ψ-correspondence theorem followed by applying γ.

$$\begin{aligned}
&\forall i\ \ s.t.\ \ 0 \le i < 6\\
&\quad \langle\ i\ \rangle\ \psi\ ((1\ \theta_0\ A) + ((-1)\ \theta_0\ A))\\
&\qquad \equiv (rav\ A)[\gamma(\langle\ (i+1)\ mod\ 6\ \rangle\ ;\ \langle\ 6\ \rangle) \times 8 + \iota 8]\ +\\
&\qquad\quad (rav\ A)[\gamma(\langle\ (i-1)\ mod\ 6\ \rangle\ ;\ \langle\ 6\ \rangle) \times 8 + \iota 8]\\
&\qquad \equiv (rav\ A)[((i+1)\ mod\ 6) \times 8 + \iota 8]\ +\\
&\qquad\quad (rav\ A)[((i-1)\ mod\ 6) \times 8 + \iota 8]
\end{aligned}$$

[1]Here, only γ on rows is considered, but other γ functions exist

Secondly, we apply rav and turn ι into a loop to reach the following generic program:

$$\begin{aligned}
&\forall j\ \ s.t.\ \ 0 \le j < 8\\
&\quad A[((i+1)\ mod\ 6) \times 8 + j]\ +\\
&\quad A[((i-1)\ mod\ 6) \times 8 + j]
\end{aligned}$$

The ONF is concerned with performance, and is where cost analysis and *dimension lifting* begins.

Regarding pure cost analysis, at this point, it is still possible to optimize this program: unfolding the loops gives us the insight that the modulo operation is only ever useful on the 0^{th} and 5^{th} row. Thus, by splitting the cases into those that require the modulo operation to be run and those that do not, we may achieve better performance.

Now imagine breaking the problem over 2 processors. Conceptually, the dimension is lifted. It is important to note that the lifting may happen on any axis, especially in the current case where we are dealing with rotations on a given axis. If we happen to apply dimension lifting on the axis on which we are rotating, we may not be able to split the memory perfectly between the different computing sites. This could require inter-process communication, or duplication of memory.

In this case, since we are rotating on the 0^{th} axis, we pick axis 1 as the candidate to be lifted. The loop on j is then split into 2 loops because we now view the 2-D resultant array as a 3-D array A' with shape $\langle\ 6\ 2\ 8/2\ \rangle\ =\ \langle\ 6\ 2\ 4\ \rangle$ in which axis 1 corresponds to the number of processors. Therefore, we get:

$$\begin{aligned}
&\forall i, j\ \ s.t.\ 0 \le i < 6,\ \ 0 \le j < 2\\
&\quad \langle\ i\ j\ \rangle\ \psi\ ((1\ \theta_0\ A') + ((-1)\ \theta_0\ A'))\\
&\qquad \equiv (rav\ A')[\gamma(\langle\ ((i+1)\ mod\ 6)\ j\ \rangle\ ;\ \langle\ 6\ 2\ \rangle) \times 4 + \iota 4]\ +\\
&\qquad\quad (rav\ A')[\gamma(\langle\ ((i-1)\ mod\ 6)\ j\ \rangle\ ;\ \langle\ 6\ 2\ \rangle) \times 4 + \iota 4]\\
&\qquad \equiv (rav\ A')[(((i+1)\ mod\ 6) \times 2 + j) \times 4 + \iota 4]\ +\\
&\qquad\quad (rav\ A')[(((i-1)\ mod\ 6) \times 2 + j) \times 4 + \iota 4]
\end{aligned}$$

This reduces to the following generic program:

$$\begin{aligned}
&\forall k\ \ s.t.\ \ 0 \le k < 4\\
&\quad A'[((i+1)\ mod\ 6) \times 4 \times 2 + j \times 4 + k]\ +\\
&\quad A'[((i-1)\ mod\ 6) \times 4 \times 2 + j \times 4 + k]
\end{aligned}$$

As discussed above, there are other ways to achieve splitting of the problem across several computing sites. In general, the size of the array and the cost of accessing different architectural components drive the decision to break the problem up over processors, GPUs, threads, etc. [16, 17].

If a decision was made to break up the operations over different calculation units, the loop would be the same but the cost of performing the operation would be different. This decision is therefore completely cost-driven.

Continuing with *dimension lifting*, a choice might be made to use vector registers. This is, once again, a cost-driven decision, which may however be decided upon statically, prior to execution.

If we were to break our problem up over several processors and using vector registers, it would conceptually go from 2 dimensional to 4 dimensional, using indexing to access each resource. The same process can be applied to hardware components [11], e.g. pipelines, memories, buffers, etc., to achieve optimal throughput.

5 PDE solver test-case

Coordinate-free numerics [10, 14] is a high-level approach to writing solvers for PDEs. Solvers are written using high-level operators on abstract tensors. Take for instance Burgers' equation [7],

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = \nu \nabla^2 \vec{u},$$

where vector $\vec{u}$ denotes a time and space varying velocity vector, t is time, and the scalar ν is a viscosity coefficient. Burgers' equation is a PDE involving temporal ($\frac{\partial}{\partial t}$) and spatial ($\nabla$) derivative operations. Applying an explicit second order Runge-Kutta time-integration method, the coordinate-free time-integrated equation can be coded in Magnolia as follows.

```
procedure burgersTimestep
(upd u:Tensor1V, obs dt:R, obs nu:R) = {
  var u_t = nu * laplacian(u) )
          - dot(u,gradient(u));
  var u_substep = u + dt/2 * u_t;
  u_t = nu * laplacian(u_substep)
      - dot(u_substep,gradient(u_substep));
  u = u + dt * u_t;
};
```

Note how close this code follows the mathematical high-level formulation (5). We can lower the abstraction level of this code by linking it with a library for 3D cartesian coordinates based on continuous ringfields [15]. Next it can be linked with a library for finite difference methods choosing, e.g., stencils $\langle -\frac{1}{2}, 0, \frac{1}{2} \rangle$ and $\langle 1, -2, 1 \rangle$ for first and second order partial derivatives, respectively. This takes us to a code at the MoA level, consisting of rotate and maps of arithmetic operations [8]. With some reorganisation, we end up with the solver code below, expressed using MoA. The code calls the `snippet` six times forming one full time integration step, one call for each of the three dimensions of the problem times two due to the half-step in the time-integration. The variables `dt,nu,dx` are scalar (floating point). The first two come from the code above, while `dx` was introducd by the finite difference method. The variables `u0,u1,u2` are multiarrays (3D each), for each of the components of the 3D velocity vectorfield. These variables will be updated during the computation. The variables `c0,c1,c2,c3` and `c4` are numeric constants. Three temporary multiarray variables `v0,v1,v2` are computed in the first three `snippet` calls, due to the half-step. They are then used in the last three `snippet` calls to update `u0,u1,u2`.

```
procedure step
(upd u0:MA, upd u1:MA, upd u2:MA,
 obs nu:Float, obs dx:Float, obs dt:Float) {

  var c0 = 0.5/dx;
  var c1 = 1/dx/dx;
  var c2 = 2/dx/dx;
  var c3 = nu;
  var c4 = dt/2;

  var v0 = u0;
  var v1 = u1;
  var v2 = u2;
   call snippet(v0,u0,u0,u1,u2,c0,c1,c2,c3,c4);
   call snippet(v1,u1,u0,u1,u2,c0,c1,c2,c3,c4);
   call snippet(v2,u2,u0,u1,u2,c0,c1,c2,c3,c4);
   call snippet(u0,v0,v0,v1,v2,c0,c1,c2,c3,c4);
   call snippet(u1,v1,v0,v1,v2,c0,c1,c2,c3,c4);
   call snippet(u2,v2,v0,v1,v2,c0,c1,c2,c3,c4);
};
```

In the actual `snippet` code, `d1a,d2a,d1b,d2b,d1c,d2c` and `shift_v` are temporary multiarray variables. The `shift` function takes as first argument the multiarray being shifted, then the direction of the shift, and lastly the distance for the rotational shift.

```
procedure snippet
(upd u:MA, obs v:MA,
 obs u0:MA, obs u1:MA, obs u2:MA,
 obs c0:Float, obs c1:Float, obs c2:Float,
 obs c3:Float, obs c4:Float) {

  var shift_v = shift ( v, 0, -1 );
  var d1a = -c0 * shift_v;
  var d2a = c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 0, 1 );
  d1a = d1a + c0 * shift_v;
  d2a = d2a + c1 * shift_v;
```

```
    shift_v = shift ( v, 1, -1 );
    var d1b = -c0 * shift_v;
    var d2b = c1 * shift_v - c2 * u0;
    shift_v = shift ( v, 1, 1 );
    d1b = d1b + c0 * shift_v;
    d2b = d2b + c1 * shift_v;

    shift_v = shift ( v, 2, -1 );
    var d1c = -c0 * shift_v;
    var d2c = c1 * shift_v - c2 * u0;
    shift_v = shift ( v, 2, 1 );
    d1c = d1c + c0 * shift_v;
    d2c = d2c + c1 * shift_v;

    d1a = u0 * d1a + u1 * d1b + u2 * d1c;
    d2a = d2a + d2b + d2c;
    u = u + c4 * ( c3 * d2a - d1a);
};
```

In essence, `snippet` is computing $1/3$ of the half-step of the PDE, using common calls to rotate to compute one first and one second order partial derivative.

5.1 Reduction using MoA

Using the reduction rules defined in the ψ-calculus, and turning our `snippet` code into an expression, we can reduce the code to a DNF representation. In the following, we spell out some of the transformation steps. The equation

$$\begin{aligned} snippet = u + c_4 \times & \\ & (c_3 \times (c_1 \times (((-1)\ \theta_0\ v) + (1\ \theta_0\ v)\ + ((-1)\ \theta_1\ v) + \\ & (1\ \theta_1\ v) + ((-1)\ \theta_2\ v) + (1\ \theta_2\ v)) - 3c_2u_0)\ - c_0 \times \\ & (((1\ \theta_0\ v) - ((-1)\ \theta_0\ v))\ u_0 + \\ & ((1\ \theta_1\ v) - ((-1)\ \theta_1\ v))\ u_1 + \\ & ((1\ \theta_2\ v) - ((-1)\ \theta_2\ v))\ u_2)) \end{aligned}$$

is a transcription of the `snippet` code above.

We use the notation θ_x to denote a rotation around the x^{th} axis, represented in Magnolia by calls to `shift(multiarray, axis, offset)`.

The Magnolia implementation of the snippet makes heavy use of the multiarrays `d1x` and `d2x`, where `x` denotes the axis around which the multiarray is rotated in lexicographical order (`a` corresponds to the 0^{th} axis, `b` to the 1^{st} and so on). For the sake of easing into it, let us start by building a generic DNF representation for `d2x`. All the steps will be detailed explicitly in order to gain insights on what is needed and what is possible.

$$\begin{aligned} \langle\ i\ j\ k\ \rangle\ \psi\ d_{2x} = & \langle\ i\ j\ k\ \rangle\ \psi\ (c_1 \times ((-1)\ \theta_x\ v) + c_1 \times \\ & (1\ \theta_x\ v) - c_2 \times u_0) \end{aligned}$$

(distribute ψ over $+/-$)

$$\begin{aligned} & = \langle\ i\ j\ k\ \rangle\ \psi\ (c_1 \times ((-1)\ \theta_x\ v))\ + \langle\ i\ j\ k\ \rangle\ \psi \\ & (c_1 \times (1\ \theta_x\ v))\ - \langle\ i\ j\ k\ \rangle\ \psi\ (c_2 \times u_0) \end{aligned}$$

(extract constant factors)

$$\begin{aligned} & = c_1 \times (\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_x\ v)) + c_1 \times \\ & (\langle\ i\ j\ k\ \rangle\ \psi(1\ \theta_x\ v)) - c_2 \times (\langle\ i\ j\ k\ \rangle\ \psi\ u_0) \end{aligned}$$

(factorize by c_1)

$$\begin{aligned} & = c_1 \times (\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_x\ v)\ + \langle\ i\ j\ k\ \rangle\ \psi \\ & (1\ \theta_x\ v)) - c_2 \times (\langle\ i\ j\ k\ \rangle\ \psi\ u_0) \end{aligned}$$

Using the MoA's concatenation of index property, we can now define $\langle\ i\ \rangle\ \psi\ d_{2x}$. However, this is only reducible if $x = 0$. The reason is that to reduce an expression using a rotation on the x^{th} axis further, one needs to apply ψ with an index of at least $x + 1$ elements. Therefore, to reduce d_{21}, we need an index vector with at least 2 elements, while we need a total index containing 3 elements to reduce d_{22}. With that in mind, we can try to reduce d_{21}:

$$\begin{aligned} \langle\ i\ j\ \rangle\ \psi\ d_{21} = & c_1 \times (\langle\ i\ j\ \rangle\ \psi((-1)\ \theta_1\ v)\ + \langle\ i\ j\ \rangle\ \psi \\ & (1\ \theta_1\ v)) - c_2 \times (\langle\ i\ j\ \rangle\ \psi\ u_0) \end{aligned}$$

(reducing rotation)

$$\begin{aligned} & = c_1 \times (\langle\ i\ ((j-1)\ mod\ s_1)\ \rangle\ \psi\ v + \\ & \langle\ i\ ((j+1)\ mod\ s_1)\ \rangle\ \psi\ v) - \\ & c_2 \times (\langle\ i\ j\ \rangle\ \psi\ u_0) \end{aligned}$$

For $x = 2$, we apply the same process with a total index:

$$\begin{aligned} \langle\ i\ j\ k\ \rangle\ \psi\ d_{22} = & c_1 \times (\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_2\ v) + \\ & \langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_2\ v)) - c_2 \times (\langle\ i\ j\ k\ \rangle\ \psi\ u_0) \end{aligned}$$

(reducing rotation)

$$\begin{aligned} & = c_1 \times (\langle\ i\ j\ ((k-1)\ mod\ s_2)\ \rangle\ \psi\ v + \\ & \langle\ i\ j\ ((k+1)\ mod\ s_2)\ \rangle\ \psi\ v) - c_2 \times \\ & (\langle\ i\ j\ k\ \rangle\ \psi\ u_0) \end{aligned}$$

Now we can define the ONF of the expression, which is the form we will use in our actual code. Let's define it for d_{21}:

$$(rav\ d_{21})[\gamma(\langle\ i\ j\ \rangle\ ;\ \langle\ s_0\ s_1\ \rangle) \times s_2 + \iota s_2] = c_1 \times$$
$$((rav\ v)[\gamma(\langle\ i\ ((j-1)\ mod\ s_1)\ \rangle\ ;\ \langle\ s_0\ s_1\ \rangle) \times s_2 + \iota s_2] +$$
$$(rav\ v)[\gamma(\langle\ i\ ((j+1)\ mod\ s_1)\ \rangle\ ;\ \langle\ s_0\ s_1\ \rangle) \times s_2 + \iota s_2]) -$$
$$c_2 \times (rav\ u_0)[\gamma(\langle\ i\ j\ \rangle\ ;\ \langle\ s_0\ s_1\ \rangle) \times s_2 + \iota s_2]$$

(apply γ on both sides)

$$(rav\ d_{21})[(i \times s_1 \times s_2 + j \times s_2 + \iota s_2] = c_1 \times$$
$$((rav\ v)[i \times s_1 \times s_2 + ((j-1)\ mod\ s_1) \times s_2 + \iota s_2] +$$
$$(rav\ v)[i \times s_1 \times s_2 + ((j+1)\ mod\ s_1) \times s_2 + \iota s_2]) -$$
$$c_2 \times (rav\ u_0)[i \times s_1 \times s_2 + j \times s_2 + \iota s_2]$$

The optimization can be done similarly for d_{22}. The fact that d_{22} can only be reduced using a total index means that *snippet* too can only be fully reduced using a total index.

$$\langle\ i\ j\ k\ \rangle\ \psi\ snippet$$
$$= \langle\ i\ j\ k\ \rangle\ \psi\ (u + c_4 \times (c_3 \times (c_1 \times$$
$$(((-1)\ \theta_0\ v) + (1\ \theta_0\ v) + ((-1)\ \theta_1\ v) +$$
$$(1\ \theta_1\ v) + ((-1)\ \theta_2\ v) + (1\ \theta_2\ v)) -$$
$$3c_2u_0)\ - c_0(((1\ \theta_0\ v) - ((-1)\ \theta_0\ v))\ u_0 +$$
$$((1\ \theta_1\ v) - ((-1)\ \theta_1\ v))\ u_1\ + ((1\ \theta_2\ v) +$$
$$((-1)\ \theta_2\ v))\ u_2)))$$

(distribute ψ over $+$ and -)

$$= \langle\ i\ j\ k\ \rangle\ \psi\ u + (\langle\ i\ j\ k\ \rangle\ \psi\ c_4 \times (c_3 \times$$
$$(c_1 \times (((-1)\ \theta_0\ v) + (1\ \theta_0\ v) +$$
$$((-1)\ \theta_1\ v) + (1\ \theta_1\ v) + ((-1)\ \theta_2\ v) +$$
$$(1\ \theta_2\ v)) - 3c_2u_0)))\ - \langle\ i\ j\ k\ \rangle\ \psi$$
$$(c_0 \times (((1\ \theta_0\ v) - ((-1)\ \theta_0\ v))\ u_0 +$$
$$((1\ \theta_1\ v) - ((-1)\ \theta_1\ v))\ u_1 +$$
$$((1\ \theta_2\ v) - ((-1)\ \theta_2\ v))\ u_2)))$$

(extract constant c_4, c_3, and c_0)

$$= \langle\ i\ j\ k\ \rangle\ \psi\ u + c_4 \times (c_3 \times (\langle\ i\ j\ k\ \rangle\ \psi$$
$$(c_1 \times (((-1)\ \theta_0\ v) + (1\ \theta_0\ v) +$$
$$((-1)\ \theta_1\ v) + (1\ \theta_1\ v) +$$
$$((-1)\ \theta_2\ v) + (1\ \theta_2\ v)) - 3c_2u_0)) -$$
$$c_0 \times (\langle\ i\ j\ k\ \rangle\ \psi$$
$$(((1\ \theta_0\ v) - ((-1)\ \theta_0\ v))\ u_0 +$$
$$((1\ \theta_1\ v) - ((-1)\ \theta_1\ v))\ u_1 +$$
$$((1\ \theta_2\ v) - ((-1)\ \theta_2\ v))\ u_2)))$$

(distribute ψ over $+$, $\times$, and -)

$$= \langle\ i\ j\ k\ \rangle\ \psi\ u + c_4 \times (c_3 \times (\langle\ i\ j\ k\ \rangle\ \psi$$
$$(c_1 \times (((-1)\ \theta_0\ v) + (1\ \theta_0\ v) +$$
$$((-1)\ \theta_1\ v) + (1\ \theta_1\ v) +$$
$$((-1)\ \theta_2\ v) + (1\ \theta_2\ v))) -$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ (3c_2u_0)) - c_0 \times$$
$$(\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_0\ v) - ((-1)\ \theta_0\ v)) \times$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ u_0 +$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_1\ v) - ((-1)\ \theta_1\ v)) \times$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ u_1 +$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_2\ v) - ((-1)\ \theta_2\ v)) \times$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ u_2))$$

(extract constant factors c_1 and $3 \times c_2$)

$$= \langle\ i\ j\ k\ \rangle\ \psi\ u + c_4 \times (c_3 \times (c_1 \times$$
$$(\langle\ i\ j\ k\ \rangle\ \psi\ (((-1)\ \theta_0\ v) + (1\ \theta_0\ v) +$$
$$((-1)\ \theta_1\ v) + (1\ \theta_1\ v) + ((-1)\ \theta_2\ v) +$$
$$(1\ \theta_2\ v))) - 3c_2(\langle\ i\ j\ k\ \rangle\ \psi\ u_0))\ - c_0 \times$$
$$(\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_0\ v) - ((-1)\ \theta_0\ v)) \times$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ u_0 +$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_1\ v) - ((-1)\ \theta_1\ v)) \times$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ u_1 +$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_2\ v) - ((-1)\ \theta_2\ v)) \times$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ u_2))$$

(distribute ψ over $+$ and -)

$$= \langle\ i\ j\ k\ \rangle\ \psi\ u + c_4 \times (c_3 \times (c_1 \times$$
$$(\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_0\ v) +$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_0\ v) +$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_1\ v) +$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_1\ v) +$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_2\ v) +$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_2\ v)) - 3c_2$$
$$(\langle\ i\ j\ k\ \rangle\ \psi\ u_0))\ - c_0 \times$$
$$((\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_0\ v) -$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_0\ v)) \times$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ u_0 +$$
$$(\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_1\ v) -$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_1\ v)) \times$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ u_1 +$$
$$(\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_2\ v) -$$
$$\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_2\ v)) \times$$

$\langle\, i\ j\ k\,\rangle\ \psi\ u_2))$

(translate rotations into indexing)

$$
\begin{aligned}
&= \langle\, i\ j\ k\,\rangle\ \psi\ u + c_4 \times (c_3 \times (c_1 \times \\
&(\langle\, ((i-1)\ mod\ s_0)\ j\ k\,\rangle\ \psi\ v\ + \\
&\langle\, ((i+1)\ mod\ s_0)\ j\ k\,\rangle\ \psi\ v\ + \\
&\langle\, i\ ((j-1)\ mod\ s_1)\ k\,\rangle\ \psi\ v\ + \\
&\langle\, i\ ((j+1)\ mod\ s_1)\ k\,\rangle\ \psi\ v\ + \\
&\langle\, i\ j\ ((k-1)\ mod\ s_2)\,\rangle\ \psi\ v\ + \\
&\langle\, i\ j\ ((k+1)\ mod\ s_2)\,\rangle\ \psi\ v) - \\
&3c_2(\langle\, i\ j\ k\,\rangle\ \psi\ u_0))\ \ - c_0 \times \\
&(((\langle\, ((i+1)\ mod\ s_0)\ j\ k\,\rangle\ \psi\ v\ - \\
&\langle\, ((i-1)\ mod\ s_0)\ j\ k\,\rangle\ \psi\ v) \times \\
&\langle\, i\ j\ k\,\rangle\ \psi\ u_0\ + \\
&(\langle\, i\ ((j+1)\ mod\ s_1)\ k\,\rangle\ \psi\ v\ - \\
&\langle\, i\ ((j-1)\ mod\ s_1)\ k\,\rangle\ \psi\ v) \times \\
&\langle\, i\ j\ k\,\rangle\ \psi\ u_1 + \\
&(\langle\, i\ j\ ((k+1)\ mod\ s_2)\,\rangle\ \psi\ v\ - \\
&\langle\, i\ j\ ((k-1)\ mod\ s_2)\,\rangle\ \psi\ v) \times \\
&\langle\, i\ j\ k\,\rangle\ \psi\ u_2))
\end{aligned}
$$

In Magnolia, the DNF can be captured as such:

```
procedure snippetDNF(
upd u:MA, obs v:MA,
obs u0:MA, obs u1:MA, obs u2:MA,
obs c0:Float, obs c1:Float,
obs c2:Float, obs c3:Float, obs c4:Float,
obs mi:MI) {

  var s0 = shape0(v);
  var s1 = shape1(v);
  var s2 = shape2(v);

  u =
  psi(mi,u) + c4*(c3*(c1*(
  psi(mod0(mi-d0,s0),v) +
  psi(mod0(mi+d0,s0),v) +
  psi(mod1(mi-d1,s1),v) +
  psi(mod1(mi+d1,s1),v) +
  psi(mod2(mi-d2,s2),v) +
  psi(mod2(mi+d2,s2))) - 3*c2* psi(mi,u0)) -
  c0 * ((psi(mod0(mi+d0,s0),v) -
  psi(mod0(mi-d0,s0),v)) * psi(mi,u0) + (
  psi(mod1(mi+d1,s1),v) -
  psi(mod1(mi-d1,s1),v)) * psi(mi,u1) + (
  psi(mod2(mi+d2,s2),v) -
  psi(mod2(mi-d2,s2),v)) * psi(mi,u2) ));
}
```

Now, we can transform `snippet` into its ONF form:

$$
\begin{aligned}
&(rav\ snippet)[\gamma(\langle\, i\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] = \\
&(rav\ u)[\gamma(\langle\, i\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] + c_4 \times (c_3 \times (c_1 \times \\
&(rav\ v)[\gamma(\langle\, ((i-1)\ mod\ s_0)\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] + \\
&(rav\ v)[\gamma(\langle\, ((i+1)\ mod\ s_0)\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] + \\
&(rav\ v)[\gamma(\langle\, i\ ((j-1)\ mod\ s_1)\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] + \\
&(rav\ v)[\gamma(\langle\, i\ ((j+1)\ mod\ s_1)\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] + \\
&(rav\ v)[\gamma(\langle\, i\ j\ ((k-1)\ mod\ s_2)\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] + \\
&(rav\ v)[\gamma(\langle\, i\ j\ ((k+1)\ mod\ s_2)\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)]) - \\
&3c_2(rav\ u)[\gamma(\langle\, i\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)]\ \ - c_0 \times \\
&(((rav\ v)[\gamma(\langle\, ((i+1)\ mod\ s_0)\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] - \\
&(rav\ v)[\gamma(\langle\, ((i-1)\ mod\ s_0)\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)]) \times \\
&(rav\ u_0)[\gamma(\langle\, i\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)]\ + \\
&((rav\ v)[\gamma(\langle\, i\ ((j+1)\ mod\ s_1)\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] - \\
&(rav\ v)[\gamma(\langle\, i\ ((j-1)\ mod\ s_1)\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)]) \times \\
&(rav\ u_1)[\gamma(\langle\, i\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)]\ + \\
&((rav\ v)[\gamma(\langle\, i\ j\ ((k+1)\ mod\ s_2)\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)] - \\
&(rav\ v)[\gamma(\langle\, i\ j\ ((k-1)\ mod\ s_2)\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)]) \times \\
&(rav\ u_2)[\gamma(\langle\, i\ j\ k\,\rangle\ ;\ \langle\, s_0\ s_1\ s_2\,\rangle)]))
\end{aligned}
$$

This is how far we can go without specific information about the layout of the data in the memory and the architecture. The current form is still fully generic, with γ and *rav* parameterized over the layout. The Magnolia implementation of this generic form is as follows:

```
procedure moaONF (
upd u:MA,
obs v:MA,
obs u0:MA, obs u1:MA, obs u2:MA,
obs c0:Float, obs c1:Float,
obs c2:Float, obs c3:Float, obs c4:Float,
obs mi:MI ){

  var s0 = shape0(v);
  var s1 = shape1(v);
  var s2 = shape2(v);

  var newu =
  get(rav(u),gamma(mi,s)) + c4*(c3*(c_1*
  get(rav(v),gamma(mod0(mi-d0,s0),s)) +
  get(rav(v),gamma(mod0(mi+d0,s0),s)) +
  get(rav(v),gamma(mod1(mi-d1,s1),s)) +
  get(rav(v),gamma(mod1(mi+d1,s1),s)) +
  get(rav(v),gamma(mod2(mi-d2,s2),s)) +
    get(rav(v),gamma(mod2(mi+d2,s2),s))) -
    3 * c_2 get(rav(u),gamma(mi,s)) - c_0 *
    ((get(rav(v),gamma(mod0(mi+d0,s0),s)) -
    get(rav(v),gamma(mod0(mi-d0,s0),s))) *
    get(rav(u_0),gamma(mi,s)) +
    (get(rav(v),gamma(mod1(mi+d1,s1),s)) -
    get(rav(v),gamma(mod1(mi-d1,s1),s))) *
    get(rav(u_1),gamma(mi,s)) +
    (get(rav(v),gamma(mod2(mi+d2,s2),s)) -
    get(rav(v),gamma(mod2(mi-d2,s2),s))) *
    get(rav(u_2),gamma(mi,s))));

    set(rav(u),gamma(mi,s),newu);
  }
```

In Section 4.3, we defined the layout of the data as row-major. Thus we can optimize the expression further by expanding the calls to γ:

$$\begin{aligned}
(rav\ snippet)[i \times s_1 \times s_2 + j \times s_2 + k] = \\
(rav\ u)[i \times s_1 \times s_2 + j \times s_2 + k] + c_4 \times (c_3 \times (c_1 \times \\
(rav\ v)[((i-1)\ mod\ s_0) \times s_1 \times s_2 + j \times s_2 + k] + \\
(rav\ v)[((i+1)\ mod\ s_0) \times s_1 \times s_2 + j \times s_2 + k] + \\
(rav\ v)[i \times s_1 \times s_2 + ((j-1)\ mod\ s_1) \times s_2 + k] + \\
(rav\ v)[i \times s_1 \times s_2 + ((j+1)\ mod\ s_1) \times s_2 + k] + \\
(rav\ v)[i \times s_1 \times s_2 + j \times s_2 + ((k-1)\ mod\ s_2)] + \\
(rav\ v)[i \times s_1 \times s_2 + j \times s_2 + ((k+1)\ mod\ s_2)]) - \\
3c_2(rav\ u)[i \times s_1 \times s_2 + j \times s_2 + k]\ - c_0 \times \\
(((rav\ v)[((i+1)\ mod\ s_0) \times s_1 \times s_2 + j \times s_2 + k] - \\
(rav\ v)[((i-1)\ mod\ s_0) \times s_1 \times s_2 + j \times s_2 + k]) \times \\
(rav\ u_0)[i \times s_1 \times s_2 + j \times s_2 + k]\ + \\
((rav\ v)[i \times s_1 \times s_2 + ((j+1)\ mod\ s_1) \times s_2 + k] - \\
(rav\ v)[i \times s_1 \times s_2 + ((j-1)\ mod\ s_1) \times s_2 + k]) \times \\
(rav\ u_1)[i \times s_1 \times s_2 + j \times s_2 + k]\ + \\
((rav\ v)[i \times s_1 \times s_2 + j \times s_2 + ((k+1)\ mod\ s_2)] - \\
(rav\ v)[i \times s_1 \times s_2 + j \times s_2 + ((k-1)\ mod\ s_2)]) \times \\
(rav\ u_2)[i \times s_1 \times s_2 + j \times s_2 + k]))
\end{aligned}$$

At this point, as indicated in section 4.3, we can convert our expression into several subexpressions in order to distinguish the general case from anomalies (i.e cases that require the modulo operation to be applied on any axis). This general case is in ONF and we can use it for code generation or to perform additional transformations, specifically dimension lifting.

6 Conclusion

Through the full analysis of an FDM solver of a PDE, we were able to extract a rewriting subsystem most relevant to our specific problem out of the rewriting rules provided by the ψ-calculus. Then, we proved that this particular set of rewriting rules constitutes a canonical rewriting system, getting one step closer to fully automating the optimization of array computations using the MoA formalism.

We are now working on the implementation of our optimizations to measure their impact on the performance of the solver for different architectures, and can report results in the near future.

By working out an approach from high level coordinate-free PDEs down to preparing for data layout and code optimization using MoA as an intermediate layer through the full exploration of a relevant example, we pave the way for building similar systems for any problem of the same category. High-efficiency code can thus easily be explored and generated from a unique high-level abstraction and potentially different implementation algorithms, layouts of data or hardware architectures.

Because tensors dominate a significant portion of science, future work may focus on figuring out what properties can be deduced from the complete ψ-calculus rewriting system with a goal to extend this currently problem-oriented approach towards a fully automated problem-independent optimization tool based on MoA.

Given the scale of the ecosystem impacted by this kind of work, such prospects are very attractive.

References

[1] Philip Samuel Abrams. 1970. *An APL machine.* Ph.D. Dissertation. Stanford University, Stanford, CA, USA.

[2] Evrim Acar, Animashree Anandkumar, Lenore Mullin, Sebnem Rusitschka, and Volker Tresp. 2016. Tensor Computing for Internet of Things (Dagstuhl Perspectives Workshop 16152). *Dagstuhl Reports* 6, 4 (2016), 57–79. https://doi.org/10.4230/DagRep.6.4.57

[3] Evrim Acar, Animashree Anandkumar, Lenore Mullin, Sebnem Rusitschka, and Volker Tresp. 2018. Tensor Computing for Internet of Things (Dagstuhl Perspectives Workshop 16152). *Dagstuhl Manifestos* 7, 1 (2018), 52–68. https://doi.org/10.4230/DagMan.7.1.52

[4] Brett W. Bader, Tamara G. Kolda, et al. 2015. MATLAB Tensor Toolbox Version 2.6. Available online. http://www.sandia.gov/~tgkolda/TensorToolbox/

[5] Anya Helene Bagge. 2009. *Constructs & Concepts: Language Design for Flexibility and Reliability.* Ph.D. Dissertation. Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway, PB 7803, 5020 Bergen, Norway. http://www.ii.uib.no/~anya/phd/

[6] Klaus Berkling. 1990. *Arrays and the Lambda Calculus.* Technical Report 93. Electrical Engineering and Computer Science Technical Reports.

[7] Johannes Martinus Burgers. 1948. A mathematical model illustrating the theory of turbulence. In *Advances in applied*

mechanics. Vol. 1. Elsevier, 171–199.

[8] Eva Burrows, Helmer André Friis, and Magne Haveraaen. 2018. An Array API for Finite Difference Methods. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2018)*. ACM, New York, NY, USA, 59–66. https://doi.org/10.1145/3219753.3219761

[9] J. A. Crotinger et al. 2000. Generic Programming in POOMA and PETE. *Lecture Notes in Computer Science* 1766 (2000).

[10] Philip W. Grant, Magne Haveraaen, and Michael F. Webster. 2000. Coordinate free programming of computational fluid dynamics problems. *Scientific Programming* 8, 4 (2000), 211–230. https://doi.org/10.1155/2000/419840

[11] Ian Grout and Lenore Mullin. 2018. Hardware Considerations for Tensor Implementation and Analysis Using the Field Programmable Gate Array. *Electronics* 7, 11 (2018). https://doi.org/10.3390/electronics7110320

[12] John L. Gustafson and Lenore M. Mullin. 2017. Tensors Come of Age: Why the AI Revolution will help HPC. *CoRR* abs/1709.09108 (2017). arXiv:1709.09108 http://arxiv.org/abs/1709.09108

[13] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[14] Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. 1999. Formal Software Engineering for Computational Modelling. *Nord. J. Comput.* 6, 3 (1999), 241–270.

[15] Magne Haveraaen, Helmer André Friis, and Hans Munthe-Kaas. 2005. Computable Scalar Fields: a basis for PDE software. *Journal of Logic and Algebraic Programming* 65, 1 (September-October 2005), 36–49. https://doi.org/10.1016/j.jlap.2004.12.001

[16] H. B. Hunt III, L. Mullin, and D. J. Rosenkrantz. 1998. *Experimental Design and Development of a Polyalgorithm for the FFT*. Technical Report 98–5. University at Albany, Department of Computer Science.

[17] Harry B. Hunt III, Lenore R. Mullin, Daniel J. Rosenkrantz, and James E. Raynolds. 2008. A Transformation–Based Approach for the Design of Parallel/Distributed Scientific Software: the FFT. *CoRR* abs/0811.2535 (2008). arXiv:0811.2535 http://arxiv.org/abs/0811.2535

[18] K. Iverson. 1962. *A Programming Language*. John Wiley and Sons, Inc. New York.

[19] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[20] J. W. Klop, Marc Bezem, and R. C. De Vrijer (Eds.). 2001. *Term Rewriting Systems*. Cambridge University Press, New York, NY, USA.

[21] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (September 2009), 455–500. https://doi.org/10.1137/07070111X

[22] Lenore Mullin. 1988. *A Mathematics of Arrays*. Ph.D. Dissertation.

[23] Lenore Mullin and Michael Jenkins. 1996. Effective Data Parallel Computation Using the Psi-Calculus. *Concurrency Journal* (1996).

[24] L Mullin and J Raynolds. 2014. *Scalable, Portable, Verifiable Kronecker Products on Multi-scale Computers*. Constraint Programming and Decision Making. Studies in Computational Intelligence, Vol. 539. Springer, Cham.

[25] L. Mullin, E. Rutledge, and R. Bond. 2002. Monolithic Compiler Experiments using C++ Expression Templates. In *Proceedings of the High Performance Embedded Computing Workshop (HPEC 2002)*. MIT Lincoln Lab, Lexington, MA.

[26] Lenore M. Restifo Mullin, Ashok Krishnamurthi, and Deepa Iyengar. 1988. The Design And Development of a Basis, alpha_L, for Formal Functional Programming Languages with Arrays Based on a Mathematics of Arrays. In *Proceedings of the International Conference on Parallel Processing, ICPP '88, The Pennsylvania State University, University Park, PA, USA, August 1988. Volume 2: Software.*

[27] L.M. R. Mullin. 1991. Psi, the Indexing Function: A Basis for FFP with Arrays. In *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers.

[28] Lenore R. Mullin. 2005. A uniform way of reasoning about array-based computation in radar: Algebraically connecting the hardware/software boundary. *Digital Signal Processing* 15, 5 (2005), 466–520.

[29] L. R. Mullin, D. Dooling, E. Sandberg, and S. Thibault. 1993. Formal Methods for Scheduling, Routing and Communication Protoc ol. In *Proceedings of the Second International Symposium on High Performance Distributed Computing (HPDC-2)*. IEEE Computer Society.

[30] L. R. Mullin, D. J. Rosenkrantz, H. B. Hunt III, and X. Luo. 2003. Efficient Radar Processing Via Array and Index Algebras. In *Proceedings First Workshop on Optimizations for DSP and Embedded Systems (ODES)*. San Francisco, CA, 1–12.

[31] Paul Chang and Lenore R. Mullin. 2002. An Optimized QR Factorization Algorithm based on a Calculus of Indexing. DOI: 10.13140/2.1.4938.2722.

[32] Jeremy G. Siek and Andrew Lumsdaine. 1998. The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*. Springer-Verlag, London, UK, UK. http://dl.acm.org/citation.cfm?id=646894.709706

[33] Google Brain Team. 2015. https://www.tensorflow.org/.

[34] Hai-Chen Tu and Alan J. Perlis. 1986. FAC: A Functional APL Language. *IEEE Software* 3, 1 (Jan. 1986), 36–45.

CHAPTER 8

A Mathemathics of Arrays evaluated

8.1 introduction

We carry on from the previous chapter to getting a sense of what improvement we can expect from our application of MOA. For this endeavour we proceed as follows: First we port the original solver code, i.e. before applying MOA, to C. We then do the same for the solver code after applying MOA. We consider further optimizations. Note we do not make any considerations as to what the algorithm we are implementing is doing; it is simply a benchmarking tool, we are concerned with how it is doing things.

8.2 The original implementation

We started out with an implementation of a PDE solver in Magnolia. Which we elect to port to C as the existing compiler toolchain for Magnolia may introduce some undesirable overhead. The algorithm consists of a step which takes as parameters three multidimensional arrays along with some constants and then alter the arrays in a kernel of computation called snippet.

```
//For the parameter upd u0:MA, u0 is a multidimensional
//array that is upd(atable). The upd qualifier means this
//procedure parameter can be read from and written to;
//a call to the procedure can alter the value of a variable
//passed to it as an effect of the call. A qualifier of
//obs(erve) means the parameter can only be read from.

procedure step(upd u0:MA, upd u1:MA, upd u2:MA,
               obs nu:Float, obs dx:Float, obs dt:Float) {

  var c0 = 0.5/dx;
  var c1 = 1/dx/dx;
  var c2 = 2/dx/dx;
  var c3 = nu;
  var c4 = dt/2;

  var v0 = u0;
  var v1 = u1;
  var v2 = u2;

  call snippet(v0,u0,u0,u1,u2,c0,c1,c2,c3,c4);
  call snippet(v1,u1,u0,u1,u2,c0,c1,c2,c3,c4);
  call snippet(v2,u2,u0,u1,u2,c0,c1,c2,c3,c4);
  call snippet(u0,v0,v0,v1,v2,c0,c1,c2,c3,c4);
  call snippet(u1,v1,v0,v1,v2,c0,c1,c2,c3,c4);
  call snippet(u2,v2,v0,v1,v2,c0,c1,c2,c3,c4);
};
```

The kernel of computation called snippet should be fairly straightforward to understand although there are some operations that perhaps require some clarification. For instance the shift operation moves the entire contents of the array indicated by its first parameter, in the dimension of the second parameter, by a distance indicated by the third parameter. Elements that fall outside of bounds wrap around, i.e. modular wraparound. One may notice this is perhaps not ideal; it is better to not move any data at all and instead alter indexing. However given that this was orignally code that was run on a GPU, the penalty in that case would perhaps have been negligible.

```
procedure snippet(upd u:MA, obs v:MA,
obs u0:MA, obs u1:MA, obs u2:MA,
obs c0:Float, obs c1:Float, obs c2:Float,
obs c3:Float, obs c4:Float) {
  var shift_v = shift ( v, 0, -1 );
  var d1a = -c0  * shift_v;
  var d2a =  c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 0, 1 );
  d1a = d1a + c0  * shift_v;
  d2a = d2a + c1 * shift_v;

  shift_v = shift ( v, 1, -1 );
  var d1b = -c0  * shift_v;
  var d2b = c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 1, 1 );
  d1b = d1b + c0  * shift_v;
  d2b = d2b + c1 * shift_v;

  shift_v = shift ( v, 2, -1 );
  var d1c = -c0  * shift_v;
  var d2c = c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 2, 1 );
  d1c = d1c + c0  * shift_v;
  d2c = d2c + c1 * shift_v;

  d1a = u0 * d1a + u1 * d1b + u2 * d1c;
  d2a = d2a + d2b + d2c;
  u = u + c4 * ( c3 * d2a - d1a);
};
```

In porting this to C(conforming to at least C99) we omit details on the step code as it is uninteresting. We only provide the snippet/kernel code here, but will make available the entire source for a complete implementation. For the rest of the snippets of code we consider that the arrays are of the shape $S = \langle s_0, s_1, s_2 \rangle$, thus we define $asize = s_0 s_1 s_2$ to be the total amount of elements in the array and we lay them out in a row major order in memory and access them with a fixed $gamma(i, j, k) = is_1 s_2 + js_2 + k$.

Listing 8.1: C port A

```
#define APPLY(x) for(int x=0;x<asize;x++)
  static void snippetorig(
  double *u,
  double *restrict v,
  double *restrict u0,
  double *restrict u1,
  double *restrict u2,
  double c0, double c1,
  double c2, double c3,
  double c4,
  struct temps* t)
{
  //block 1

  //var shift_v = shift ( v, 0, -1 );
  shift0(v,t->shift_v, -1);

  //var d1a = -c0 * shift_v;
  APPLY(i) {
    (t->d1a)[i] = (t->shift_v)[i]*(-c0);
  }

  //var d2a = c1 * shift_v - c2 * u0;
  APPLY(i) {
    (t->d1a)[i] = (t->shift_v)[i]*c1 - c2*u0[i];
  }

  //shift_v = shift ( v, 0, 1 );
  shift0(v,t->shift_v,1);

  //d1a = d1a + c0 * shift_v;
  APPLY(i) {
    (t->d1a)[i] = (t->d1a)[i] + c0*(t->shift_v)[i];
  }

  //d2a = d2a + c1 * shift_v;
  APPLY(i) {
    (t->d2a)[i] = (t->d2a)[i]+c1*(t->shift_v)[i];
  }
  //Block 2

  //shift_v = shift ( v, 1, -1 );
  shift1(v,t->shift_v,-1);

  //var d1b = - c0 * shift_v;
  APPLY(i){
    (t->d1b)[i] = -c0*(t->shift_v)[i];
  }

  //var d2b = c1 * shift_v - c2 * u0;
```

```
APPLY(i){
  (t->d2b)[i] = c1*(t->shift_v)[i] - c2*u0[i];
}
// shift_v =  shift ( v, 1, 1 );
shift1(v,t->shift_v ,1);

//d1b = d1b + c0 * shift_v
APPLY(i) {
  (t->d1b)[i] = c0*(t->shift_v)[i];
}

//d2b = d2b + c0 * shift_v
APPLY(i) {
  (t->d2b)[i] = (t->d2b)[i] + c0*(t->shift_v[i]);
}
//Block 3

//shift_v = shift ( v, 2, -1 );
shift2(v,t->shift_v ,-1);

//var d1c = - c0 * shift_v;
APPLY(i) {
  (t->d1c)[i] = -c0*(t->shift_v)[i];
}

//var d2c = c1 * shift_v - c2 * u0;
APPLY(i) {
  (t->d2c)[i] = c1*(t->shift_v)[i] - c2*u0[i];
}
// shift_v =  shift ( v, 2, 1 );
shift2(v,t->shift_v ,1);

//d1c = d1c + c0 * shift_v
APPLY(i) {
  (t->d1c)[i] = (t->d1c)[i] + c0*(t->shift_v)[i];
}

//d2c = d2c + c0 * shift_v
APPLY(i) {
  (t->d2c)[i] = (t->d2c)[i] + c0*(t->shift_v)[i];
}

//d1a = u0 * d1a + u1 * d1b + u2 * d1c;
```

```
  APPLY(i) {
    (t->d1a)[i] = u0[i]*(t->d1a)[i] + u1[i]*(t->d1b)[i] +
    u2[i]*(t->d1c)[i];
  }

  //d2a = d2a + d2b + d2c;
  APPLY(i) {
    (t->d2a)[i] = (t->d2a)[i] + (t->d2b)[i] + (t->d2c)[i];
  }

  //u = u + c4 * ( c3 * d2a - d1a);
  APPLY(i) {
    u[i] = u[i] + c4 * (c3 * (t->d2a)[i] - (t->d1a)[i]);
  }
}
```

The shifting operation is straightforward, but one is included for completeness.

Listing 8.2: shifting along the 0th axis

```
static inline void shift0(double *v, double *out,
                          int distance)
{
  int ijk=0;
  for(int i=0;i<s0;i++) {
    for(int j=0;j<s1;j++) {
      for(int k=0;k<s2;k++,ijk++) {
        out[ijk] = v[gamma(mod(i+distance,s0),j,k)];
      }
    }
  }
}
```

We note that this implementation mainly has two immediately discernable undesirable properties. It does several passes over the arrays spread over many operations and it moves quite a lot of data around in its shift operations. It also requires the use of temporaries in addition to the arrays it operates on.

8.3 A first look at applying MOA

In our conference paper we ended at one step after the following stage, i.e. we had expanded gamma. The following is the direct port of the resulting MOA interpretation of the kernel of computation called snippet. Note that we may omit the parameters to the kernel snippet in some of the following examples. Whenever this is the case they are simply the same as in preceding examples.

Listing 8.3: C port B

```
// row major indexing
static inline int gamma(int i, int j, int k)
{
  return  i*s1*s2 + j*s2 + k;
}

static void snippetONF1(
double *u,
double *restrict v,
double *restrict u0,
double *restrict u1,
double *restrict u2,
double c0,double c1,
double c2,double c3,
double c4)
{
  for (int i=0; i<s0; i++) {
    for (int j=0; j<s1; j++) {
      for (int k=0; k<s2; k++) {
        u[gamma(i,j,k)] =
        u[gamma(i,j,k)] + c4 * (c3 * (c1 *
        v[gamma((mod(i-1,s0)),j,k)] +
        v[gamma((mod(i+1,s0)),j,k)] +
        v[gamma(i,(mod(j-1,s1)),k)] +
        v[gamma(i,(mod(j+1,s1)),k)] +
        v[gamma(i,j,(mod(k-1,s2)))] +
        v[gamma(i,j,(mod(k+1,s2)))]) -
        3 * c2 * u[gamma(i,j,k)] - c0 *
        ((v[gamma((mod(i+1,s0)),j,k)] -
        v[gamma((mod(i-1,s0)),j,k)]) *
        u0[gamma(i,j,k)] +
        (v[gamma(i,(mod(j+1,s1)),k)] -
        v[gamma(i,(mod(j-1,s1)),k)]) *
        u1[gamma(i,j,k)] +
        (v[gamma(i,j,(mod(k+1,s2)))] -
        v[gamma(i,j,(mod(k-1,s2)))]) *
        u2[gamma(i,j,k)]));
      }
    }
  }
}
```

Comparing this to the MOA expression reached in the conference paper we see it is essentially a one-to-one translation. Comparing it to the original code 8.1 the immediate difference is that it does one pass over the arrays. We have collected operations into one pass instead of over several passes, one per operation. We opt to make this a little more human-friendly by extracting some subexpressions.

Listing 8.4: exctracting subexpressions

```
        u[gamma(i,j,k)] =
        u[gamma(i,j,k)] + c4 * (c3 *

        (c1 *
        v[gamma((mod(i-1,s0)),j,k)] +
        v[gamma((mod(i+1,s0)),j,k)] +
        v[gamma(i,(mod(j-1,s1)),k)] +
        v[gamma(i,(mod(j+1,s1)),k)] +
        v[gamma(i,j,(mod(k-1,s2)))] +
        v[gamma(i,j,(mod(k+1,s2)))])

        - 3 * c2 * u[gamma(i,j,k)] - c0 *

        ((v[gamma((mod(i+1,s0)),j,k)] -
        v[gamma((mod(i-1,s0)),j,k)]) *
        u0[gamma(i,j,k)] +

        (v[gamma(i,(mod(j+1,s1)),k)] -
        v[gamma(i,(mod(j-1,s1)),k)]) *
        u1[gamma(i,j,k)] +

        (v[gamma(i,j,(mod(k+1,s2)))] -
        v[gamma(i,j,(mod(k-1,s2)))]) *
        u2[gamma(i,j,k)]));
```

Thus we obtain a more readable form for further exploration. Again, note nothing but readability has changed here.

Listing 8.5: C port C

```
static void snippetONF1experimental1(...) {
  int i,j,k;
  double vijnk, vijpk, vnijk, vpijk, vinjk, vipjk;
  double m,n;

  for (i=0; i<s0; i++) {
    for (j=0; j<s1; j++) {
      for (k=0; k<s2; k++) {
        vijnk = v[gamma(i,j,mod(k-1,s2))];
        vijpk = v[gamma(i,j,mod(k+1,s2))];
        vnijk = v[gamma(mod(i-1,s0),j,k)];
        vpijk = v[gamma(mod(i+1,s0),j,k)];
        vinjk = v[gamma(i,mod(j-1,s1),k)];
        vipjk = v[gamma(i,mod(j+1,s1),k)];

        m = c1 * vnijk + vpijk + vinjk +
                 vipjk + vijnk + vijpk;
        n = (vpijk - vnijk) * u0[gamma(i,j,k)] +
            (vipjk - vinjk) * u1[gamma(i,j,k)] +
            (vijpk - vijnk) * u2[gamma(i,j,k)];

        u[gamma(i,j,k)] +=
        c4 * (c3 * m - 3 * c2 * u[gamma(i,j,k)] - c0 * n);

      }
    }
  }
}
```

8.4 Further optimizations

We now make an attempt to optimize the code from 8.5. What we may do is to expand the gamma index calculations and avoid needless recomputations in the looping. Sometimes this is referred to as hoisting recomputations out of loops. This however is really expected to be done automatically by the compiler when optimizations are enabled.

Listing 8.6: C port D

```
static void snippetONF1experimental2(...) {
  int i,j,k, ijk;
  double vijnk, vijpk, vnijk, vpijk, vinjk, vipjk;
  double m,n;

  int gi, gj, pmgi, nmgi, pmgj, nmgj;

  ijk =0;
  for (i=0; i<s0; i++) {
    gi = i*s1*s2;
    nmgi = mod(i-1,s0)*s1*s2;
    pmgi = mod(i+1,s0)*s1*s2;

    for (j=0; j<s1; j++) {
      gj = j*s1;
      nmgj = mod(j-1,s1)*s1;
      pmgj = mod(j+1,s1)*s1;

      for (k=0; k<s2; k++,ijk++) {
        vijnk = v[gi+gj+mod(k-1,s2)];
        vijpk = v[gi+gj+mod(k+1,s2)];

        vnijk = v[nmgi+gj+k];
        vpijk = v[pmgi+gj+k];

        vinjk = v[gi+nmgj+k];
        vipjk = v[gi+pmgj+k];

        m = c1 * vnijk + vpijk + vinjk +
                 vipjk + vijnk + vijpk;
        n = (vpijk - vnijk) * u0[ijk] +
            (vipjk - vinjk) * u1[ijk] +
            (vijpk - vijnk) * u2[ijk];

        u[ijk] += c4 * (c3 * m - 3 * c2 * u[ijk] - c0 * n);
      }
    }
  }
}
```

8.5 Final optimizations

We now make more drastic transformations. First, we see that modular arithmethic operations only interfere for the cases where we are at the beginning or end of the bounds of an array. Thus it then makes sense to split the looping domain in three parts:

$$\begin{aligned}\langle 0,0,0\rangle &\rightarrow \langle 1,1,1\rangle\\ \langle 1,1,1\rangle &\rightarrow \langle s_0-2,s_1-2,s_2-2\rangle\\ \langle s_0-2,s_1-2,s_2-2\rangle &\rightarrow \langle s_0-1,s_1-1,s_2-1\rangle\end{aligned}$$

Furthermore seeing as we are iterating through the arrays in a lexicographical order on indices(from right component to left) $\langle i,j,k\rangle$ we know that if $\langle i_x,j_x,k_x\rangle < \langle i_y,j_y,k_y\rangle$ such that $\langle i_y,j_y,k_y\rangle$ is the least such index that satisfies that condition then $gamma(i_y,j_y,k_y) = gamma(i_x,j_x,k_x)+1$. In the cases where we cannot eliminate modular arithmethic we still need the indices. We do this by keeping counting them upward in sync with a flat incrementation. i.e. we keep $\langle i,j,k\rangle$ in sync with $gamma(i,j,k) = ijk$.

Listing 8.7: C port E

```
static void snippetONF1experimental3(...) {
  double vijnk, vijpk, vnijk, vpijk, vinjk, vipjk;
  int gi, gj;
  double m,n;
  struct lpvar l;
  int start,stop;

  // loop (i,j,k) from (0,0,0) to (1,1,1)
  start = 0;
  stop = gamma(1,1,1);
  l = (struct lpvar){0,0,0};

  // ijk = i*s2*s1+j*s2+k
  for (int ijk=start;ijk<stop; ijk++,succ(&l)) {

    gi = (l.i)*s1*s2;
    gj = (l.j)*s2;

    vijnk = v[gi+gj+mod(l.k-1,s2)];
    vijpk = v[gi+gj+mod(l.k+1,s2)];
    vnijk = v[mod(l.i-1,s0)*s1*s2+gj+l.k];
    vpijk = v[mod(l.i+1,s0)*s1*s2+gj+l.k];
    vinjk = v[gi+mod(l.j-1,s1)*s2+l.k];
    vipjk = v[gi+mod(l.j+1,s1)*s2+l.k];

   m = c1 * vnijk + vpijk + vinjk + vipjk + vijnk + vijpk;
   n = (vpijk - vnijk) * u0[ijk] +
     (vipjk - vinjk) * u1[ijk] +
     (vijpk - vijnk) * u2[ijk];

   u[ijk] += c4 * (c3 * m - 3 * c2 * u[ijk] - c0 * n);
}

// loop (i,j,k) from (1,1,1) to (s2-2,s1-2,s0-2)
start = stop;
stop = gamma(s0-2,s1-2,s2-2);
for (int ijk=start;ijk < stop; ijk++) {
    vijnk = v[ijk-1];
    vijpk = v[ijk+1];
    vnijk = v[ijk-s1*s2];
    vpijk = v[ijk+s1*s2];
    vinjk = v[ijk-s2];
    vipjk = v[ijk+s2];

   m = c1 * vnijk + vpijk + vinjk + vipjk + vijnk + vijpk;
   n = (vpijk - vnijk) * u0[ijk] +
     (vipjk - vinjk) * u1[ijk] +
     (vijpk - vijnk) * u2[ijk];

   u[ijk] += c4 * (c3 * m - 3 * c2 * u[ijk] - c0 * n);
}

// loop (i,j,k) from (s2-2,s1-2,s0-2) to (s2-1,s1-1,s0-1)
start = stop;
stop = gamma(s0-1,s1-1,s2-1);
l = (struct lpvar){s0-2,s1-2,s2-2};

for (int ijk=start;ijk<stop; ijk++,succ(&l)) {

    gi = (l.i)*s1*s2;
    gj = (l.j)*s2;

    vijnk = v[gi+gj+mod(l.k-1,s2)];
    vijpk = v[gi+gj+mod(l.k+1,s2)];
    vnijk = v[mod(l.i-1,s0)*s1*s2+gj+l.k];
    vpijk = v[mod(l.i+1,s0)*s1*s2+gj+l.k];
    vinjk = v[gi+mod(l.j-1,s1)*s2+l.k];
    vipjk = v[gi+mod(l.j+1,s1)*s2+l.k];

    m = c1 * vnijk + vpijk + vinjk +
             vipjk + vijnk + vijpk;
    n = (vpijk - vnijk) * u0[ijk] +
        (vipjk - vinjk) * u1[ijk] +
        (vijpk - vijnk) * u2[ijk];

    u[ijk] +=
    c4 * (c3 * m - 3 * c2 * u[ijk] - c0 * n);
  }
}
```

For completeness we include the $\langle i, j, k \rangle$ incrementation logic.

Listing 8.8: incrementation logic

```
struct lpvar {
  int i;
  int j;
  int k;
};

static void inline succ(struct lpvar* l){
  l->k++;
  if((l->k)==s2){
    l->k=0;
    l->j++;
    if((l->j)==s1) {
      l->j=0;
      l->i++;
      if((l->i)==s0) {
        l->i=0;
      }
    }
  }
}
```

8.6 Benchmarking results

We now turn to measuring the effects of these optimizations. The problem size is fixed as arrays of shape $\langle 50, 50, 50\rangle$ i.e. of $50 \cdot 50 \cdot 50 = 125000$ elements. The step code is run with the differently optimized kernels 50 times in repetition and timed. That is timing is done as follows:

Listing 8.9: benchmarking timing

```
// One core ONF
memcpy(chunk, start, total*sizeof(double));
printf("ONF_1-core\n");
begin = clock();
for(int i=0;i<50;i++)
  step(u0,u1,u2,s_nu,s_dx,s_dt,snippetONF1);
end = clock();
tspent = (double)(end - begin) / CLOCKS_PER_SEC;
printf("time:%lf\n",tspent);
```

The following results measured in seconds are produced with gcc 9.2.1 and clang 8.0.0 on an i5-8250U Intel CPU.

Table 8.1: gcc

	-O0	-O1	-O2	-O3
A	3.182	0.717	0.648	0.550
B	3.474	0.374	0.360	0.356
C	2.187	0.373	0.367	0.369
D	0.834	0.371	0.366	0.366
E	0.529	0.375	0.377	0.087

Table 8.2: clang

	-O0	-O1	-O2	-O3
A	3.121	0.893	0.525	0.543
B	3.631	0.629	0.365	0.365
C	2.281	0.640	0.363	0.363
D	0.847	0.425	0.364	0.364
E	0.530	0.383	0.254	0.254

8.7 Introducing data redundancy

Taking another look at 8.3 we see that modular arithmetic only needs to be done at the boundaries of arrays. A technique to eliminate such boundary cases is by padding the array with its wraparound elements. That is if we have the array of values $\langle x_0, \ldots, x_{n-1} \rangle$ and we are attempting to retrieve, in an iteration through the array, element $i + 1 \, mod \, n$ and $i - 1 \, mod \, n$ that is the same as iterating through the array $\langle x'_0, x'_1, \ldots, x'_n, x'_{n+1} \rangle$ where $x'_0 = x_{n-1}, x'_{n+1} = x_0$ and the middle elements are the old array, i.e. for $1 \leq k \leq n$ we have $x'_k = x_{k-1}$. We must change our iteration from $i = 0, \ldots, i = n - 1$ to $i = 1, \ldots, i = n$. We can then change the retrievals of $i + 1 \, mod \, n$ and $i - 1 \, mod \, n$ simply to $i + 1$ and $i - 1$. In this little example we have bumped up the size from n to $n + 2$ but have eliminated n instances of modular arithmethic. We will refer to this technique of data redundancy as padding.

For our case in 8.3 we pad the contiguous chunks of the last dimension. We have a cubical array of shape $\langle s, s, s \rangle$ and thus total size s^3 so we get a shape with padding of $\langle s, s, s + 2 \rangle$ and a total size of $s^3 + 2s^2$.

Listing 8.10: Padding

```
static inline int gamma2(int i, int j, int k)
{
  return  i*s1*s2padded + j*s2padded + k;
}
static void SC_PADDING (...) {
  for (int i=0; i<s0; i++) {
    for (int j=0; j<s1; j++) {
      for (int k=1; k<s2+1; k++) {
        u[gamma2(i,j,k)] =
        u[gamma2(i,j,k)] + c4 * (c3 * (c1 *
        v[gamma2((mod(i-1,s0)),j,k)] +
        v[gamma2((mod(i+1,s0)),j,k)] +
        v[gamma2(i,(mod(j-1,s1)),k)] +
        v[gamma2(i,(mod(j+1,s1)),k)] +
        v[gamma2(i,j,k-1)] +
        v[gamma2(i,j,k+1)]) -
        3 * c2 * u[gamma2(i,j,k)] - c0 *
        ((v[gamma2((mod(i+1,s0)),j,k)] -
        v[gamma2((mod(i-1,s0)),j,k)]) *
        u0[gamma2(i,j,k)] +
        (v[gamma2(i,(mod(j+1,s1)),k)] -
        v[gamma2(i,(mod(j-1,s1)),k)]) *
        u1[gamma2(i,j,k)] +
          (v[gamma2(i,j,k+1)] -
          v[gamma2(i,j,k-1)]) *
          u2[gamma2(i,j,k)]));
        }
        // update the padding values
        u[gamma2(i,j,0)] = u[gamma2(i,j,s2)];
        u[gamma2(i,j,s2+1)] = u[gamma2(i,j,0)];
      }
    }
}
```

As can be seen in 8.10 we have eliminated modular arithmethic for the looping variable of the last dimension at the cost of a one time expansion preprocessing phase(not shown) where the 3-dimensional array is padded, the need to update the padding values as values are changed and upon completion a one time shortening to remove the padding(not shown).

We benchmark 8.10 against 8.3 with various array sizes. In this table we list the unpadded size s meaning we are operating with $\langle s, s, s\rangle$ shaped arrays and the padded array is a $\langle s, s, s + 2\rangle$ shaped array. The measurements are in seconds and were done on a Intel(R) Xeon(R) Gold 6130 CPU running at 2.10GHz.

Table 8.3: gcc 8.2.0 -O3

s	unpadded	padded
50	0.128	0.079
128	3.119	2.586
256	25.90	22.09
512	216.6	199.7

In principle this padding procedure should be able to be done on all dimensions at once however the cost of maintaining the padding values and the increase in memory usae may outweigh its benefits.

Padding is likely a more significant improvement in a multicore or multiprocessor utilizing implementation of the problem where access far apart incurs the cost of either memory access that is not local to a core in a NUMA architecture or communication costs when spread across several processors. Although the initial motivation for padding was to remove modular arithmetic perhaps it is more relevant that with padding when there are not too many boundary conditions then it is likely that a cache line already contains the boundary cases. That is with a cache line of 64 bytes and word size of 8 bytes then a load at an index of an array will put 8 words(from the index and onwards) in a cache line. Of these 1 was the requested index, 6 may have been within an original unpadded array, but 1 would require filling a new cache line if not already present in a cache line. Eliminating a cache line fill for every boundary condition can be important depending on the size of its compounding effect.

As it may be unclear we stress that a drawback of padding is the irregularity it introduces. Consider an example of looping over a two-dimensional array here shown as nested arrays whose flat repesentation is retrieved by removing the inner bracketing:

$$\langle\langle x_{0,0}, x_{0,1}, \ldots, x_{0,n-1}\rangle, \ldots, \langle x_{n-1,0}, x_{n-1,1}, \ldots, x_{n-1,n-1}\rangle\rangle$$

imagine in our iteration we are performing an indexing of the form were we need to get $x_{i,n+1\,mod\,n}$ and $x_{i,-1\,mod\,n}$ at each index. We pad the array accordingly to simply need the indexing $x_{i,j+1}$ and $x_{i,j-1}$. Now we have eliminated the need for the modular arithmethic, but we need to skip the padding values; we cannot eliminate all branching overhead by simply incrementing an index into the flat representation of the array. Padding changes the iteration space and introduces irregularity that requires bookeeping; profiling is needed on a case by case basis for whether padding makes sense.

8.8 Multithreading and dimension lifting

We now investigate different ways of partitioning the array in a multithreading setting, the tool of choice is to Open Multi-Processing (OpenMP) which enables, among other things, annotating looping with pragmas to indicate splitting of work by iteration space. Although it may be inferred we offer some explanation of the used tools.

```
#pragma omp parallel for schedule(static) \
                         num_threads(THREADS)
for (int i=0;i<K; i++) {...}
```

Here we have annotated a loop such that the iteration space of i will be split among the given number of threads. The master thread will split off iterations blocks, the size of which are statically decided(exact size can be specified as well), to each thread which will run the loop. This is a simple fork/join model of concurrency on the outermost loop. Another directive that can be specified in a parallel for pragma is to specify a collapse.

```
#pragma omp parallel for collapse(2)
for (int i=0;i<K;i++) {
  for (int j=0;j<L;i++) {...}
}
```

This will fuse the iteration space to something like the following:

```
#pragma omp parallel for
for (int f=0;f<K*L;f++) {
  int i = f/L; int j = f % L;
  ...
}
```

In the following experiments we do not change the layout of the array in any way, we simply regard and iterate through it differently. Given an array $\langle s, s, s\rangle$ with its elements layed out in a row-major order consider the notation of $\{i\,j\,k\}$ to denote looping variables that iterate through the array with the rightmost variable being the fastest changing variable.

8.8.1 Dimension lifting on the first dimension

Given that T divides s We may consider the transformation of $\langle s, s, s\rangle \mapsto \langle T, s/T, s, s\rangle$ where we get corresponding iteration variables of $\{p\,i\,j\,k\}$. Here we regard T as the amount of threads allocated and we want to partition $\langle s, s, s\rangle$ such that each thread gets its own contiguous chunk of memory to iterate through. We can achieve this through parallelization of the outermost looping variable p and changing the $gamma(i, j, k)$ function we are using to $gamma'(p, i, j, k) = gamma(p(s/T) + i, j, k)$. A challenge presents itself where operations are applied to i in the original body of our loop. Such operations must propagate into our new $gamma'$ and create modified version. That is we may originally have $gamma(i + 1\, mod\, s_0, j, k)$ which must become $gamma'_{i+1}(p, i, j, k) = gamma(p(s/T) + i + 1\, mod\, s_0, j, k)$ and similarily $gamma'_{i-1}(p, i, j, k) = gamma(p(s/T) + i - 1\, mod\, s_0, j, k)$

Listing 8.11: dimension lifting on first dimension

```
static inline int gammaDL(int p, int i, int j, int k)
{
  return (p*s0/THREADS+i)*s1*s2 + j*s2 + k;X
}
// here MN means mod negative
static inline int gammaDLMN(int p, int i, int j, int k)
```

```
{
  return  mod(p*s0/THREADS+i-1,s0)*s1*s2 + j*s2 + k;
}
// here MP means mod positive
static inline int gammaDLMP(int p, int i, int j, int k)
{
  return  mod(p*s0/THREADS+i+1,s0)*s1*s2 + j*s2 + k;
}
static void MC_DL_ON_THREADS(...) {
  #pragma omp parallel for schedule(static) \
                              num_threads(THREADS)
  for (int p=0;p<THREADS;p++) {
    for (int i=0; i<s0/THREADS; i++) {
      for (int j=0; j<s1; j++) {
        for (int k=0; k<s2; k++) {
          u[gammaDL(p,i,j,k)]=
          u[gammaDL(p,i,j,k)] + c4 * (c3 * (c1 *
          v[gammaDLMN(p,i,j,k)] +
          v[gammaDLMP(p,i,j,k)] +
          v[gammaDL(p,i,mod(j-1,s1),k)] +
          v[gammaDL(p,i,mod(j+1,s1),k)] +
          v[gammaDL(p,i,j,mod(k-1,s2))] +
          v[gammaDL(p,i,j,mod(k+1,s2))]) -
          3 * c2 * u[gammaDL(p,i,j,k)] - c0 *
          ((v[gammaDLMP(p,i,j,k)] -
          v[gammaDLMN(p,i,j,k)]) *
          u0[gammaDL(p,i,j,k)] +
          (v[gammaDL(p,i,mod(j+1,s1),k)] -
          v[gammaDL(p,i,mod(j-1,s1),k)]) *
          u1[gammaDL(p,i,j,k)] +
          (v[gammaDL(p,i,j,mod(k+1,s2))] -
          v[gammaDL(p,i,j,mod(k-1,s2))]) *
          u2[gammaDL(p,i,j,k)]));
        }
      }
    }
  }
}
```

8.8.2 Dimension lifting on the last dimension

Let T divide s and let us consider the transformation $\langle s, s, s\rangle \mapsto \langle s, s, T, s/T\rangle$ and the corresponding looping of $\{i\,j\,p\,k\}$. Much the same considerations must be made for this case as in the case of the dimension lifting on the first dimension. We present an implementation for this case without remarking upon its essentialy equivalent form.

Listing 8.12: dimension lifting on last dimensiton

```
static inline int gammaDLK(int i, int j, int c, int k)
{
  return  i*s1*s2 + j*s2 + c*(s2/s2split) + k;
}
static inline int gammaDLKMN(int i, int j, int c, int k)
{
  return  i*s1*s2 + j*s2 + mod(c*s2/s2split + k-1,s0);
}
static inline int gammaDLKMP(int i, int j, int c, int k)
{
  return  i*s1*s2 + j*s2 + mod(c*s2/s2split + k+1,s0);
}
static void MC_DL_ON_DIMK(...) {
//assuming cache line of 64 bytes,
//try to spread by at least that
#pragma omp parallel for collapse(2) \
schedule(static,8) num_threads(THREADS)
  for (int i=0; i<s0; i++) {
    for (int j=0; j<s1; j++) {
      for (int c=0; c<s2split; c++) {
        for (int k=0; k<s2/s2split; k++) {
          u[gammaDLK(i,j,c,k)] =
          u[gammaDLK(i,j,c,k)] + c4 * (c3 * (c1 *
          v[gammaDLK(mod(i-1,s0),j,c,k)] +
          v[gammaDLK(mod(i+1,s0),j,c,k)] +
          v[gammaDLK(i,mod(j-1,s1),c,k)] +
          v[gammaDLK(i,mod(j+1,s1),c,k)] +
          v[gammaDLKMN(i,j,c,k)] +
          v[gammaDLKMP(i,j,c,k)]) -
          3 * c2 * u[gammaDLK(i,j,c,k)] - c0 *
          ((v[gammaDLK((mod(i+1,s0)),j,c,k)] -
          v[gammaDLK((mod(i-1,s0)),j,c,k)]) *
          u0[gammaDLK(i,j,c,k)] +
          (v[gammaDLK(i,mod(j+1,s1),c,k)] -
          v[gammaDLK(i,mod(j-1,s1),c,k)]) *
          u1[gammaDLK(i,j,c,k)] +
          (v[gammaDLKMP(i,j,c,k)] -
          v[gammaDLKMN(i,j,c,k)]) *
          u2[gammaDLK(i,j,c,k)]));
        }
      }
    }
  }
}
```

8.8.3 Dimension lifting on all dimensions

Let T divide s and let us consider the transformation:

$$\langle s, s, s\rangle \mapsto \langle T, T, T, s/T, s/T, s/T\rangle$$

and the corresponding looping variables of $\{t_i\, t_j\, t_k\, i\, j\, k\}$. Here we let T denote a sub-cube size; we subdivide our $s \times s \times s$ cube into a $T \times T \times T$ grid sub-cubes where each sub-cube contains $s/T \times s/T \times s/T$ elements. We alter the stride of the looping variables t_i, t_j, t_k from 1 to T and consider them to select an offset into a sub-cube ie. $\langle t_i = 3T, t_j = 2T, t_k = 0T\rangle$ denotes sub-cube $\langle 3, 2, 0\rangle$ which has the elements $gamma(i, j, k)$ where $3T \leq i < 3T + T$, $2T \leq j < 2T + T$, $0 \leq k < T$. This technique is often referred to as tiling and our sub-cubes are in fact 3-dimensional tiles.

Listing 8.13: Dimension lifting for tiling

```
static void MC_DL_TILED(...) {
#pragma omp parallel for collapse(3) \
schedule(static) num_threads(THREADS)
  for (int ti=0; ti<s0; ti+=s0/s0tiles) {
    for (int tj=0; tj<s1; tj+=s1/s1tiles) {
      for (int tk=0; tk<s2; tk+=s2/s2tiles) {
        for (int i=ti; i<ti+s0/s0tiles; i++) {
          for (int j=tj; j<tj+s1/s1tiles; j++) {
            for (int k=tk; k<tk+s2/s2tiles; k++) {
              u[gamma(i,j,k)] =
              u[gamma(i,j,k)] + c4 * (c3 * (c1 *
              v[gamma((mod(i-1,s0)),j,k)] +
              v[gamma((mod(i+1,s0)),j,k)] +
              v[gamma(i,(mod(j-1,s1)),k)] +
              v[gamma(i,(mod(j+1,s1)),k)] +
              v[gamma(i,j,(mod(k-1,s2)))] +
              v[gamma(i,j,(mod(k+1,s2)))]) -
              3 * c2 * u[gamma(i,j,k)] - c0 *
              ((v[gamma((mod(i+1,s0)),j,k)] -
              v[gamma((mod(i-1,s0)),j,k)]) *
              u0[gamma(i,j,k)] +
              (v[gamma(i,(mod(j+1,s1)),k)] -
              v[gamma(i,(mod(j-1,s1)),k)]) *
              u1[gamma(i,j,k)] +
              (v[gamma(i,j,(mod(k+1,s2)))] -
              v[gamma(i,j,(mod(k-1,s2)))]) *
              u2[gamma(i,j,k)]));
            }
          }
        }
      }
    }
  }
}
```

8.8.4 Benchmarking dimension lifting

For each of the dimension liftings we have benchmarked them on various machines with an array of shape $\langle 512, 512, 512 \rangle$, which with a double size of 8 bytes yields right above 1GB of data to process. Furthermore the amount of threads used is set at 32. Many of the machines we have run these dimension liftings on have more cores than 32 threads. Granted we have not taken steps to ensure a one to one relation between a thread and a core or that memory allocations are ensured to be local to a core with a thread's workload; such considerations is a topic for further examination and may very well turn out to be crucial for further improvements.

For the following table of results the measurements are in seconds. S is short for a single threaded implementation corresponding to one found in listing 8.3, the rest are multithreaded. DL(i) is short for a dimension lifted implementation on the first dimension corresponding to one found in listing 8.11, DL(k) is short for a dimension lifted implementation on the last dimension corresponding to one found in listing 8.12 and DL(tiled) is short for a dimension lifted implementation corresponding to one found in listing 8.13. For DL(i) we have opted to split the dimension by the number of threads(32). For DL(k) we have opted to split the dimension by 8. For DL(tiled) we have opted to split all dimensions by 8 (making each tile of size $64 \times 64 \times 64$).

Table 8.4: gcc 8.2.0 -O3 -march=native -mtune=native

Method	x86 Intel 1	x86 Intel 2	x86 Amd	AArch Arm
S	236.2	174.6	299.7	659.1
DL(i)	68.9	82.8	58.5	110.2
DL(k)	61.7	82.9	65.4	89.7
DL(tiled)	89.5	145.2	77.5	116.2

Table 8.5: Models

Machine	Model	Cores	Ghz max
x86 Intel 1	Xeon(R) Gold 6130	16×2	2.1
x86 Intel 2	Xeon(R) Silver 4112	4	2.6
x86 Amd	EPYC 7601	32×2	2.6
AArch Arm	ThunderX2 99xx	32×2	2.5

Table 8.6: Cache sizes

Machine	L1	L2	L3
x86 Intel 1	32K	1024K	22528K
x86 Intel 2	32K	1024K	8448K
x86 Amd	32K	512K	8192K
AArch Arm	32K	256K	32768K

8.9 Summary

We have examined the effects of using MOA to reformulate a piece of code heavily reliant on array operations. To take a high level look at the improvement made we can summarize it as going from a several pass style of computation to fusing operations and doing computations in one pass. In a sense we may regard this as going from an expression of the form $g(f(h(a)))$ to constructing a composition and applying it $(g \circ f \circ h)(x)$. From a purely mathemathical sense there is nothing interesting there, but from a programming sense such compositions can have dramatic effects. Where before, given that x is an array and the functions affect its elements, we went from three passes through a possibly big chunk of memory to one pass. This is one kind of optimization that MOA affords us; it elevates our thinking to larger concerns where the fusion of operations becomes a more easily expressible optimization.

When we have described our operations and what we want to compute we may explore considerations of iteration, data layout and specific considerations for the hardware the computation is to run on. This separation of concerns is as mentioned before the DNF in which we describe arrays by their shape and a function or some assignment of values at every index, how values change under operations and the ONF which takes the DNF description and relates an array to an arrangement in memory. Here we have considered certain ONF transformations in an effort to trigger further optimizations or to paralellize the code. It is likely that the removal of looping may trigger better vectorization of array code, which may account for the significant improvement in one case. The exploration of this optimization space is one of great tedium where it is easy to make mistakes. This suggests we should further formalize and semi-automate this particular exploration space; axiom directed code generation could be useful tool in this area. Exploring the optimizations that do occur is somewhat of an arcane art of reading through generated assembly which is generally a special case for differing architectures and processors.

We argue that the optimization exploration space should be expressible in the language itself with some kind of axiom system on statements and expressions. Take for instance nested loops that are admissible to be unrolled at boundary cases of some operation: Here we have a contrived and simple

```
for i = 0 to 100 {
  for j = 0 to 10 {
    a[mod(j+1,10)]=...;
    a[mod(j-1,10)]=...;
  }
}
```

```
for i = 0 to 100 {
  a[1]=...;
  a[9]=...;
  for j = 1 to 9 {
    a[mod(j+1,10)]=...;
    a[mod(j-1,10)]=...;
  }
  a[1]=...;
  a[9]=...;
}
```

example, but one could imagine much more complicated nesting and expressions. We have two choices here. We can keep this form of expressing the problem and write a transformation system that gradually moves out statements until we reach the righthand side from the lefthand side. Alternatively we could consider the need for such exploration as a sign that we are missing something in the formalism that was used to generate the code; that it should have been removed at a higher level.

CHAPTER 9

Conclusion

We conclude our exploration of transformations for array programming by retracing our steps. We have explored some of the transformation systems in practical use in programming languages as well as up-and-coming languages and frameworks, in particular for array programming. We have explored the array concept in its many variations and have devoted a significant amount of the thesis to examining one, namely MOA. Is there one all-encompassing array formalism? Likely not; although we may shrug and say, in MOA, the indices fitting the shape of $\langle 2,2,2\rangle$ are isomorphic to the same indices but negative, an actual user may refuse to accept that they must perform this translation and use something like Petalisp instead. While evaluating MOA we(more than the author), noted that the indices fitting a shape of $\langle 5,1,5\rangle$,$\langle 1,5,5\rangle$ or $\langle 1,5,5\rangle$ are all isomorphic to the ones fitting $\langle 5,5\rangle$ and pondered a notion of compression such that such arrays could be used as a $\langle 5,5\rangle$ shaped array. The conveniences offered by an array formalism should not be discounted in evaluating its merits or likelihood of being put to use by the masses. As such we believe work remains in gathering variations of the array concept and investigating whether MOA or extensions of MOA can acommodate them.

Concepts in array formalisms recur and the formalisms that can capture a great deal of our real world problems are worthile objects of study. We believe we have made a case for MOA with the results achieved in putting it to use to describe a select problem, generating efficient code by hand from that description and showing we can tune and adapt the code by thinking in terms of the constructs the formalism provides.

We believe we have shown that MOA is rife with identities and especially admissible for a transformation based approach for deriving array code and is also powerful as a vehicle of thought contrasted with the reasoning one can achieve by thinking in terms of loops and single pass updates of arrays. All too often without abstraction we, at the very least *this* author, end up lost in a fog where progress can be made but progress is greatly hampered by all details being present at the same time instead of at different tiers of an unravelable abstraction such as is the case with the tiers of MOA in the form of the DNF and the ONF.

We believe that to effectively put formalisms such as MOA to use will require a transformation based approach to programming libraries seeing as the distance from description to code can be far, although every step along the way can be described. We have shown work in this direction with Petalisp and Lift and whereas rewriting and metaprogramming has long been a technique employed in Haskell(rewrite rules) and C++(expression templates) for writing libraries with some ability to self-optimize according to codified identities, these facilities are often difficult to use. They are often an afterthought or a technique without any support from the language.

Some make monolithic compilers for each formalism to generate code. This we argue is the wrong approach as it black boxes too large an aspect of going from(e.g.) algebraic specification to code. Rather the formalisms should be collected in the form of domain knowledge, such as can be can be gathered in a language like Magnolia with its axiomatic specification facility, and a programmable rewrite system should be made for code generation from such domain knowledge. As well as being a stage of code generation, the domain knowledge should also be a part of the stage we would like to call axiom directed exploratory programming. When prompted a compiler should list admissible transformations at a certain expression for an axiom directed kind of programming. We hope to see such directions further explored.

Appendices

APPENDIX A

C++ examples

Listing A.1: Minimal ET example of a vector type wrapping fixed size integer arrays

```
#include <iostream>
#include <stdlib.h>

template <typename A, typename B>
class Sum
{
  public:
  explicit Sum(const A &a, const B &b)
  : a_(a)
  , b_(b) {}
  std::size_t size() const {
    return a_.size();
  }
  int operator[](std::size_t i) const {
    return a_[i]+b_[i];
  }
  private:
  const A &a_;
  const B &b_;
};
```

```
template <typename A, typename B>
Sum <A,B> operator +(const A &a, const B &b)
{
  return Sum <A,B>(a, b) ;
}

template <std::size_t N>
class Vector
{
  public:
  Vector(int (&v)[N]) : v_(v) {}
  int operator[](std::size_t i) const {
    return v_[i];
  }
  std::size_t size() const {return N;}
  template <typename A>
  Vector & operator=(const A &expr) {
    for (std::size_t i=0; i<expr.size(); ++i )
      v_[i] = expr[i];
    return *this ;
  }
  private:
  int *v_;
};
int main() {
  int a[] = {1,2,3,4,5};
  int b[] = {6,7,8,9,10};
  int c[] = {11,12,13,14,15};
  int d[] = {0,0,0,0,0};

  Vector va(a);
  Vector vb(b);
  Vector vc(c);
  Vector vd(d);

  vd = va+vb+vc;

  for(auto e : d)
    std::cout << e << '\n';
  return 0;
}
```

APPENDIX B

C examples

Listing B.1: benchmark of various memory access patterns

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX 10000
#define BLOCK 10

int A[MAX][MAX];
int B[MAX][MAX];
int C[MAX][MAX];

void fill(int R[MAX][MAX])
{
  for (int i=0; i< MAX; i++) {
    for (int j=0; j< MAX; j++) {
      R[i][j] = i+j;
    }
  }
}

void copy(int S[MAX][MAX],int D[MAX][MAX])
```

```
{
  for (int i=0; i< MAX; i++) {
    for (int j=0; j< MAX; j++) {
      D[i][j]=S[i][j];
    }
  }
}

int check(int X[MAX][MAX],int Y[MAX][MAX], const char *str)
{
  for (int i=0; i< MAX; i++) {
    for (int j=0; j< MAX; j++) {
      if (X[i][j] != Y[i][j]) {
        printf("%s\n",str);
        return 1;
      }
    }
  }
  return 0;
}


void rowmajorA(int R[MAX][MAX])
{
  for (int i=0; i< MAX; i++) {
    for (int j=0; j< MAX; j++) {
      R[i][j] = R[i][j] + B[j][i];
    }
  }
}

void columnmajorA(int R[MAX][MAX])
{
  for (int i=0; i< MAX; i++) {
    for (int j=0; j< MAX; j++) {
      R[j][i] = R[j][i] + B[i][j];
    }
  }
}

void tilingA(int R[MAX][MAX])
{
  for (int i=0; i< MAX; i+=BLOCK) {
    for (int j=0; j< MAX; j+=BLOCK) {
      for (int ii=i; ii<i+BLOCK; ii++) {
        for (int jj=j; jj<j+BLOCK; jj++) {
          R[ii][jj] = R[ii][jj] + B[jj][ii];
        }
      }
    }
  }
}

void rowmajorB(int R[MAX][MAX])
{
  for (int i=0; i< MAX; i++) {
    for (int j=0; j< MAX; j++) {
      R[i][j] = 2*R[i][j];
    }
  }
}

void columnmajorB(int R[MAX][MAX])
{
  for (int i=0; i< MAX; i++) {
    for (int j=0; j< MAX; j++) {
      R[j][i] = 2*R[j][i];
    }
  }
}

void tilingB(int R[MAX][MAX])
{
  for (int i=0; i< MAX; i+=BLOCK) {
    for (int j=0; j< MAX; j+=BLOCK) {
      for (int ii=i; ii<i+BLOCK; ii++) {
        for (int jj=j; jj<j+BLOCK; jj++) {
          R[ii][jj] = 2*R[ii][jj];
        }
      }
    }
  }
}

int main()
{
```

```
int error=0;
clock_t begin;
clock_t end;
double tspent;

fill(A);
fill(B);

rowmajorA(A);
copy(A,C);
fill(A);
columnmajorA(A);
error += check(A,C,"rowmajorA");

fill(A);
tilingA(A);
copy(A,C);
fill(A);
rowmajorA(A);
error += check(A,C,"tilingA");

fill(A);
rowmajorB(A);
copy(A,C);
fill(A);
columnmajorB(A);
error += check(A,C,"rowmajorB");

fill(A);
tilingB(A);
copy(A,C);
fill(A);
rowmajorB(A);
error += check(A,C,"tilingB");

printf("Correctness check: ");
if(error == 0) {
  printf("Passed!\n");
} else {
  printf("Failed!\n");
  exit(1);
}
```

```
printf("Computation: A = A+B\n");
fill(A);
printf("Row major\n");
begin = clock();
rowmajorA(A);
end = clock();
tspent = (double)(end - begin) / CLOCKS_PER_SEC;
printf("time: %lf\n",tspent);

fill(A);
printf("Column major\n");
begin = clock();
columnmajorA(A);
end = clock();
tspent = (double)(end - begin) / CLOCKS_PER_SEC;
printf("time: %lf\n",tspent);

fill(A);
printf("Tiling\n");
begin = clock();
tilingA(A);
end = clock();
tspent = (double)(end - begin) / CLOCKS_PER_SEC;
printf("time: %lf\n",tspent);

printf("Computation: A = 2*A\n");
fill(A);
printf("Row major\n");
begin = clock();
rowmajorB(A);
end = clock();
tspent = (double)(end - begin) / CLOCKS_PER_SEC;
printf("time: %lf\n",tspent);

fill(A);
printf("Column major\n");
begin = clock();
columnmajorB(A);
end = clock();
tspent = (double)(end - begin) / CLOCKS_PER_SEC;
printf("time: %lf\n",tspent);

fill(A);
```

```
    printf("Tiling\n");
    begin = clock();
    tilingB(A);
    end = clock();
    tspent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("time: %lf\n",tspent);

    return 0;
}
```

APPENDIX C

The affine-linear mapping

Consider vector spaces over integers, that is let $s \in \mathbb{Z},\ u, v, t \in \mathbb{Z}^n$ and $f, g, T : \mathbb{Z}^n \to \mathbb{Z}^n$.

The mapping f is said to be linear if it satisfies:

$$\begin{aligned} f(u+v) &= f(u) + f(v) && \text{additivity} \\ f(cu) &= cf(u) && \text{homogeneity} \end{aligned}$$

The mapping g is said to be affine if:

$$g(u) = u + v \qquad \text{affinity}$$

That is one can regard g as translating or moving a point u to $u + v$.

An affine-linear transformation is the composition of an affine and a linear transformation. That is, let $g(x) = x + t$ be an affine transformation and f be a linear transformation, then then an affine-linear transformation can be defined as

$$T(x) = g(f(x)) = f(x) + t$$

Affine-linear transformations have some well-known properties:

- Points on lines are mapped to points on lines(lines to lines)
- Parallel lines remain parallel
- Ratios between points are preserved (e.g. the midpoint on a line is mapped to the midpoint of the line it is mapped to)

APPENDIX D

Subranges

Consider a sequence of bounds for non-negative integers $k_i \in \{z \geq 0 \mid z \in \mathbb{Z}\}$ such that $k_0 = 0 < k_1 < \cdots < k_n < k_{n+1}$ and an integer $0 \leq k \leq k_{n+1}$.

We wish to determine in which subrange k fits, meaning if $k_{i-1} < k \geq k_i$ then it fits in the $i-1$th subrange. For $k_0 = 0 < k_1 < \cdots < k_n < k_{n+1}$ there are $n+1$ subranges $\{0, 1, \ldots, n\}$.

Consider k fitting a subrange, then:

$$\begin{aligned} k \geq k_i &\Rightarrow k - k_i - 1 < 0 \\ k_i < k &\Rightarrow 0 < k - k_i \end{aligned}$$

and then define

$$cmp(x, k) = clamp(x - k - 1)$$

$$clamp(x) = \begin{cases} 1 & \text{when } x < 0 \\ 0 & \text{otherwise} \end{cases}$$

The subrange i fits in can then be computed by counting how many subranges it doesn't fit into:

$$subrange(k, k_1, k_2, \ldots, k_{n-1}) = cmp(k, k_1) + cmp(k, k_2) + \cdots + cmp(k, k_{n-1})$$

APPENDIX E

An unconventional modulo

A peculiar convention in Mullin's work in MOA is to define, for non-negative integers, that $a\,mod\,0 = a$. Certainly it cannot be interpreted as the remainder of integer division by 0 which as we know is undefined. However we may yet interpret it meaningfully. The first and obvious interpretation is that $a\,mod\,b = r = a - qb$ where $a = qb + r$ for non-negative integers, thus $a\,mod\,0 = a = a - q \cdot 0$. This is the interpretation Mullin provides..

We can also make sense of it by considering it to produce the remainder of slicing division for non-negative integers rather than the remainder of regular division. Slicing division [22] can be defined as $a/_s b = a/((b -_s 1) + 1)$ where $/$ is integer division and $a -_s b = a - min(a, b)$ is sometimes called a saturating subtraction(from saturation arithmetic). Slicing division is a way of making division total for the non-negative integers. Under this interpretation we have $a/_s 0 = a/((0 -_s 1) + 1) = a/1 = a$ and as such a is the remainder of $a/_s 0$. In other cases slicing division is equivalent to regular integer division.

APPENDIX F

An unpublished paper on padding and dimension lifting

As a result of the COVID-19 outbreak the planned "7th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming" was first postponed and then cancelled. We had prepared and submitted the following paper, which will likely find another venue. This is again a paper with many authors and in the work therein the author of this thesis contributed benchmarking results as well as aided in checking of definitions.

Padding Stencil Computations in the Mathematics of Arrays Formalism

Benjamin Chetioui
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
benjamin.chetioui@uib.no

Ole Abusdal
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
ole.abusdal@student.uib.no

Magne Haveraaen
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
https://www.ii.uib.no/~magne/

Jaakko Järvi
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
jaakko.jarvi@uib.no

Lenore Mullin
College of Engineering and Applied Sciences
University at Albany, SUNY
Albany, NY, USA
lmullin@albany.edu

Abstract

Multi-dimensional array manipulation constitutes a core component of numerous numerical methods, e.g. finite difference solvers of Partial Differential Equations (PDEs). The efficiency of such computations are tightly connected to traversing array data in a hardware-friendly way.

The Mathematics of Arrays (MoA) allow us to reason about array computations at a high level, and enable systematic transformations of array-based programs. In a previous paper we investigated stencil computations, and proved that these can be reduced to Denotational Normal Form (DNF).

Here we demonstrate another set of transformations on array expressions: the Operational Normal Forms (ONFs) which allow us to adapt array computations to hardware characteristics. The ONF transformations start from the DNF array format. Alongside the ONF transformations, we extend MoA with the rewriting rules pertaining to padding. These new rules allow both a simplification of array indexing and a systematic approach to introducing halos (ghost cells) to PDE solvers, thus reducing the cost of communication between different computation loci.

CCS Concepts • **Software and its engineering → Software design engineering**;

Keywords Mathematics of Arrays, Finite Difference Methods, PDE Solvers, High-Performance Computing

ACM Reference Format:
Benjamin Chetioui, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Lenore Mullin. 2020. Padding Stencil Computations in the Mathematics of Arrays Formalism. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '20), June ??, 2020, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In the past few decades, a large variety of high-performance computing (HPC) architectures has appeared. On the path towards exascale computing, the emergence of an equally varying set of architectures is expected. Software for HPC therefore needs to be highly adaptable. This includes adjusting to, among others, different memory hierarchies and changing intra- and interprocess communication hardware. This is especially important for large scale computations, e.g. Partial Differential Equation (PDE) solvers. One important class of PDE solvers are the Finite Difference Methods (FDM). FDM solvers are stencil-based array computations.

Our previous work [3] established means of transforming stencil-based array code to Denotational Normal Form (DNF). These are irreducible expressions in the language of the Mathematics of Arrays (MoA). Given knowledge of the target parallel distribution and memory layout, another set of transformations takes a DNF expression to the architecture-aware Operational Normal Form (ONF), which we describe in Section 4. ONF transformations include the dimension lifting operation that reshapes an array so as to split it conveniently over different computation loci (whether they be threads, cores, or even systems).

The contribution of this paper is a formalisation of MoA's ONF along with a new extension of the MoA operations to

	S	MDL	MDLSL	MDLTM
CPU 1	225.74	70.96	66.66	61.81
CPU 2	299.42	59.16	68.14	68.70
CPU 3	172.71	85.97	85.59	117.11
CPU 4	660.53	85.06	72.80	77.86

Table 1. Execution time (in seconds) of a PDE solver implementation depending on hardware and dimension lifting parameters. The gray background marks the fastest version(s) of the solver for each row. The labels are as follows: S: Single core (no Dimension Lifting); MDL: Multicore (Dimension Lifting on 0-th dimension); MDLSL: Multicore (Dimension Lifting on $n-2$-th dimension); MDLTM: Multicore (Dimension Lifting on 0-th dimension using tiled memory); CPU 1: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz; CPU 2: AMD EPYC 7601 32-Core; CPU 2: Intel(R) Xeon(R) Silver 4112 CPU @ 2.60GHz; and CPU 2: ThunderX2 99xx.

deal with padding of data. This gives a framework for transforming regular array stencil code to distributed code with *halo zones* (also referred to as *ghost cells* in the literature).

The paper is organised as follows; next is a motivation section, then a discussion of related work. Section 4 subsequently covers the required prerequisites in MoA and our previous work at the DNF layer. Section 5 explains dimension lifting, padding and data layout, giving both their formal definitions and select examples. We then have a short look at some experiments and conclude in Section 7.

2 Motivation

To motivate our work, we ran the PDE solver presented in our previous work [3] on a set of experimental architectures. We also implemented some of the ONF transformations on the code. Table 1 shows a matrix where each column corresponds to a different version of the solver and each row to different hardware. It is clear from the table that different architectures benefit from different transformations. While on CPU 1 the *dimension lifting on* 0*-th axis and tiled memory* approach performs best, it is clearly inefficient compared to the other dimension lifting-based scenarios on CPU 3. The seemingly inconsistent variations in performance, and the sheer number of different memory layouts, outline the need for a vehicle for easy exploration of different memory layouts, so that programmers can get close-to-optimal performance with as little effort as possible for any given architecture. In the following, we demonstrate that MoA, when extended with operations for padding, provides us with enough expressivity to accomplish just that.

3 Related

Ken Iverson introduced whole-array operations in the APL programming language [6]. Building on further explorations by Abrams [1], Mullin created the Mathematics of Arrays formalism [8] in order to address various shortcomings of the universal algebra underlying APL (most notably the lack of a calculus for indexing). MoA is intended to serve as a foundation for exploring and optimizing array/tensor operations. Mullin further explored MoA through case studies of scientific algorithms, including QR Decomposition [10], and Fast Fourier Transforms (FFTs) [5]. The latter paper introduced the dimension lifting operation that is crucial to our work.

Burrows et al. identified an array API for FDM solvers of PDEs [2]. We previously explored the fragment of MoA corresponding to this API and concluded that stencil computations could systematically be reduced to a normal form at the DNF level [3]. Other researchers have also looked into optimizing stencil computations, such as Hagedorn et al. [4], who augmented LIFT with the same operations we explore here for that purpose.

4 Background and Notation

4.1 Mathematics of Arrays

We give a short introduction, following roughly the presentation in our earlier paper [3], to the MoA algebra [8, 9] for representing and describing operations on arrays. MoA defines a set of rules, the ψ-calculus, for manipulating array shapes and expressions. By systematically applying a set of terminating rewriting rules, we can transform an array expression to a single array with standard layout and operations on the array elements, the Denotational Normal Form (DNF). DNF can further be transformed into a corresponding Operational Normal Form (ONF), which represents the array access patterns in terms of *start*, *stride* and *length*. Together with *dimension lifting*, this lets us reorganize the memory layout and data access patterns, and to thus take into account distribution of data and memory hierarchies, data locality, etc., a flexiblity needed for current and future hardware architectures.

4.1.1 Notation and operations

The *dimension* of an array A is denoted $\dim(A)$. We define the *shape* of an n-dimensional array A as a list of sizes $\langle s_0 \ \dots \ s_{n-1}\rangle$. The *size* of an array is the number of elements it contains, i.e. $\text{size}(A) = s_0 \times \cdots \times s_{n-1}$. An empty array is an array with size 0, i.e. an array of which at least one of the shape components is 0. For example, a 6 by 8 matrix M is a 2-dimensional array with shape $\langle 6\ 8\rangle$ and size 48. The content of an array A at a given index is defined by an indexing function relevant to the abstraction layer we are considering.

We summarize the core operations of MoA used in this paper below.

Relevant operations at the DNF level At the DNF level, we make use of the following core operations:

- the shape function ρ returns the shape of an array, e.g. $\rho(M) = \langle 6\ 8 \rangle$;
- the indexing function ψ takes an index, that is to say a list $\langle i_0\ \dots\ i_k \rangle$ where $k < \dim(A)$, and an array A, such that $0 \le i_j < s_j$ for $j \in \{0, \dots, k\}$ where $\rho(A) = \langle s_0\ \dots\ s_{n-1} \rangle$, and returns the subarray of A at this position. For all arrays A, $\langle\rangle\ \psi\ A = A$. In our example, $N = \langle 3 \rangle\ \psi\ M$ is the subarray of M at position 3 with the shape $\rho(N) = \langle 8 \rangle$. When the length of the index is the same as the dimension of A, we call the index a *total index*. Using ψ, we can thus describe the content of an array for each index;
- we may change the shape of an array, *reshape* it, as long as the size of said array remains the same. Let $s' = \langle s'_0\ \dots\ s'_k \rangle$ such that $s'_0 \times \dots \times s'_k = \text{size}(A)$, then $A' = \text{reshape}(s', A)$ gives the array A' with the same size and elements as A, but such that $\rho(A') = s'$;
- The binary function take, given an n-dimensional array A with $\rho(A) = \langle s_0\ \dots\ s_{n-1} \rangle$ and a positive (respectively negative) integer t, $0 \le |t| \le s_0$, returns the slice of A that contains the first (respectively last) $|t|$ subarrays of A. Thus, we have that: $\rho(\text{take}(t, A)) = \langle |t|\ s_1\ \dots\ s_{n-1} \rangle$ and for $i \in \{0, \dots |t| - 1\}$,

$$\langle i \rangle\ \psi\ \text{take}(t, A) = \begin{cases} \langle i \rangle\ \psi\ A & t \ge 0 \\ \langle s_0 + t + i \rangle\ \psi\ A & \text{otherwise;} \end{cases}$$

- The binary function drop, given an n-dimensional array A with $\rho(A) = \langle s_0\ \dots\ s_{n-1} \rangle$ and a positive (respectively negative) integer t, $0 \le |t| \le s_0$, returns the slice of A stripped of its first (respectively last) $|t|$ subarrays. Thus, we have that: $\rho(\text{drop}(t, A)) = \langle s_0 - |t|\ s_1\ \dots\ s_{n-1} \rangle$ and for $i \in \{0, \dots, s_0 - |t| - 1\}$,

$$\langle i \rangle\ \psi\ \text{drop}(t, A') = \begin{cases} \langle i + t \rangle\ \psi\ A & t \ge 0 \\ \langle i \rangle\ \psi\ A & \text{otherwise;} \end{cases}$$

- The function cat concatenates two arrays A, B with compatible shapes, i.e. two arrays whose shapes differ only in their first component. Let A and B be two arrays with shapes $\rho(A) = \langle s_0\ s_1\ \dots\ s_{n-1} \rangle$ and $\rho(B) = \langle s'_0\ s_1\ \dots\ s_{n-1} \rangle$, respectively. Concatenation of A and B is defined by $\rho(\text{cat}(A, B)) = \langle (s_0 + s'_0)\ s_1\ \dots\ s_{n-1} \rangle$ and

$$\langle i \rangle\ \psi\ \text{cat}(A, B) = \begin{cases} \langle i \rangle\ \psi\ A & i < s_0 \\ \langle i - s_0 \rangle\ \psi\ B & \text{otherwise.} \end{cases}$$

Since the empty array with shape $\langle 0\ s_1\ \dots\ s_{n-1} \rangle$ is compatible with the arrays of shape $\langle s_0\ s_1\ \dots\ s_{n-1} \rangle$, it is easy to see that this gives us a monoid structure with concatenate as the binary operation and the empty array with compatible shape as neutral element.
For an array A with $\rho(A) = \langle s_0\ \dots\ s_{n-1} \rangle$, the relation $\text{cat}(\text{take}(|k|, A), \text{drop}(|k|, A)) = A$ holds for $|k| \le s_0$.

We can also define a relaxed version of concatenation which adequately reshapes one of its parameters B in 2 situations:

1. if B is an empty array, then B can automatically be reshaped to the empty array with compatible shape with regards to the other parameter;
2. if B has shape $\rho(B) = \langle s_1\ \dots\ s_{n-1} \rangle$ and $\langle 1\ s_1\ \dots\ s_{n-1} \rangle$ is a shape compatible with regards to the other parameter, then B can automatically be reshaped to $\langle s_1\ \dots\ s_{n-1} \rangle$.

In the following, we overload cat also with the relaxed version;

- The rotation function θ rotates an array. We write $p\ \theta_i\ A$ to represent the rotation of A by offset p along axis i. Since a rotation is just some kind of a permutation, $\rho(p\ \theta_i\ A) = \rho(A)$. Informally, given s_i (the shape component of A for axis i), θ_i is defined as follows:

$$x_i\ \psi\ (p\ \theta_i\ A) = \begin{cases} \text{cat}(\text{drop}(|p|, B), \text{take}(|p|, B)) & p < 0 \\ \text{cat}(\text{drop}(s_i - p, B), \text{take}(s_i - p, B)) & p \ge 0 \end{cases}$$

where $B = x_i\ \psi\ A$ and $x_i = \langle k_0\ \dots\ k_{i-1} \rangle$ is a valid partial index in A.

Relevant operations at the ONF level To work at the ONF level, we add the following operations:

- The Ravel operator rav is the *flattening* operation (borrowed from APL). It takes a multidimensional array and reshapes it into a unidimensional array (i.e. a vector). We use rav to transport an array into its corresponding representation in memory;
- The indexing function γ, given an index and the shape of an array, returns the corresponding index in the flattened representation of the array. γ requires knowledge of the specific memory layout of the array, which is why it can only be used at the ONF level. In the same way that ψ is used to determine the content of A at each index at the DNF level, γ is used to determine the content of $\text{rav}(A)$ at each index at the ONF level;
- The range (or *stride*) operator ι, given a positive integer n, returns a 1-dimensional array such that $\langle i \rangle\ \psi\ \iota(n) = i$ for $0 \le i < n$.

Formal definitions for the operations described above can be found in Mullin's original work [8]. Section 5 defines new operations and extends the notation of MoA to express the transformations required to work with padded data, both with and without dimension lifting.

4.2 PDE Solver Framework

We remind the reader of the PDE solver framework that we use as an example; we follow roughly our earlier description [3]. Figure 1 illustrates the solver's design structured by layers. The first abstraction layer defines the problem through the domain's concepts: PDEs are expressed using collective and continuous operations to relate the physical

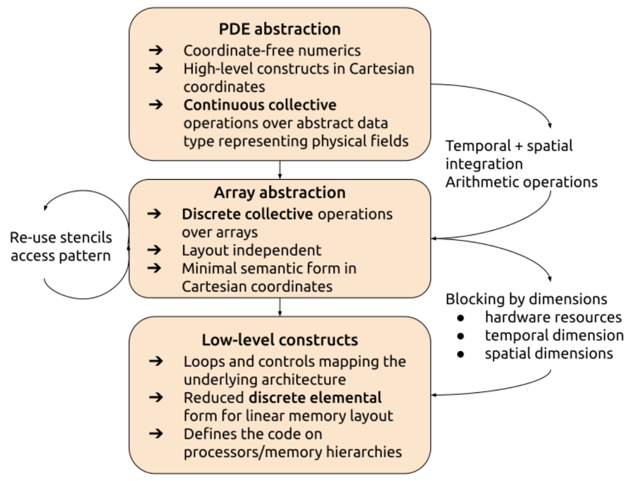

PDE abstraction
➔ Coordinate-free numerics
➔ High-level constructs in Cartesian coordinates
➔ Continuous collective operations over abstract data type representing physical fields
Temporal + spatial integration
Arithmetic operations
Array abstraction
➔ Discrete collective operations over arrays
➔ Layout independent
➔ Minimal semantic form in Cartesian coordinates
Re-use stencils access pattern
Blocking by dimensions
• hardware resources
• temporal dimension
• spatial dimensions
Low-level constructs
➔ Loops and controls mapping the underlying architecture
➔ Reduced discrete elemental form for linear memory layout
➔ Defines the code on processors/memory hierarchies

Figure 1. Layer abstraction design; detailed environment designed for a PDE solver (figure taken and adapted from our previous work [3]).

fields involved. Through the functions encapsulating the numerical methods, the high-level continuous abstraction is mapped to a discrete array-based layer. The MoA algebra defines the problem through collective array operations in a layout independent manner. At this point, array manipulation functions and operations may be defined in the MoA formalism and reduced according to the ψ-reduction process. From the user's array-abstracted expression we obtain an equivalent optimal and minimal semantic form.

Our prior work [3] developed the rewriting system required for obtaining this DNF; the present paper focuses on the third and lowest-level layer of the figure. On this layer, the indexing algebra of the ψ-calculus relates the monolithic operations to elemental operations, defining the code on processors and memory hierarchies through loops and controls. We use the ψ-correspondence theorem to transform the high-level abstracted array expressions into operational expressions, i.e. from a form involving Cartesian coordinates into one involving linearly arranged memory accesses.

5 Memory Layout in the MoA

In the rest of the paper, we assume a row-major memory layout. Let A be an array with shape $\langle 6\ 8\rangle$. Consider the DNF expression $\langle i\ j\rangle\ \psi\ ((1\ \theta_0\ A) + (-1\ \theta_0\ A))$. Assuming a one-core processor, we apply the ψ-correspondence theorem to transform it to the corresponding ONF expression:

$$\begin{aligned}
&\forall i\ \ s.t.\ \ 0 \le i < 6\\
&\qquad \langle i\ j\rangle\ \psi\ ((1\ \theta_0\ A) + (-1\ \theta_0\ A))\\
&\quad \equiv (\mathrm{rav}\, A)[\gamma(\langle ((i+1) \bmod 6\rangle; \langle 6\rangle) \times 8 + \iota 8] +\\
&\qquad (\mathrm{rav}\, A)[\gamma(\langle ((i-1) \bmod 6\rangle; \langle 6\rangle) \times 8 + \iota 8].
\end{aligned}$$

We follow up by applying γ:

$$\begin{aligned}
&\equiv (\mathrm{rav}\, A)[((i+1) \bmod 6) \times 8 + \iota 8] +\\
&\quad (\mathrm{rav}\, A)[((i-1) \bmod 6) \times 8 + \iota 8].
\end{aligned}$$

Applying rav and turning ι into a loop we get the following generic program:

$$\begin{aligned}
&\forall j\ \ s.t.\ \ 0 \le j < 8\\
&\quad A[((i+1) \bmod 6) \times 8 + j] +\\
&\quad A[((i-1) \bmod 6) \times 8 + j].
\end{aligned}$$

Optimizations would still be possible here, but they would require a cost analysis of the operations used in the program (*addition*, *multiplication* and *modulo*). In this paper, we do not concern ourselves with such optimizations, but instead explore the effects of dimension lifting.

We call *dimension lifting* the action of "splitting" a given axis of the shape of an array into two or more dimensions. This allows us to establish a correspondence between the shape of the array and the underlying hardware architecture, which may help achieve better performance. We explain dimension lifting first with an example, then a formal definition follows in Section 5.1.

Assume we have two one-core processors. We decide to apply dimension lifting on axis 1 of A, to create A' whose shape is $\langle 6\ 2\ \frac{8}{2}\rangle = \langle 6\ 2\ 4\rangle$, in which axis 1 now corresponds to the number of cores. We get the following:

$$\begin{aligned}
&\forall i, j\ \ s.t.\ \ 0 \le i < 6,\ \ 0 \le j < 2\\
&\qquad \langle i\ j\rangle\ \psi\ ((1\ \theta_0\ A') + (-1\ \theta_0\ A'))\\
&\quad \equiv (\mathrm{rav}\, A')[\gamma(\langle ((i+1) \bmod 6)\ j\rangle; \langle 6\ 2\rangle) \times 4 + \iota 4] +\\
&\qquad (\mathrm{rav}\, A')[\gamma(\langle ((i-1) \bmod 6)\ j\rangle; \langle 6\ 2\rangle) \times 4 + \iota 4]\\
&\quad \equiv (\mathrm{rav}\, A')[(((i+1) \bmod 6) \times 2 + j) \times 4 + \iota 4] +\\
&\qquad (\mathrm{rav}\, A')[(((i-1) \bmod 6) \times 2 + j) \times 4 + \iota 4].
\end{aligned}$$

This reduces to the following generic program:

$$\begin{aligned}
&\forall k\ \ s.t.\ \ 0 \le k < 4\\
&\quad A'[((i+1) \bmod 6) \times 4 \times 2 + j \times 4 + k] +\\
&\quad A'[((i-1) \bmod 6) \times 4 \times 2 + j \times 4 + k].
\end{aligned}$$

The two programs above are equivalent, but use a different looping structure — they are adapted to two different hardware architectures.

While dimension lifting can be carried out across any axis (or axes, dimension lifting can be performed on several axes simultaneously), this choice should be guided the memory hierarchy and by the operations involved in the expression. For example, the rotations above are applied on axis 0; if we were to perform dimension lifting on this axis, we would not be able to perfectly split the memory between the two processors. This case would require either inter-process communication or some kind of data redundancy (discussed in Section 5.1).

5.1 Padding

In high-performance computing, it is often desirable to transform code to use less costly operations in lieu of more expensive ones. For example, even on today's hardware, division and modulo operations remain considerably more expensive than multiplication and addition [7].

The example above uses a modulo operation on the index. Below we describe a *padding* operation on DNF expressions that transforms the shape and content of an array to introduce data redundancy. We then show two examples where this data redundancy eliminates expensive operations that are applied to edge cases during conversion to ONF. Example 1 shows how the padding operations defined in Section 5.1.1 eliminate *modulo* operations in a single core setting. The padding operations are then generalized to work in a setting involving dimension lifting in Section 5.1.2; Example 2 puts them to work to reduce the need for inter-process communication in a distributed setting.

Increasing data redundancy is a way to get rid of expensive operations, but it may also carry further benefits: carefully chosen padding may increase data locality, e.g. by ensuring that relevant slices of an array do not span over more cache lines than necessary.

5.1.1 One Core with Constant Memory Access Cost

We informally define *padding an array* as the action of prepending or appending data to it. For our purposes, we want these data to be specific slices of the array. In order to define and properly use the prepending (referred to as *left padding*) and appending (referred to as *right padding*) operations in MoA, we introduce some new notation, operations, and properties.

Definition 1. *Given an n-dimensional array A and an interger $i \in \mathbb{Z}/n\mathbb{Z}$, we define the shorthand notation*

$$A_{\langle k_0 \ \dots \ k_i \rangle} = \langle k_0 \ \dots \ k_i \rangle \ \psi \ A.$$

Definition 2. *Given an n-dimensional array A' such that*

$$\rho(A') = \langle s_0 \ \dots \ s_{n-1} \rangle$$

we write

$$\rho\text{'}(A') = \langle s_{0_{b_0,e_0}} \ \dots \ s_{n-1_{b_{n-1},e_{n-1}}} \rangle$$

the shape of A' annotated with the shape information of the n-dimensional slice A of A' defined by

$$\rho(A) = \langle e_0 - b_0 \ \dots \ e_{n-1} - b_{n-1} \rangle$$

and

$$A_{\langle k_0 \ \dots \ k_{n-1} \rangle} = A'_{\langle k_0 + b_0 \ \dots \ k_{n-1} + b_{n-1} \rangle}.$$

For $i \in \mathbb{Z}/n\mathbb{Z}$, if a given dimension s_i is not annotated, we assume $b_i = 0$ and $e_i = s_i$. For an array A', if the result of $\rho\text{'}(A')$ is left unspecified, we assume $\rho\text{'}(A') = \rho(A')$.

Definition 3. *Given an array A such that*

$$\rho\text{'}(A) = \langle s_{0_{b_0,e_0}} \ \dots \ s_{n-1_{b_{n-1},e_{n-1}}} \rangle$$

and an integer $i \in \mathbb{Z}/n\mathbb{Z}$ we define the right padding operation on axis i as $\text{padr}_i(A) = A^{(r)}$ such that

$$A^{(r)}_{\langle k_0 \ \dots \ k_{i-1} \rangle} = \text{cat}(A_{\langle k_0 \ \dots \ k_{i-1} \rangle}, A_{\langle k_0 \ \dots \ k_{i-1} \ b_i + s_i - e_i \rangle})$$

for j, k_j integers such that $0 \leq j < i$, $0 \leq k_j < s_j$.
The shape of $A^{(r)}$ is given by

$$\rho\text{'}(A^{(r)}) = \langle s_{0_{b_0,e_0}} \ \dots \ (s_i + 1)_{b_i,e_i} \ \dots \ s_{n-1_{b_{n-1},e_{n-1}}} \rangle.$$

In the same way, we define the left padding operation on axis i as $\text{padl}_i(A) = A^{(l)}$ such that

$$A^{(l)}_{\langle k_0 \ \dots \ k_{i-1} \rangle} = \text{cat}(A_{\langle k_0 \ \dots \ k_{i-1} \ e_i - b_i - 1 \rangle}, A_{\langle k_0 \ \dots \ k_{i-1} \rangle}).$$

The shape of $A^{(l)}$ is given by

$$\rho\text{'}(A^{(l)}) = \langle s_{0_{b_0,e_0}} \ \dots \ (s_i + 1)_{b_i+1,e_i+1} \ \dots \ s_{n-1_{b_{n-1},e_{n-1}}} \rangle.$$

Finally, we call unpadl *(respectively* unpadr*) the inverse function of* padl *(respectively* padr*).*

Proposition 1. *For a given axis i, padl_i and padr_i are commutative, i.e.*

$$\text{padl}_i \circ \text{padr}_i = \text{padr}_i \circ \text{padl}_i \, .$$

Proposition 1 can be proven using associativity of cat.

Proposition 2. *Let A be an array without right padding, i.e. an array such that*

$$\rho\text{'}(A) = \langle s_{0_{b_0,e_0}} \ \dots \ s_{n-1_{b_{n-1},e_{n-1}}} \rangle$$

with $e_i = s_i$ for all $i \in \mathbb{Z}/n\mathbb{Z}$. For an axis i of A and an integer $m \in \{0, \dots, s_i\}$, let $A^{(rm)} = \text{padr}_i^m(A)$. Then, the following holds:

$$A^{(rm)}_{\langle k_0 \ \dots \ k_{i-1} \rangle} = \text{cat}(A_{\langle k_0 \ \dots \ k_{i-1} \rangle}, \text{take}(m, \text{drop}(b_i, A_{\langle k_0 \ \dots \ k_{i-1} \rangle}))).$$

In the same way, for A an array without left padding and $A^{(lm)} = \text{padl}_i^m(A)$, then we have

$$A^{(lm)}_{\langle k_0 \ \dots \ k_{i-1} \rangle} = \text{cat}(\text{drop}(e_i - b_i - m, \text{take}(e_i - b_i, A_{\langle k_0 \ \dots \ k_{i-1} \rangle})), A_{\langle k_0 \ \dots \ k_{i-1} \rangle}).$$

Proof 1. *We want to prove that*

$$A^{(rm)}_{\langle k_0 \ \dots \ k_{i-1} \rangle} = \text{cat}(A_{\langle k_0 \ \dots \ k_{i-1} \rangle}, \text{take}(m, \text{drop}(b_i, A_{\langle k_0 \ \dots \ k_{i-1} \rangle}))) \quad (1)$$

This can be done by induction on m.
Base step: *assume $m = 0$, then*

$$A^{(r0)}_{\langle k_0 \ \dots \ k_{i-1} \rangle} = \text{cat}(A_{\langle k_0 \ \dots \ k_{i-1} \rangle}, \text{take}(0, \text{drop}(b_i, A_{\langle k_0 \ \dots \ k_{i-1} \rangle})))$$
$$\Leftrightarrow \ \text{padr}_i^0(A)_{\langle k_0 \ \dots \ k_{i-1} \rangle} = \text{cat}(A_{\langle k_0 \ \dots \ k_{i-1} \rangle}, \epsilon)$$
$$\Leftrightarrow A_{\langle k_0 \ \dots \ k_{i-1} \rangle} = \text{cat}(A_{\langle k_0 \ \dots \ k_{i-1} \rangle}, \epsilon)$$
$$\Leftrightarrow A_{\langle k_0 \ \dots \ k_{i-1} \rangle} = A_{\langle k_0 \ \dots \ k_{i-1} \rangle}$$

Inductive step: *assume there exists m satisfying $m < s_i - b_i$ such that Equation 1 holds. Then, we have that*

$$A^{(r(m+1))}_{\langle k_0 \ \dots \ k_{i-1} \rangle} = \text{cat}(A_{\langle k_0 \ \dots \ k_{i-1} \rangle}, \text{take}(m+1, \text{drop}(b_i, A_{\langle k_0 \ \dots \ k_{i-1} \rangle})))$$

$$\Leftrightarrow \operatorname{padr}_i(A^{(rm)})_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \operatorname{cat}(A_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{take}(m+1, \operatorname{drop}(b_i, A_{\langle k_0\ \ldots\ k_{i-1}\rangle})))$$

$$\Leftrightarrow \operatorname{cat}(\operatorname{cat}(A_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{take}(m, \operatorname{drop}(b_i, A_{\langle k_0\ \ldots\ k_{i-1}\rangle}))), A^{(rm)}_{\langle k_0\ \ldots\ k_{i-1}\ b_i+(s_i+m)-e_i\rangle}) = \operatorname{cat}(A_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{take}(m+1, \operatorname{drop}(b_i, A_{\langle k_0\ \ldots\ k_{i-1}\rangle})))$$

$$\Leftrightarrow \operatorname{cat}(A_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{cat}(\operatorname{take}(m, \operatorname{drop}(b_i, A_{\langle k_0\ \ldots\ k_{i-1}\rangle})), A^{(rm)}_{\langle k_0\ \ldots\ k_{i-1}\ b_i+(s_i+m)-e_i\rangle})) = \operatorname{cat}(A_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{take}(m+1, \operatorname{drop}(b_i, A_{\langle k_0\ \ldots\ k_{i-1}\rangle})))$$

$$\Leftrightarrow A^{(rm)}_{\langle k_0\ \ldots\ k_{i-1}\ b_i+(s_i+m)-e_i\rangle} = A_{\langle k_0\ \ldots\ k_{i-1}\ b_i+m\rangle}$$

$$\Leftrightarrow A^{(rm)}_{\langle k_0\ \ldots\ k_{i-1}\ b_i+(e_i+m)-e_i\rangle} = A_{\langle k_0\ \ldots\ k_{i-1}\ b_i+m\rangle}$$

$$\Leftrightarrow A^{(rm)}_{\langle k_0\ \ldots\ k_{i-1}\ b_i+m\rangle} = A_{\langle k_0\ \ldots\ k_{i-1}\ b_i+m\rangle}$$

which is given by definition, since $b_i + m$ is a valid index in $A_{\langle k_0\ \ldots\ k_{i-1}\rangle}$. □

Similarly, a proof for the second case of Proposition 2 can be made by induction on m.

Remark 1. *For simplicity, we ignore the case when $m > s_i$ for an axis i in Proposition 2, as it would require chaining several concatenation operations since* take *does not behave well in that case. However, it works the same way in principle.*

Proposition 3. *Let x, x_1, x_2 be n-dimensional MoA expressions and* op *a binary map operation such that*

$$x = x_1 \operatorname{op} x_2$$

then on any axis i of x_1 and x_2, we have that

$$x = \operatorname{unpadr}_i(\operatorname{padr}_i(x)) = \operatorname{unpadr}_i(\operatorname{padr}_i(x_1) \operatorname{op} \operatorname{padr}_i(x_2)).$$

Similarly, we have that

$$x = \operatorname{unpadl}_i(\operatorname{padl}_i(x)) = \operatorname{unpadl}_i(\operatorname{padl}_i(x_1) \operatorname{op} \operatorname{padl}_i(x_2)).$$

This idea is trivially extensible to unary map operations.

Proof 2. *Since* op *is a binary map operation, we have:*

$$(\operatorname{padr}_i(x_1) \operatorname{op} \operatorname{padr}_i(x_2))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \operatorname{cat}(x_{1\langle k_0\ \ldots\ k_{i-1}\rangle}, x_{1\langle k_0\ \ldots\ k_{i-1}\ b_i+s_i-e_i\rangle}) \operatorname{op} \operatorname{cat}(x_{2\langle k_0\ \ldots\ k_{i-1}\rangle}, x_{2\langle k_0\ \ldots\ k_{i-1}\ b_i+s_i-e_i\rangle})$$

$$\Leftrightarrow (\operatorname{padr}(x_1) \operatorname{op} \operatorname{padr}_i(x_2))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \operatorname{cat}(x_{1\langle k_0\ \ldots\ k_{i-1}\rangle} \operatorname{op} x_{2\langle k_0\ \ldots\ k_{i-1}\rangle}, x_{1\langle k_0\ \ldots\ k_{i-1}\ b_i+s_i-e_i\rangle} \operatorname{op} x_{2\langle k_0\ \ldots\ k_{i-1}\ b_i+s_i-e_i\rangle})$$

$$\Leftrightarrow \operatorname{unpadr}_i(\operatorname{padr}_i(x_1) \operatorname{op} \operatorname{padr}_i(x_2))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = x_{1\langle k_0\ \ldots\ k_{i-1}\rangle} \operatorname{op} x_{2\langle k_0\ \ldots\ k_{i-1}\rangle}$$

$$\Leftrightarrow \operatorname{unpadr}_i(\operatorname{padr}_i(x_1) \operatorname{op} \operatorname{padr}_i(x_2))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = x_{\langle k_0\ \ldots\ k_{i-1}\rangle}$$

$$\Leftrightarrow \operatorname{unpadr}_i(\operatorname{padr}_i(x_1) \operatorname{op} \operatorname{padr}_i(x_2)) = x.$$

□

Proof 3. *Since* op *is a binary map operation, we have:*

$$(\operatorname{padl}_i(x_1) \operatorname{op} \operatorname{padl}_i(x_2))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \operatorname{cat}(x_{1\langle k_0\ \ldots\ k_{i-1}\ e_i-b_i-1\rangle}, x_{1\langle k_0\ \ldots\ k_{i-1}\rangle}) \operatorname{op} \operatorname{cat}(x_{2\langle k_0\ \ldots\ k_{i-1}\ e_i-b_i-1\rangle}, x_{2\langle k_0\ \ldots\ k_{i-1}\rangle})$$

$$\Leftrightarrow (\operatorname{padl}_i(x_1) \operatorname{op} \operatorname{padl}_i(x_2))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \operatorname{cat}(x_{1\langle k_0\ \ldots\ k_{i-1}\ e_i-b_i-1\rangle} \operatorname{op} x_{2\langle k_0\ \ldots\ k_{i-1}\ e_i-b_i-1\rangle}, x_{1\langle k_0\ \ldots\ k_{i-1}\rangle} \operatorname{op} x_{2\langle k_0\ \ldots\ k_{i-1}\rangle})$$

$$\Leftrightarrow \operatorname{unpadl}_i(\operatorname{padl}_i(x_1) \operatorname{op} \operatorname{padl}_i(x_2))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = x_{1\langle k_0\ \ldots\ k_{i-1}\rangle} \operatorname{op} x_{2\langle k_0\ \ldots\ k_{i-1}\rangle}$$

$$\Leftrightarrow \operatorname{unpadl}_i(\operatorname{padl}_i(x_1) \operatorname{op} \operatorname{padl}_i(x_2))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = e_{\langle k_0\ \ldots\ k_{i-1}\rangle}$$

$$\Leftrightarrow \operatorname{unpadl}_i(\operatorname{padl}_i(x_1) \operatorname{op} \operatorname{padl}_i(x_2)) = e.$$

□

Proposition 4. *Let x, x' be n-dimensional unpadded MoA expressions, j an axis of x' and r an integer such that*

$$x = r\ \theta_j\ x'$$

then on any axis i of x', we have that

$$x = \operatorname{unpadr}_i^{m_2}(\operatorname{unpadl}_i^{m_1}(\operatorname{padl}_i^{m_1}(\operatorname{padr}_i^{m_2}(x)))) = \operatorname{unpadr}_i^{m_2}(\operatorname{unpadl}_i^{m_1}(r\ \theta_j\ \operatorname{padl}_i^{m_1}(\operatorname{padr}_i^{m_2}(x'))))$$

holds if either one of the following cases holds:

1. $j \neq i$;
2. $r = 0$;
3. $r < 0$ *and* $m_2 \geq |r|$;
4. $r > 0$ *and* $m_1 \geq r$.

Proof 4. *We do not prove cases* 1 *and* 2 *of Proposition 4 since they are trivial (in case* 1*, padding does not affect the rotation, and in case* 2 *we have $0\ \theta_j\ x' = x'$). We thus want to prove that*

$$i = j, r < 0, m_2 \geq |r| \implies x = \operatorname{unpadr}_i^{m_2}(\operatorname{unpadl}_i^{m_1}(r\ \theta_j\ \operatorname{padl}_i^{m_1}(\operatorname{padr}_i^{m_2}(x'))))$$

Using Proposition 2, we can write $\operatorname{padl}_i^{m_1}(\operatorname{padr}_i^{m_2}(x'))$ as an array A such that

$$A_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \operatorname{cat}(pl, \operatorname{cat}(x'_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{take}(m_2, \operatorname{drop}(b_i, x'_{\langle k_0\ \ldots\ k_{i-1}\rangle}))))$$

where pl represents the left-padding of the array. Since x' is originally unpadded, we rewrite A as:

$$A_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \operatorname{cat}(pl, \operatorname{cat}(x'_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{take}(m_2, \operatorname{drop}(0, x'_{\langle k_0\ \ldots\ k_{i-1}\rangle})))) = \operatorname{cat}(pl, \operatorname{cat}(x'_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{take}(m_2, x'_{\langle k_0\ \ldots\ k_{i-1}\rangle}))).$$

Since $r < 0$, we have:

$$(r\ \theta_i\ A)_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \operatorname{cat}(\operatorname{drop}(|r|, \operatorname{cat}(pl, \operatorname{cat}(x'_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{take}(m_2, x'_{\langle k_0\ \ldots\ k_{i-1}\rangle})))), \operatorname{take}(|r|, \operatorname{cat}(pl, \operatorname{cat}(x'_{\langle k_0\ \ldots\ k_{i-1}\rangle}, \operatorname{take}(m_2, x'_{\langle k_0\ \ldots\ k_{i-1}\rangle})))))$$

$$= \operatorname{cat}(\operatorname{drop}(|r|, \operatorname{cat}(\operatorname{cat}(pl, x'_{\langle k_0\ \ldots\ k_{i-1}\rangle}), \operatorname{take}(m_2, x'_{\langle k_0\ \ldots\ k_{i-1}\rangle}))),$$

$$\text{take}(|r|, \text{cat}(pl, \text{cat}(x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}, \text{take}(m_2, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}))))).$$

Using Proposition 1, we get

$$\text{unpadr}_i^{m_2}(\text{unpadl}_i^{m_1}(r\ \theta_i\ A)) = \text{unpadl}_i^{m_1}(\text{unpadr}_i^{m_2}(r\ \theta_i\ A)),$$

and since $m_2 \geq |r|$, we have

$$\begin{aligned}&\text{unpadr}_i^{m_2}(r\ \theta_i\ A))_{\langle k_0 \ \dots \ k_{i-1}\rangle}\\ &= \text{drop}(|r|, \text{cat}(\text{cat}(pl, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}),\\ &\qquad \text{take}(m_2 - (m_2 - |r|), x'_{\langle k_0 \ \dots \ k_{i-1}\rangle})))\\ &= \text{drop}(|r|, \text{cat}(\text{cat}(pl, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}), \text{take}(|r|, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}))).\end{aligned}$$

We can thus write:

$$\begin{aligned}&\text{unpadl}_i^{m_1}(\text{unpadr}_i^{m_2}(r\ \theta_i\ A)))_{\langle k_0 \ \dots \ k_{i-1}\rangle}\\ &= \text{drop}(m_1, \text{drop}(|r|, \text{cat}(\text{cat}(pl, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}), \text{take}(|r|, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}))))\\ &= \text{drop}(m_1 + |r|, \text{cat}(\text{cat}(pl, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}), \text{take}(|r|, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}))).\end{aligned}$$

Since r is a valid rotation offset in x', we can write

$$\begin{aligned}&\text{drop}(m_1 + |r|, \text{cat}(\text{cat}(pl, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}), \text{take}(|r|, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle})))\\ &= \text{cat}(\text{drop}(m_1 + |r|, \text{cat}(pl, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle})), \text{take}(|r|, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}))\\ &= \text{cat}(\text{drop}(|r|, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}), \text{take}(|r|, x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}))\\ &= r\ \theta_i\ x'_{\langle k_0 \ \dots \ k_{i-1}\rangle}\end{aligned}$$

and thus

$$x = \text{unpadr}_i^{m_2}(\text{unpadl}_i^{m_1}(r\ \theta_j\ \ \text{padl}_i^{m_1}(\text{padr}_i^{m_2}(x'))))$$

by function extensionality. □

The proof for case 4 *follows the same pattern as case* 3 *on the opposite side of the array.*

Proposition 5. *Given an expression x with*

$$\rho\text{'}(x) = \langle s_{0_{b_0,e_0}} \ \dots \ s_{n-1_{b_{n-1},e_{n-1}}}\rangle$$

and a padding p on an axis $i \in \mathbb{Z}/n\mathbb{Z}$ such that

$$p = \text{padl}_i^{m_1}\ \text{padr}_i^{m_2},$$

define the expression

$$x' = p(x).$$

Then, evaluating x' in indexes whose i-th component k_i is such that $m_1 \leq i < s_i + m_1$ is equivalent to evaluating x in all of its indexes. We call the values m_1 and m_2 consumption speed for a given axis i in the following, and define a function speed_i *on expressions such that*

$$\text{speed}_i(x') = (m_1, m_2).$$

Example 1. Consider once again the following DNF expression:

$$x = \langle i\ j\rangle\ \psi\ ((1\ \theta_0\ A) + (-1\ \theta_0\ A)),$$

with

$$A = \begin{pmatrix} 1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix}.$$

This time, $\rho(A) = \langle 2\ 3\rangle$. After applying the ψ-correspondence theorem like in the previous example, this reduces to the following generic program:

$$\begin{aligned}&\forall j\ \ s.t.\ \ 0 \leq j < 3\\ &\quad A[((i+1) \bmod 2) \times 3 + j]+\\ &\quad A[((i-1) \bmod 2) \times 3 + j].\end{aligned}$$

In order to get rid of the mod operation, we create a new array A' such that

$$A' = \text{padl}_0(\text{padr}_0(A))$$

From Definition 3, we have that

$$\rho\text{'}(A') = \langle 4_{1,3}\ 3\rangle$$

and

$$A' = \left(\begin{array}{ccc} 4 & 5 & 6\\ \hline 1 & 2 & 3\\ 4 & 5 & 6\\ \hline 1 & 2 & 3\end{array}\right).$$

By definition of padr and padl, we have the following:

$$\begin{aligned}x &= \text{unpadr}_0(\text{unpadl}_0(\text{padl}_0(\text{padr}_0(x))))\\ &= \text{unpadr}_0(\text{unpadl}_0(\text{padl}_0(\text{padr}_0((1\ \theta_0\ A) + (-1\ \theta_0\ A))))).\end{aligned}$$

By applying Proposition 3, we get:

$$\begin{aligned}x &= \text{unpadr}_0(\text{unpadl}_0(\text{padl}_0(\text{padr}_0((1\ \theta_0\ A) + (-1\ \theta_0\ A)))))\\ &= \text{unpadr}_0(\text{unpadl}_0(\text{padl}_0(\text{padr}_0(1\ \theta_0\ A))+\\ &\qquad \text{padl}_0(\text{padr}_0(-1\ \theta_0\ A)))).\end{aligned}$$

Finally, we apply Proposition 4, and get:

$$\begin{aligned}x &= \text{unpadr}_0(\text{unpadl}_0(\text{padl}_0(\text{padr}_0(1\ \theta_0\ A))+\\ &\qquad \text{padl}_0(\text{padr}_0(-1\ \theta_0\ A))))\\ &= \text{unpadr}_0(\text{unpadl}_0((1\ \theta_0\ \ \text{padl}_0(\text{padr}_0(A)))+\\ &\qquad (-1\ \theta_0\ \ \text{padl}_0(\text{padr}_0(A)))))\\ &= \text{unpadr}_0(\text{unpadl}_0((1\ \theta_0\ A') + (-1\ \theta_0\ A'))).\end{aligned}$$

We can now transform it to ONF; the bounds of the relevant indices i and j are given by Proposition 5. Thus, we have the following:

$$\begin{aligned}&\forall i\ \ s.t.\ \ 1 \leq i < 4 - 1\\ &\quad \langle i\ j\rangle\ \psi\ ((1\ \theta_0\ A') + (-1\ \theta_0\ A'))\\ &\quad \equiv (\text{rav}\, A')[\gamma(\langle i+1\rangle; \langle 4\rangle) \times 3 + \iota 3]+\\ &\quad (\text{rav}\, A')[\gamma(\langle i-1\rangle; \langle 4\rangle) \times 3 + \iota 3].\end{aligned}$$

We then apply γ, rav and turn ι into a loop, and we get the following generic program:

$$\begin{aligned}&\forall j\ \ s.t.\ \ 0 \leq j < 3\\ &\quad A'[(i+1) \times 3 + j]+\\ &\quad A'[(i-1) \times 3 + j].\end{aligned}$$

Finally, we apply $\text{unpadr}_0\ \text{unpadl}_0$ and retrieve the exact same result as we would have gotten evaluating x. Notice that thanks to the notion of consumption speed, we managed to avoid computing the result of the expression for irrelevant indices. In the end, both of the expressions had 6 loop iterations, but we managed to get rid of the expensive mod operation by adding data redundancy into A.

5.1.2 Non-Uniform Memory Access

Consider now that the example expression is embedded within a loop and needs to be executed several times. Then, in order to avoid the mod operation at each iteration, the array must be padded at each iteration as well.

Considering hardware and software implementations, it is reasonable to investigate a case in which the application of a big padding operation $p = \text{padl}_i^n\ \text{padr}_i^m$ for some positive integers n, m, i at a given point in the program is cheaper than applying parts of p in different parts of the program. For example, if the padding operation depends on inter-process communication, it is usually significantly cheaper to open one socket and send four elements than to open two sockets each sending two elements (each opened connection probably also requiring synchronization of some sort, etc). We however consider unpadding to have negligible cost.

But, to be able to reduce the number of loci where a padding operation is required and to use the resulting padding efficiently, we need to define a slightly more complex padding function as well as further notation. We also need to define the notion of *padding exhaustion.*

Informally, padding exhaustion corresponds to reaching the state where there is not enough unconsumed padding left to apply the expression. Ideally, this state is reached at the end of the computation of all the expressions; if it is reached in the middle of execution, however, the padding must be replenished to proceed.

Definition 4. *Given a n-dimensional array $A^{(dl)}$ such that*

$$\rho(A^{(dl)}) = \langle s_0\ \dots\ s_{n-1} \rangle$$

we write

$$\rho^{(dl)}(A^{(dl)}) = \langle s_{0_{\text{dl}(d_0,b_0,e_0)}}\ \dots\ s_{n-1_{\text{dl}(d_{n-1},b_{n-1},e_{n-1})}} \rangle$$

the shape of $A^{(dl)}$ annotated with the shape information of the underlying n-dimensional array A defined by

$$\rho(A) = \langle (e_0 - b_0) \times d_0\ \dots\ (e_{n-1} - b_{n-1}) \times d_{n-1} \rangle$$

such that $s_i \equiv 0 \bmod d_i$ for $i \in \mathbb{Z}/n\mathbb{Z}$ and

$$A_{\langle k_0\ \dots\ k_{n-1} \rangle} = A^{(dl)}_{\langle \text{dlb}_0^{(A^{(dl)})}(k_0)\ \dots\ \text{dlb}_{n-1}^{(A^{(dl)})}(k_{n-1}) \rangle}$$

with

$$\text{dlb}_i^{(A^{(dl)})}(k) = \left\lfloor \frac{k}{e_i - b_i} \right\rfloor \times \frac{s_i}{d_i} + b_i + k \bmod (e_i - b_i).$$

Note that the notation provided by ρ' is a special case of the notation provided by $\rho^{(dl)}$ where $d_i = 1$ for all $i \in \mathbb{Z}/n\mathbb{Z}$. The notations b_i, e_i and $\text{dl}(1, b_i, e_i)$ are thus considered equivalent in the following.

Definition 5. *Given an array A such that*

$$\rho^{(dl)}(A) = \langle s_{0_{\text{dl}(d_0,b_0,e_0)}}\ \dots\ s_{n-1_{\text{dl}(d_0,b_{n-1},e_{n-1}}} \rangle,$$

and an axis i, we write $q_i = \frac{s_i}{d_i}$. We then define the dimension lifting operation on axis i as $\text{dlift}_i(A) = A^{(lift)}$ such that $A^{(lift)}$ has $n + 1$ dimensions,

$$\rho^{(dl)}(A^{(lift)}) = \langle s_{0_{\text{dl}(d_0,b_0,e_0)}}\ \dots\ d_i\ q_{i_{b_i,e_i}}\ \dots\ s_{n-1_{\text{dl}(d_{n-1},b_{n-1},e_{n-1})}} \rangle,$$

and

$$A^{(lift)}_{\langle k_0\ \dots\ k_{i-1} \rangle} = \text{take}(q_i, \text{drop}(k_{i-1} \times q_i, A_{\langle k_0\ \dots\ k_{i-2} \rangle}))$$

for any valid index $\langle k_0\ \dots\ k_{i-1} \rangle$ in $A^{(lift)}$.

Definition 6. *Consider an array A such that*

$$\rho^{(dl)}(A) = \langle s_{0_{\text{dl}(d_0,b_0,e_0)}}\ \dots\ s_{n-1_{\text{dl}(d_0,b_{n-1},e_{n-1}}} \rangle.$$

In a MoA setting without any notion of padding, any array is implicitly annotated with $d_i = 1$, $b_i = 0$ and $e_i = s_i$ on any given axis i. However, the dimension lifting operation on axis i if $d_i = 1$ is the identity operation. To properly use dlift *as it is defined above, we do the following: assuming $b_i = 0$ and $e_i = s_i$ for a given axis i of A, we define the prelift operation on that axis for any d such that $s_i \equiv 0 \bmod d$ as $A^{(pl)} = \text{prelift}_i(d, A)$, such that*

$$\rho^{(dl)}(A^{(pl)}) = \langle s_{0_{\text{dl}(d_0,b_0,e_0)}}\ \dots\ s_{i_{\text{dl}(d,0,s_i/d)}}\ \dots\ s_{n-1_{\text{dl}(d_0,b_{n-1},e_{n-1}}} \rangle$$

and

$$A_{\langle k_0\ \dots\ k_{n-1} \rangle} = A^{(pl)}_{\langle k_0\ \dots\ k_{n-1} \rangle}.$$

The precondition on prelift_i *means that it can only be applied to arrays that are unpadded on axis i.*

Definition 7. *Given an array A with shape*

$$\langle s_{0_{\text{dl}(d_0,b_0,e_0)}}\ \dots\ s_{n-1_{\text{dl}(d_{n-1},b_{n-1},e_{n-1}}} \rangle$$

and an integer $i \in \mathbb{Z}/n\mathbb{Z}$ we define the right pre-dimension lifting padding operation on axis i as $\text{dlpadr}_i(A) = A^{(rdl)}$ such that

$$\begin{aligned}\text{dlift}_i(A^{(rdl)})_{\langle k_0\ \dots\ k_{i-1} \rangle} = {}&\text{cat}(\text{take}(q_i, \text{drop}(k_{i-1} \times q_i,\\ &\qquad A_{\langle k_0\ \dots\ k_{i-2} \rangle})),\\ &\quad A_{\langle k_0\ \dots\ ((k_{i-1}+2)\times q_i + b_i - e_i) \bmod s_i \rangle})\end{aligned}$$

for j, k_j integers such that $0 \le j < i$, $0 \le k_j < s_j$. Not that we consider operations on the axis i to be done in $\mathbb{Z}/n\mathbb{Z}$, e.g. for $i = 0$, we have $k_{i-1} = k_{n-1}$.
The shape of $A^{(rdl)}$ is given by

$$\begin{aligned}&\rho^{(dl)}(A^{(rdl)})\\ &= \langle s_{0_{\text{dl}(d_0,b_0,e_0)}}\ \dots\ (s_i + d_i)_{\text{dl}(d_i,b_i,e_i)}\ \dots\ s_{n-1_{\text{dl}(d_{n-1},b_{n-1},e_{n-1}}} \rangle.\end{aligned}$$

In the same way, we define the left pre-dimension lifting padding operation on axis i as $\text{dlpadl}_i(A) = A^{(ldl)}$ such that

$$\begin{aligned}\text{dlift}_i(A^{(ldl)})_{\langle k_0\ \dots\ k_{i-1} \rangle} = {}&\text{cat}(A_{\langle k_0\ \dots\ ((k_{i-1}-1)\times q_i + e_i - b_i - 1) \bmod s_i \rangle},\\ &\quad \text{take}(q_i, \text{drop}(k_{i-1} \times q_i, A_{\langle k_0\ \dots\ k_{i-2} \rangle})))\end{aligned}$$

The shape of $A^{(ldl)}$ is given by

$$\rho^{(dl)}(A^{(rdl)})$$

$=\langle s_{0_{\mathrm{dl}(d_0,b_0,e_0)}} \ \ldots \ (s_i+d_i)_{\mathrm{dl}(d_i,b_i+1,e_i+1)} \ \ldots \ s_{n-1_{\mathrm{dl}(d_{n-1},b_{n-1},e_{n-1}}}\rangle$.

Finally, we call dlunpadl *(respectively* dlunpadr*) the inverse function of* dlpadl *(respectively* dlpadr*).*

Proposition 6. *For a given axis i,* dlpadl_i *and* dlpadr_i *are commutative, i.e.*

$$\mathrm{dlpadl}_i \circ \mathrm{dlpadr}_i = \mathrm{dlpadr}_i \circ \mathrm{dlpadl}_i \,.$$

Proposition 6 can be proven using the associativity of cat.

Proposition 7. *Let* $A, A^{(rdl)}$ *be two arrays and i an integer such that*

$$A^{(rdl)} = \mathrm{dlpadr}_i(A).$$

Then,

$$\begin{aligned}&\mathrm{unpadr}_i(\mathrm{dlift}_i(A^{(rdl)})_{\langle k_0\ \ldots\ k_{i-1}\rangle}) \\ &= \mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, A_{\langle k_0\ \ldots\ k_{i-2}\rangle}))\end{aligned} \tag{2}$$

holds. In the same way, for an array $A^{(ldl)}$ *such that*

$$A^{(ldl)} = \mathrm{dlpadl}_i(A),$$

then, we have

$$\begin{aligned}&\mathrm{unpadl}_i(\mathrm{dlift}_i(A^{(ldl)})_{\langle k_0\ \ldots\ k_{i-1}\rangle}) \\ &= \mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, A_{\langle k_0\ \ldots\ k_{i-2}\rangle})).\end{aligned} \tag{3}$$

Proof 5. *We give a proof for Equation 2:*

$$\begin{aligned}&\mathrm{unpadr}_i(\mathrm{dlift}_i(A^{(rdl)})_{\langle k_0\ \ldots\ k_{i-1}\rangle}) \\ &= \mathrm{unpadr}_i(\mathrm{cat}(\mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, A_{\langle k_0\ \ldots\ k_{i-2}\rangle})), \\ &\qquad A_{\langle k_0\ \ldots\ ((k_{i-1}+2)\times q_i+b_i-e_i) \bmod s_i\rangle})) \\ &= \mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, A_{\langle k_0\ \ldots\ k_{i-2}\rangle})).\end{aligned}$$

□

Proof 6. *We give a proof for Equation 3:*

$$\begin{aligned}&\mathrm{unpadl}_i(\mathrm{dlift}_i(A^{(ldl)})_{\langle k_0\ \ldots\ k_{i-1}\rangle}) \\ &= \mathrm{unpadl}_i(\mathrm{cat}(A_{\langle k_0\ \ldots\ ((k_{i-1}-1)\times q_i+e_i-b_i-1) \bmod s_i\rangle}, \\ &\qquad \mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, A_{\langle k_0\ \ldots\ k_{i-2}\rangle})))) \\ &= \mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, A_{\langle k_0\ \ldots\ k_{i-2}\rangle}))\end{aligned}$$

□

Informally, Proposition 7 tells us that for a given array $A^{(\mathrm{dl})}$ resulting from padding and dimension lifting an array A on an axis i, unpadding and concatenating all the subarrays resulting from the dimension lifting operation is the same as concatenating them and *dlunpadding* the result.

Proposition 8. *Let A be an array without right padding on its i-th axis, i.e. an array such that*

$$\rho^{(dl)}(A) = \langle s_{0_{dl(d_0,b_0,e_0)}} \ \ldots \ s_{n-1_{dl(d_{n-1},b_{n-1},e_{n-1})}}\rangle$$

with $e_i = s_i$, $i \in \mathbb{Z}/n\mathbb{Z}$. *Given an integer* $m \in \{0, \ldots, s_i\}$, *let* $A^{(rdlm)} = \mathrm{dlpadr}_i^m(A)$. *Then, the following holds:*

$$\begin{aligned}&\mathrm{dlift}_i(A^{(rdlm)})_{\langle k_0\ \ldots\ k_{i-1}\rangle} \\ &= \mathrm{cat}(\mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, A_{\langle k_0\ \ldots\ k_{i-2}\rangle})), \\ &\quad \mathrm{take}(m, drop(((k_{i-1}+1)\times q_i+b_i) \bmod s_i, A_{\langle k_0\ \ldots\ k_{i-2}\rangle})))\end{aligned}$$

In the same way, for A an array without left padding and $A^{(ldlm)} = \mathrm{dlpadl}_i^m(A)$, *then*

$$\begin{aligned}&\mathrm{dlift}_i(A^{(ldlm)})_{\langle k_0\ \ldots\ k_{i-1}\rangle} \\ &= \mathrm{cat}(\mathrm{drop}(e_i-b_i-m, \mathrm{take}(e_i-b_i, \mathrm{drop}((k_{i-1}-1)\times q_i \bmod s_i, \\ &\quad A_{\langle k_0\ \ldots\ k_{i-2}\rangle}))), \mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, A_{\langle k_0\ \ldots\ k_{i-2}\rangle})))\end{aligned}$$

holds.

A proof for Proposition 8 may be written using induction on m, similarly to Proof 1.

Proposition 9. *Let* x, x_1, x_2 *be n-dimensional MoA expressions with identical shapes and* op *a binary map operation such that*

$$x = x_1 \,\mathrm{op}\, x_2$$

then on any axis i of x_1 *and* x_2, *we have that*

$$\begin{aligned}x &= \mathrm{dlunpadr}_i(\mathrm{dlunlift}_i(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x)))) \\ &= \mathrm{dlunpadr}_i(\mathrm{dlunlift}_i(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1))\ \mathrm{op} \\ &\qquad \mathrm{dlift}_i(\mathrm{dlpadr}_i(x_2)))).\end{aligned} \tag{4}$$

Similarly, we have that

$$\begin{aligned}x &= \mathrm{dlunpadl}_i(\mathrm{dlunlift}_i(\mathrm{dlift}_i(\mathrm{dlpadl}_i(x)))) \\ &= \mathrm{dlunpadl}_i(\mathrm{dlunlift}_i(\mathrm{dlift}_i(\mathrm{dlpadl}_i(x_1))\ \mathrm{op} \\ &\qquad \mathrm{dlift}_i(\mathrm{dlpadl}_i(x_2)))).\end{aligned} \tag{5}$$

This idea is trivially extensible to unary map operations.

Proof 7. *Since* op *is a binary map operation, we have:*

$$\begin{aligned}&(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1))\ \mathrm{op}\ \mathrm{dlift}_i(\mathrm{dlpadr}_i(x_2)))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \\ &\mathrm{cat}(\mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, x_{1_{\langle k_0\ \ldots\ k_{i-2}\rangle}})), \\ &\quad x_{1_{\langle k_0\ \ldots\ ((k_{i-1}+2)\times q_i+b_i-e_i) \bmod s_i\rangle}})\ \mathrm{op} \\ &\mathrm{cat}(\mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, x_{2_{\langle k_0\ \ldots\ k_{i-2}\rangle}})), \\ &\quad x_{2_{\langle k_0\ \ldots\ ((k_{i-1}+2)\times q_i+b_i-e_i) \bmod s_i\rangle}}) \\ \Leftrightarrow\ &(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1))\ \mathrm{op}\ \mathrm{dlift}_i(\mathrm{dlpadr}_i(x_2)))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \\ &\mathrm{cat}(\quad \mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, x_{1_{\langle k_0\ \ldots\ k_{i-2}\rangle}})) \\ &\qquad \mathrm{op}\ \mathrm{take}(q_i, \mathrm{drop}(k_{i-1}\times q_i, x_{2_{\langle k_0\ \ldots\ k_{i-2}\rangle}})), \\ &\qquad\quad x_{1_{\langle k_0\ \ldots\ ((k_{i-1}+2)\times q_i+b_i-e_i) \bmod s_i\rangle}} \\ &\qquad \mathrm{op}\ \ x_{2_{\langle k_0\ \ldots\ ((k_{i-1}+2)\times q_i+b_i-e_i) \bmod s_i\rangle}}) \\ \Leftrightarrow\ &(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1))\ \mathrm{op}\ \mathrm{dlift}_i(\mathrm{dlpadr}_i(x_2)))_{\langle k_0\ \ldots\ k_{i-1}\rangle} = \\ &(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1\ \mathrm{op}\ x_2)))_{\langle k_0\ \ldots\ k_{i-1}\rangle} \\ \Leftrightarrow\ &\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1))\ \mathrm{op}\ \mathrm{dlift}_i(\mathrm{dlpadr}_i(x_2)) = \\ &\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1\ \mathrm{op}\ x_2)) \\ \Leftrightarrow\ &\mathrm{dlunpadr}_i(\mathrm{dlunlift}_i(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1))\ \mathrm{op} \\ &\qquad\qquad \mathrm{dlift}_i(\mathrm{dlpadr}_i(x_2)))) = \\ &\mathrm{dlunpadr}_i(\mathrm{dlunlift}_i(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1\ \mathrm{op}\ x_2)))) \\ \Leftrightarrow\ &\mathrm{dlunpadr}_i(\mathrm{dlunlift}_i(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1))\ \mathrm{op} \\ &\qquad\qquad \mathrm{dlift}_i(\mathrm{dlpadr}_i(x_2)))) = x_1\ \mathrm{op}\ x_2 \\ \Leftrightarrow\ &\mathrm{dlunpadr}_i(\mathrm{dlunlift}_i(\mathrm{dlift}_i(\mathrm{dlpadr}_i(x_1))\ \mathrm{op}\end{aligned}$$

$$\text{dlift}_i(\text{dlpadr}_i(x_2)))) = x.$$

□

The proof for Equation 5 follows the same pattern as above. Since it does not provide any additional insight, We do not develop it here.

Proposition 10. *Let x, x' be n-dimensional unpadded MoA expressions, j an axis of x' and r an integer such that*

$$x = r\ \theta_j\ x'$$

then on any axis i of x', we have that

$$\begin{aligned} x &= \text{dlunpadr}_i^{m_2}(\text{dlunpadl}_i^{m_1}(\text{dlunlift}_i(\text{dlift}_i(\\ &\quad \text{dlpadl}_i^{m_1}(\text{dlpadr}_i^{m_2}(x)))))) \\ &= \text{dlunpadr}_i^{m_2}(\text{dlunpadl}_i^{m_1}(\text{dlunlift}_i(\\ &\quad r\ \theta_j\ \text{dlift}_i(\text{dlpadl}_i^{m_1}(\text{dlpadr}_i^{m_2}(x')))))) \end{aligned}$$

holds if either one of the following cases holds:

1. $j \neq i$;
2. $r = 0$;
3. $r < 0$ *and* $m_2 \geq |r|$;
4. $r > 0$ *and* $m_1 \geq r$.

A proof for Proposition 10 may be given by following the same pattern as Proof 4.

Example 2. Consider once again the following DNF expression:

$$x = ((1\ \theta_0\ A) + (-1\ \theta_0\ A)),$$

with

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{pmatrix}.$$

This time, we define $\rho(A) = \langle 6\ 3\rangle$. Given an index $\langle i\ j\rangle$ into x, we apply the ψ-correspondence theorem [9] and get the following reduced generic program running on 1 core:

```
∀$j$  $s.t.$  $0 \leq j < 3$
  $A[((i+1) \bmod 6) \times 3 + j] +$
  $A[((i-1) \bmod 6) \times 3 + j]$.
```

We wish to distribute the computation over 2 machines. This can be achieved by a combination of dimension lifting and padding. To distribute the computation over 2 machines it is natural to perform dimension lifting along the 0-th axis of A, taking $d_0 = 2$. We thus start out by creating a new array $A^{(\text{pl})}$ such that

$$A^{(\text{pl})} = \text{prelift}_0(2, A).$$

From Definition 6, we have that

$$\rho^{(\text{dl})}(A^{(\text{pl})}) = \langle 6_{\text{dl}(2,0,3)}\ 3\rangle.$$

Since prelift_0 does not modify the layout of the array it operates on in any way, we have

$$x = ((1\ \theta_0\ A^{(\text{pl})}) + (-1\ \theta_0\ A^{(\text{pl})})).$$

We are now ready to start padding $A^{(\text{pl})}$. In this case, we would like the 2 workers to only communicate at the start and at the end of the computation. To do that, we need to provide each machine with enough padding to do all of the required computations on its own.

This is very similar to what we did in Example 1. We create a new array $A'^{(\text{pl})}$ such that

$$A'^{(\text{pl})} = \text{dlpadl}_0(\text{dlpadr}_0(A^{(\text{pl})})).$$

From Definition 7, we have that

$$\rho^{(\text{dl})}(A'^{(\text{pl})}) = \langle 10_{\text{dl}(2,1,4)}\ 3\rangle$$

and

$$A'^{(\text{pl})} = \left(\begin{array}{ccc} 16 & 17 & 18 \\ \hline 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ \hline 10 & 11 & 12 \\ 7 & 8 & 9 \\ \hline 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \\ \hline 1 & 2 & 3 \end{array}\right).$$

Finally, we create an array $A^{(\text{lift})} = \text{dlift}_0(A'^{(\text{pl})})$. From Definition 5, we have:

$$\rho^{(\text{dl})}(A^{(\text{lift})}) = \langle 2\ 5_{1,4}\ 3\rangle$$

and

$$A^{(\text{lift})}_{\langle 0\rangle} = \left(\begin{array}{ccc} 16 & 17 & 18 \\ \hline 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ \hline 10 & 11 & 12 \end{array}\right)$$

$$A^{(\text{lift})}_{\langle 1\rangle} = \left(\begin{array}{ccc} 7 & 8 & 9 \\ \hline 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \\ \hline 1 & 2 & 3 \end{array}\right).$$

By definition of dlpadr and dlpadl, we have the following:

$$\begin{aligned} x &= \text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlpadl}_0(\text{dlpadr}_0(x)))) \\ &= \text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlunlift}_0(\text{dlift}_0(\text{dlpadl}_0(\text{dlpadr}_0(x)))))) \\ &= \text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlunlift}_0(\text{dlift}_0(\text{dlpadl}_0(\text{dlpadr}_0(\\ &\quad (1\ \theta_0\ A) + (-1\ \theta_0\ A))))))). \end{aligned}$$

By applying Proposition 9, we get:

$$\begin{aligned} x &= \text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlunlift}_0(\text{dlift}_0(\text{dlpadl}_0(\text{dlpadr}_0(\\ &\quad (1\ \theta_0\ A) + (-1\ \theta_0\ A))))))) \\ &= \text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlunlift}_0(\end{aligned}$$

$$\text{dlift}_0(\text{dlpadl}_0(\text{dlpadr}_0(1\ \theta_0\ A)))+$$
$$\text{dlift}_0(\text{dlpadl}_0(\text{dlpadr}_0(-1\ \theta_0\ A)))))).$$

Finally, we apply Proposition 10, and get:

$$\begin{aligned}
x =\ & \text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlunlift}_0(\\
& \quad \text{dlift}_0(\text{dlpadl}_0(\text{dlpadr}_0(1\ \theta_0\ A)))+\\
& \quad \text{dlift}_0(\text{dlpadl}_0(\text{dlpadr}_0(-1\ \theta_0\ A))))))\\
=\ & \text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlunlift}_0(\\
& (1\ \theta_0\ \ \text{dlift}_0(\text{dlpadl}_0(\text{dlpadr}_0(A))))+\\
& (-1\ \theta_0\ \ \text{dlift}_0(\text{dlpadl}_0(\text{dlpadr}_0(A)))))))\\
=\ & \text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlunlift}_0((1\ \theta_0\ A^{(\text{lift})}) + (-1\ \theta_0\ A^{(\text{lift})})))).
\end{aligned}$$

We can now transform the resulting expression x to ONF for each machine. The bounds of i and j are once again given by Proposition 5. Thus, for $c \in \{0, 1\}$, we have the following:

$$\begin{aligned}
&\forall i\ \ s.t.\ \ 1 \le i < 5 - 1\\
&\quad \langle i\ j\rangle\ \psi\ ((1\ \theta_0\ A^{(\text{lift})}_{\langle c\rangle}) + (-1\ \theta_0\ A^{(\text{lift})}_{\langle c\rangle}))\\
&\quad \equiv (\text{rav}\, A^{(\text{lift})}_{\langle c\rangle})[\gamma(\langle i+1\rangle; \langle 5\rangle) \times 3 + \iota 3]+\\
&\quad\ \ \ (\text{rav}\, A^{(\text{lift})}_{\langle c\rangle})[\gamma(\langle i-1\rangle; \langle 4\rangle) \times 3 + \iota 3].
\end{aligned}$$

We then apply γ, rav and turn ι into a loop, and we get the following generic program:

$$\begin{aligned}
&\forall j\ \ s.t.\ \ 0 \le j < 3\\
&\quad A^{(\text{lift})}_{\langle c\rangle}[(i+1) \times 3 + j]+\\
&\quad A^{(\text{lift})}_{\langle c\rangle}[(i-1) \times 3 + j].
\end{aligned}$$

Finally, we join the results using dlunlift_0 and apply dlunpadl_0 and dlunpadr_0 to obtain the same result as we would have gotten evaluating x directly. Moreover, in this case, both expressions had the same number of loop iterations, and *exactly* all the padding was consumed in the computation.

In practice however, what we studied above corresponds to a single step of the PDE solver. Assume the same scenario as above, except that the solver actually runs this step 2 times. For simplicity, we build a function step such that, for a given array A, $\text{step}(A) = x$. Written as a MoA expression, 2 executions of step would be written as $\text{step}^2(A)$.

According to Proposition 5, we have $\text{speed}_0(x) = (1, 1)$. Thus, for the padding to last 2 steps and thus avoid padding exhaustion before the end of the full computation, we need to pad the 0-th axis of A m_l times on the left and m_r times on the right, where m_l and m_r are given by:

$$(m_l, m_r) = 2 \times \text{speed}_0(e) = 2 \times (1, 1) = (2, 2).$$

We start again by creating an $A'^{(\text{pl2})}$ such that

$$A'^{(\text{pl2})} = \text{dlpadl}_0^2(\text{dlpadr}_0^2(A^{(\text{pl})})).$$

From Definition 7, we have that

$$\rho^{(\text{dl})}(A'^{(\text{pl2})}) = \langle 14_{\text{dl}(2,2,5)}\ 3\rangle$$

and

$$A'^{(\text{pl2})} = \left(\begin{array}{ccc}
13 & 14 & 15\\
16 & 17 & 18\\ \hline
1 & 2 & 3\\
4 & 5 & 6\\
7 & 8 & 9\\ \hline
10 & 11 & 12\\
13 & 14 & 15\\
4 & 5 & 6\\
7 & 8 & 9\\ \hline
10 & 11 & 12\\
13 & 14 & 15\\
16 & 17 & 18\\ \hline
1 & 2 & 3\\
4 & 5 & 6
\end{array}\right).$$

Then, we create an array $A^{(\text{lift2})} = \text{dlift}(A'^{(\text{pl2})})$. From Definition 5, we have:

$$\rho^{(\text{dl})}(A^{(\text{lift2})}) = \langle 2\ 7_{2,5}\ 3\rangle$$

and

$$A^{(\text{lift2})}_{\langle 0\rangle} = \left(\begin{array}{ccc}
13 & 14 & 15\\
16 & 17 & 18\\ \hline
1 & 2 & 3\\
4 & 5 & 6\\
7 & 8 & 9\\ \hline
10 & 11 & 12\\
13 & 14 & 15
\end{array}\right)$$

$$A^{(\text{lift2})}_{\langle 1\rangle} = \left(\begin{array}{ccc}
4 & 5 & 6\\
7 & 8 & 9\\ \hline
10 & 11 & 12\\
13 & 14 & 15\\
16 & 17 & 18\\ \hline
1 & 2 & 3\\
4 & 5 & 6
\end{array}\right).$$

Once again, using Proposition 9 and Proposition 10, we get the equation

$$\text{step}^2(A) = \text{dlunpadr}_0^2(\text{dlunpadl}_0^2(\text{dlunlift}_0(\text{step}^2(A^{(\text{lift2})})))).$$

We can now transform the resulting expression x to ONF for each machine. The bounds of i and j are once again given by Proposition 5. Thus, for $c \in \{0, 1\}$, we have the following:

$$\begin{aligned}
&\forall i\ \ s.t.\ \ 1 \le i < 7 - 1\\
&\quad \langle i\ j\rangle\ \psi\ ((1\ \theta_0\ A^{(\text{lift})}_{\langle c\rangle}) + (-1\ \theta_0\ A^{(\text{lift})}_{\langle c\rangle}))\\
&\quad \equiv (\text{rav}\, A^{(\text{lift})}_{\langle c\rangle})[\gamma(\langle i+1\rangle; \langle 7\rangle) \times 3 + \iota 3]+\\
&\quad\ \ \ (\text{rav}\, A^{(\text{lift})}_{\langle c\rangle})[\gamma(\langle i-1\rangle; \langle 7\rangle) \times 3 + \iota 3].
\end{aligned}$$

We one again apply γ, rav and turn ι into a loop, and we get the following generic program:

$$\begin{aligned}
&\forall j\ \ s.t.\ \ 0 \le j < 3\\
&\quad A^{(\text{lift})}_{\langle c\rangle}[(i+1) \times 3 + j] + A^{(\text{lift})}_{\langle c\rangle}[(i-1) \times 3 + j].
\end{aligned}$$

No padding	Padding third axis	Padding second axis
227.12	199.79	196.90

Table 2. Execution time (in seconds) of a 3-dimensional PDE solver implementation with different padding parameters on a single core (CPU: Intel Xeon Gold 6130 CPU @ 2.10GHz).

At that point, we can rewrite our expression as such:

$$\begin{aligned}&\text{step}^2(A)\\&=\text{dlunpadr}_0^2(\text{dlunpadl}_0^2(\text{dlunlift}_0(\text{step}^2(A^{(\text{lift2})}))))\\&=\text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlunlift}_0(\text{step}(\\&\quad\text{dlift}_0(\text{dlunpadr}_0(\text{dlunpadl}_0(\text{dlunlift}_0(\text{step}(A^{(\text{lift2})})))))))))).\end{aligned}$$

But here, as given by Proposition 7, applying

$$\text{dlift}_0\ \text{dlunpadr}_0\ \text{dlunpadl}_0\ \text{dlunlift}_0$$

to $\text{step}(A^{(\text{lift2})})$ is equivalent to applying unpadr_0 and unpadl_0 once to both $A^{(\text{lift2})}_{\langle 0\rangle}$ and $A^{(\text{lift2})}_{\langle 1\rangle}$. As a result, for $c \in \{0, 1\}$,

$$\rho^{(\text{dl})}(\text{unpadr}_0(\text{unpadl}_0(A^{(\text{lift2})}_{\langle c\rangle}))) = \langle 5_{1,4}\ 3\rangle.$$

The rest of the computation follows the single step distributed case study presented directly above. Notice that in this case, 4 intermediate rows of the result were computed twice (once on each machine), resulting in 4 additional outer loop iterations compared to the equivalent single machine unpadded 2-step case. Thus, getting rid of inter-process communication in this case involved both data redundancy and duplicated calculations. Whether it is worth it to perform calculations several times instead of exchanging states between the different computation loci has to be determined based on hardware-dependent cost functions.

6 Experiments

We extended the scenario depicted in Example 1 to our implementation of a PDE solver using a $(-1, 0, 1)$ stencil along every axis; that is to say that at every derivation step, the left and right padding operation are applied once along the specified axis. The execution times of 50 derivation steps given different padding parameters are gathered in Table 2.

We see a performance improvement of roughly 10% between the original code in which no padding was applied and the cases with padding on either axis. Though the elimination of the modulo operation at the edges of the padded axis likely reduces the execution time a little, most of the performance improvement is due to better data locality and thus better cache line usage.

7 Conclusion

We have successfully shown that MoA provides the required building blocks to discuss padding as well as data distribution given an arbitrary architecture, and thus that it is well-suited to explore the space of optimal computations for array expressions at a high level of abstraction. Along the way, we have built 2 examples demonstrating exactly how to use these notions to optimize stencil computations. We expect future work to focus on better qualifying the benefits of using the aforementioned approach instead of already existing solutions; moreover, with the present results as well as those from our previous work focusing on the DNF [3], we will attempt to build a compiler from high-level array expressions to highly efficient code on a given architecture.

Acknowledgments

The research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

References

[1] Philip Samuel Abrams. 1970. *An APL machine.* Ph.D. Dissertation. Stanford University, Stanford, CA, USA.

[2] Eva Burrows, Helmer André Friis, and Magne Haveraaen. 2018. An Array API for Finite Difference Methods. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2018).* ACM, New York, NY, USA, 59–66. https://doi.org/10.1145/3219753.3219761

[3] Benjamin Chetioui, Lenore Mullin, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Sandra Macià. 2019. Finite Difference Methods Fengshui: Alignment through a Mathematics of Arrays. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY 2019).* Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/3315454.3329954

[4] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018).* ACM, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[5] Harry B. Hunt III, Lenore R. Mullin, Daniel J. Rosenkrantz, and James E. Raynolds. 2008. A Transformation–Based Approach for the Design of Parallel/Distributed Scientific Software: the FFT. *CoRR* abs/0811.2535 (2008). arXiv:0811.2535 http://arxiv.org/abs/0811.2535

[6] K. E. Iverson. 1962. *A Programming Language.* Wiley, New York.

[7] Daniel Lemire, Owen Kaser, and Nathan Kurz. 2019. Faster remainder by direct computation: Applications to compilers and software libraries. *Softw., Pract. Exper.* 49, 6 (2019), 953–970. https://doi.org/10.1002/spe.2689

[8] Lenore Mullin. 1988. *A Mathematics of Arrays.* Ph.D. Dissertation. Syracuse University.

[9] Lenore M. R. Mullin and Michael A. Jenkins. 1996. Effective data parallel computation using the Psi calculus. *Concurrency - Practice and Experience* 8, 7 (1996), 499–515. https://doi.org/10.1002/(SICI)1096-9128(199609)8:7<499::AID-CPE230>3.0.CO;2-1

[10] Paul Chang and Lenore R. Mullin. 2002. An Optimized QR Factorization Algorithm based on a Calculus of Indexing. DOI: 10.13140/2.1.4938.2722.

Glossary

API (Application Programming Interface) In a programming language setting this typically means the types and operations on types exposed as some logical unit. 3, 4

APL (A programming Language) a dynamically typed programming language developed in the 1960s by Kenneth E. Iverson centered around the multidimensional array as its fundamental data type and operators on it. 38

BLAS (Basic Linear Algebra Subprograms) is a specification of a set of low-level routines for performing linear algebra operations for which there exists many implementations, among others a reference implementation in Fortran.. 24, 167

BLDL BLDL Bergen Language Design Laboratory. 4

confluent confluent In term rewriting systems a set of rules for rewriting terms are said to be confluent if, when different chains of rewrites may apply, making different choices does not yield a different final term. 14

DNF DNF Denotational Normal Form. 81, 137, 139

DRY (Don't Repeat Yourself) a principle of software engineering promoting the removal of needless repetition. 3

DSL (Domain Specific Language) A language tailored specifically for a certain domain. 82, 93, 94

ET (Expression Template) A way of building abstract intermediary expressions with template metaprogramming in C++. 7, 8, 12, 15

GHC (Glasgow Haskell Compiler) A popular Haskell compiler. 12–15

GPU (Graphics Processing Unit) a highly parallel, in terms of operations, processing unit typically found in commodity graphics cards and initiallyintended used for vector operations in graphics and image processing but suited for any parallel workload operating on large blocks of vectors. 93, 94, 115

inlining inlining the substitution of a function body for function call, removing call overhead but potentially adding to compiled code size. 12, 13, 15

ISA (Instuction Set Architecture) An abstract model of a computer describing an instruction set, data types, what state there is and the effects of executing instructions in terms state change. 8

JIT (Just In Time Compiler) A just in time compiler is a run-time compiler. It compiles code during the execution of a program and can take advantage of run-time information(such as hot paths) that is only available during program execution. 83

LAPACK (Linear Algebra Package) is library for numerical linear algebra, initially released in 1992 in Fortran 77 and moved to Fortran 90 2018, usually implemented in terms of the routines BLAS provides. 24

MOA (A Mathemathics of Arrays) an algebra describing a view of multidimensional arrays, in particular indexing them in relation to operations over them. 38, 45, 49, 62–64, 66, 70, 76, 79–81, 101, 114, 120, 121, 136, 138, 139, 152

NUMA NUMA Non-uniform memory access. 129

NumPy NumPy a library for numerical analysis written in Python. 92

ONF ONF Operational Normal Form. 81, 137, 139

OpenCL OpenCL is a framework for writing programs that run on a variety of processors, among them the heavily parallell GPUs found on commodity graphics cards. 93, 94, 96, 98

OpenMP (Open Multi-Processing) A set of language extensions for e.g. C, C++, Fortran that eases the task of paralell programming by providing simple annotations that instruct how to parallelize code. 130

PDE PDE Partial Differential Equation. 114

PE (Partial Evaluation) the fixing of values in a function and evaluating the function to yield a new function, a trivial example is fixing n=1 in divide(m,n) transforming divide into the identity function. In a programming context such a transformation may transform a function call into a value as its result is fully determined by fixing one parameter. 13

SIMD (Single Instruction Multiple Data) Typically a set of CPU instructions, some for loading N machine words from memory in one instruction into a special purpose register, some for operations between such special purpose registers and some for storing these special purpose registers back into memory. The purpose being data-level parallelism, operating in a single instruction on vectors of machine words rather than several instructions for each machine word. 8

SPARK/ADA SPARK/ADA a programming language that is a subset of ADA extended with various additional abilities aiding in program verficiation. 3

terminating terminating In term rewriting systems a set of rules for rewriting terms are said to be terminating if no infinite chain of rewrites is possible. 14

TMP (Template Metaprogramming) a metaprogramming technique involving compile-time filling of templates(code with holes), the generation of which may involve a fully Turing complete language as in the case of C++. 7

x86 x86 a family of instruction set architectures. 8

x86-64 x86-64 a family of instruction set architectures that includes 64-bit modes of operation. 28

Bibliography

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge university press, 1999.

[2] Anya Helene Bagge. “Constructs & Concepts: Language Design for Flexibility and Reliability”. PhD thesis. PB 7803, 5020 Bergen, Norway: Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway, 2009. ISBN: 978-82-308-0887-0. URL: `http://bldl.ii.uib.no/phd/bagge-phd-web.pdf`.

[3] Anya Helene Bagge and Magne Haveraaen. “Specification of generic APIs, or: why algebraic may be better than pre/post”. In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*. Ed. by Michael Feldman and S. Tucker Taft. ACM, 2014, pp. 71–80. ISBN: 978-1-4503-3217-0. DOI: `10.1145/2663171.2663183`. URL: `https://doi.org/10.1145/2663171.2663183`.

[4] Ulrich Drepper. “What every programmer should know about memory”. In: *Red Hat, Inc* 11 (2007), p. 2007.

[5] *Glasgow Haskell Compiler User’s Guide: 13.33. Rewrite rules.* URL: `https://downloads.haskell.org/~ghc/8.6.5/docs/html/users_guide/index.html` (visited on 2019).

[6] Bastian Hagedorn et al. “High performance stencil code generation with lift”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018.* Ed. by Jens Knoop et al. ACM, 2018,

pp. 100–112. DOI: 10.1145/3168824. URL: https://doi.org/10.1145/3168824.

[7] Marco Heisig and Harald Köstler. "Petalisp: run time code generation for operations on strided arrays". In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI 2018, Philadelphia, PA, USA, June 19, 2018*. Ed. by Sven-Bodo Scholz and Olin Shivers. ACM, 2018, pp. 11–17. DOI: 10.1145/3219753.3219755. URL: https://doi.org/10.1145/3219753.3219755.

[8] Christopher Hollings. "Partial actions of monoids". In: *Semigroup Forum*. Vol. 75. 2. Springer. 2007, pp. 293–316.

[9] Klaus Iglberger et al. "Expression Templates Revisited: A Performance Analysis of Current Methodologies". In: *SIAM J. Scientific Computing* 34.2 (2012). DOI: 10.1137/110830125. URL: https://doi.org/10.1137/110830125.

[10] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf (visited on 2019).

[11] *International standard ISO / IEC 9899 Programming languages C - reference number ISO/IEC 9899:1999(E), Second Edition 1999-12-01*. ISO, 1999.

[12] *Latency Numbers Every Programmer Should Know*. URL: https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html (visited on 2019).

[13] *Linear Algebra library Eigen*. URL: http://eigen.tuxfamily.org (visited on 2019).

[14] *Mouldable programming Magnolia*. URL: https://bldl.ii.uib.no/dmpl.html (visited on 2019).

[15] Lenore Mullin. "A Mathematics of Arrays". PhD thesis. Dec. 1988.

[16] Lenore M Mullin and James E Raynolds. "Tensors and nd Arrays: A Mathematics of Arrays (MoA), psi-Calculus and the Composition of Tensor and Array Operations". In: *arXiv preprint arXiv:0907.0796* (2009).

[17] Lenore Mullin and M Jenkins. "Effective data parallel computation using the Psi calculus". In: *Concurrency: Practice and Experience* 8 (Sept. 1996). DOI: 10.1002/(SICI)1096-9128(199609)8:7<499::AID-CPE230>3.0.CO;2-1.

[18] Lenore Mullin and Scott Thibault. "A Reduction semantics for array expressions: the PSI compiler". In: *Department of Computer Science, University of Missouri-Rolla, Rolla, Missouri* 65401 (1994).

[19] *P0542R5 Support for contract based programming in C++*. URL: `https://isocpp.org/files/papers/p0542r5.html` (visited on 2019).

[20] *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. URL: `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf` (visited on 2019).

[21] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. "Playing by the rules: rewriting as a practical optimisation technique in GHC". In: *2001 Haskell Workshop*. ACM SIGPLAN. Sept. 2001. URL: `https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/`.

[22] Colin Runciman. "What About the Natural Numbers?" In: *Comput. Lang.* 14.3 (1989), pp. 181–191. DOI: `10.1016/0096-0551(89)90004-0`. URL: `https://doi.org/10.1016/0096-0551(89)90004-0`.

[23] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a functional data-parallel IR for high-performance GPU code generation". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. Ed. by Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang. ACM, 2017, pp. 74–85. ISBN: 978-1-5090-4931-8. URL: `http://dl.acm.org/citation.cfm?id=3049841`.

[24] Michel Steuwer et al. "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 205–217. ISBN: 978-1-4503-3669-7. DOI: `10.1145/2784731.2784754`. URL: `https://doi.org/10.1145/2784731.2784754`.

[25] *The Java Language Specification: 15.18.1. String Concatenation Operator +*. URL: `https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.18.1` (visited on 2019).

[26] *The Java Language Specification: 15.18.2. Additive Operators (+ and -) for Numeric Types*. URL: `https://docs.oracle.com/javase/specs/jls/se13/html/jls-15.html#jls-15.18.2` (visited on 2019).

[27] Divakar Viswanath. *Scientific Programming and Computer Architecture*. MIT Press, 2017.