A Rosetta Stone for Array Languages

Artjoms Šinkarovs
Heriot-Watt University
Edinburgh, Scotland, UK
a.sinkarovs@hw.ac.uk

Robert Bernecky
Snake Island Research Inc
Toronto, Ontario, Canada
bernecky@snakeisland.com

Hans-Nikolai Vießmann
Heriot-Watt University
Edinburgh, Scotland, UK
hv15@hw.ac.uk

Sven-Bodo Scholz
Heriot-Watt University
Edinburgh, Scotland, UK
s.scholz@hw.ac.uk

Abstract

This paper aims to foster cross-fertilisation between programming language and compiler research performed on different array programming language infrastructures. We study how to enable better comparability of concepts and techniques by looking into generic translations between array languages. Our approach is based on the idea of a basic core language Heh which only captures the absolute essentials of array languages: multi-dimensional arrays and shape-invariant operations on them. Subsequently, we investigate how to map these constructs into several existing languages: SaC, APL, Julia, Python, and C. This approach provides us with some first understanding on how the peculiarities of these languages affect their suitability for expressing the basic building-blocks of array languages. We show that the existing tool-chains by-and-large are very sensitive to the way code is specified. Furthermore, we expose several fundamental programming patterns where optimisations missing in one or the other tool chain inhibit fair comparisons and, with it, cross-fertilisation.

CCS Concepts • **Software and its engineering → General programming languages**; *Functional languages*;

Keywords functional languages, array languages, performance portability

ACM Reference Format:
Artjoms Šinkarovs, Robert Bernecky, Hans-Nikolai Vießmann, and Sven-Bodo Scholz. 2018. A Rosetta Stone for Array Languages. In *Proceedings of 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219753.3219754

ARRAY'18, June 19, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5852-1/18/06.
https://doi.org/10.1145/3219753.3219754

1 Introduction

Over time, several different languages with a focus on array processing have been defined and implemented. These reach from fairly long-standing languages such as APL or Fortran to more recent additions such as Julia or Futhark. While all these languages differ in their syntax and the exact set of built-in operations, they do share the core setup: they support a data structure for representing multi-dimensional arrays, and they support a set of data-parallel map-/reduce-like array manipulation operations. Despite this similarity, cross-fertilisation between the languages and their implementations seems to be happening at a slow pace. Advanced implementation and optimisation techniques, runtime solutions and example codes developed for one language may never find their way into other array languages.

Transferring ideas from one setup to another often fails due to a perceived language barrier. This paper explores how to eliminate this barrier by looking into mechanical translations between different array languages while preserving the nature of array-based programs.

While the similarity in their core constructs should render a translation simple, in practice, it often turns out to be non-trivial: most languages offer several ways to translate the same program. These programs generate the same results, yet their runtimes may be orders of magnitude apart.

If we envision translation T as a function from any array language to any other array language:

$$T : \mathbb{X} \to \mathbb{X} \qquad \mathbb{X} = \{X_1, \ldots, X_n\}$$

it would be reasonable to avoid the need to create n^2 translators. We can do this by adopting L, an intermediate language for T:

$$T = LT \circ TL \qquad TL : \mathbb{X} \to \{L\} \qquad LT : \{L\} \to \mathbb{X}$$

T can be seen as a composition of the TL function that translates any language to L and the LT function that translates L to any other language. This reduces the number of translators from n^2 to $2n$, and given that L exists, can be seen as a universal representation of array-based programs.

L, itself, seems like a powerful idea, as it would simplify extensive benchmarking of array programs, and can be seen

as a way to share program transformations among compilers, *e.g.* translating X to L and back to X can be seen as running shared optimisations on X. If so, what is the best choice for L?

Contributions We propose such an L and focusing on the LT function, we implementation translations from L to APL, SaC, Python, Julia, C. We ensure that all the translations support shape invariance, we evaluate performance of our translations on three micro benchmarks and investigate causes of poor performance. This study offers a first approximation of the interplay among desired features of L, and language features of target compilers that LT can rely on.

2 Intermediate Language

We use a minimalist intermediate functional language with native support for multi-dimensional arrays. Our translations are straight-forward which puts some stress on the underlying compilers/interpreters, helping us to identify essential features that have to be included so that performance portability is preserved. Our language is strict, as most array-based languages are strict. For simplicity, we avoid type system aspects, assuming that the only element types that exists in our language are `Nat` and `Bool`, representing natural numbers and booleans, and that we can derive arrays of natural numbers or booleans and function types in the usual way. We do not envision a practical intermediate language to be untyped, yet type systems offer little help when translating expressions where array ranks and shapes are only available at runtime, a main concern of our study.

Arrays are treated as mappings from indices to values, where the set of indices is bound by a shape and all the values are of the same type. We chose hyper-rectangular index spaces, meaning that the shape of any array can be described by a tuple of natural numbers, where each element represent the extent (length) of the corresponding axis. For example, we describe the shape of a matrix of m rows and n columns by a tuple $\langle m, n\rangle$.

We implement[1] these concepts in a strict version of the Heh programming language [18], an applied λ-calculus, with the syntax shown in Fig. 1.

Constants are boolean or natural numbers; the usual arithmetic functions and comparison operations are built in; conditional expressions are included; let bindings are always recursive.

Arrays can be constructed as a sequence of expressions in square brackets. For example, $[1, 2, 3, 4]$ is a four-element vector, while $[[1, 2], [3, 4]]$ is a two-by-two-element matrix. Inhomogeneous arrays are not supported, so the term $[[1, 2], [3]]$ is irreducible.

Array selection, denoted by a dot symbol, is used as an infix binary operator between an array to select from a valid

[1]Source code, documentation, and examples are available at https://github.com/ashinkarov/heh.

$$
\begin{array}{lcll}
e & ::= & x \mid \lambda x.e \mid e\ e & \text{(core } \lambda\text{-calculus)} \\
 & \mid & \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e & \text{(conditionals)} \\
 & \mid & \mathit{letrec}\ x = e\ \mathit{in}\ e & \text{(recursive let)} \\
 & \mid & c \mid e + e \mid \ldots & \text{(consts/binary ops)} \\
 & \mid & [e, \ldots, e] & \text{(array constructor)} \\
 & \mid & e.e & \text{(array selection)} \\
 & \mid & |e| & \text{(shape operation)} \\
 & \mid & \mathit{reduce}\ e\ e\ e & \text{(reduction)} \\
 & \mid & \mathit{imap}\ s \begin{cases} g_1 : & e_1, \\ \quad \ldots & \\ g_n : & e_n \end{cases} & \text{(index map)} \\
\\
s & ::= & e & \text{(scalar imap)} \\
 & \mid & e|e & \text{(generic imap)} \\
g & ::= & e <= x < e & \text{(index set)} \\
 & \mid & \_(x) & \text{(full index set)}
\end{array}
$$

Figure 1. Syntax of Heh

index into that array. A valid index is a vector containing as many elements as the array has dimensions; otherwise it is undefined.

$$[1, 2, 3, 4].[0] = 1 \quad [[1, 2], [3, 4]].[1, 1] = 4 \quad [[1, 2], [3, 4]].[1] = \bot$$

An array shape can be found with the primitive *shape* operation, denoted by enclosing vertical bars. It is applicable to arbitrary expressions, and returns the shape of its argument as a vector:

$$\left|[1, 2]\right| = [2] \quad \left|[[]]\right| = [1, 0] \quad \left|\mathit{true}\right| = [] \quad \left|42\right| = [] \quad \left|\lambda x.x\right| = []$$

We expressed reduction operations in Heh using the *reduce* combinator, a variant of *foldl*, extended to support multi-dimensional arrays, instead of lists. *reduce* takes three arguments: the binary function, the neutral element and the array to reduce. For example, assuming row-major order, we have:

$$\mathit{reduce}\ (+)\ 0\ [[1, 2], [3, 4]] = ((((0 + 1) + 2) + 3) + 4)$$

The key construct of Heh is *imap*, which restricts the domain of an index-value-mapping function f to shape s. For example, the index domain here is restricted by the shape $[5]$:

```
imap [5] {_(iv):1
```

describing indices $\{[0], [1], [2], [3], [4]\}$. This maps every index to the constant 1, which evaluates to $[1, 1, 1, 1, 1]$.

We specify advanced index mappings via partitions, using the following syntax:

```
imap [5] {[0] <= iv < [3]: iv.[0] + 10,
          [3] <= iv < [5]: iv.[0] + 20
```

This breaks our index space into two partitions, one for indices {[0], [1], [2]}, and the other for indices {[3], [4]}. The `iv` variable binds to each value within the corresponding index-space, and the overall expression evaluates to [10, 11, 12, 23, 24].

When index mapping within the *imap* evaluates non-scalar results (objects of non-empty shape), we require a programmer to specify that shape using the following syntax:

```
imap [3]|[2] {_(iv): [1,2]
```

which evaluates to the array [[1, 2], [1, 2], [1, 2]], of shape [3, 2].

2.1 Compilation

In the rest of this paper we explore translations of programs in this language to the following target languages: SaC, APL, Python, Julia, C, Futhark. We start with the following assumptions.

Higher-order functions Translation starts with function lifting and defunctionalisation, using normal methods. Many programs can be rewritten as a set of first-order (potentially recursive) functions of multiple arguments. Here, we ignore cases when this fails, or when partial applications have not been eliminated.

Arrays of functions We ignore Heh programs that use arrays of functions, because that support requires runtime function closure representations that are absent from some of our target languages.

Normalising imap We treat scalar *imap*s and the `_(iv)` syntax as syntactic sugar, eliminated during parsing, using these rewrites[2]:

$$imap\ s\ \{\ p_1, \ldots, p_n \rightsquigarrow imap\ s|[]\ \{\ p_1, \ldots, p_n$$

$$imap\ s_1|s_2\ \{\ \_(iv) : e \rightsquigarrow imap\ s_1|s_2\ \{\ (zz\ s_1) <= iv < s_1 : e$$

where `zz` is a function that generates a 1-dimensional array of zeroes of shape identical to the shape of s_1, defined as:

```
letrec zz = λs.imap |s| {[0] <= iv < |s|: 0
```

Experiments For all the backends we verify shape-invariant functions are expressible. For example, consider the following increment function:

```
letrec inc = λa.imap |a| {_(iv): a.iv + 1
```

It is shape invariant, since we did not specify the shape of *a*. We can apply `inc` to any array of natural numbers, including any natural number, as in Heh, numbers are zero-dimensional arrays.

We verify that empty arrays are expressible. Like APL, Heh makes it possible to generate an infinite number of empty arrays. An empty array is described by the shape that contains at least one zero. For example:

[2] To simplify presentation, we assume that s_1 is a variable, therefore rewrites do not introduce code duplication.

```
imap [1,0,5] {_(iv): 1
```

evaluates to an empty array of shape [1, 0, 5].

We verify that reductions on empty arrays, arrays containing a single element and regular arrays succeed.

We test performance of the generated code by running matrix addition and two versions of matrix multiplication:

```
letrec matplus = λa.λb.
    imap |a| {_(iv): a.iv + b.iv

letrec matmul1 = λa.λb.
    imap |a| {_(iv):
        letrec i = iv.[0] in
        letrec j = iv.[1] in
        letrec n = |a|.[0] in
        reduce (λx.λy.x+y) 0
               imap [n] {_(jv):
                   letrec k = jv.[0] in
                   a.[i, k] * b.[k, j]

letrec sum = λa.λb.λn.λi.λj.λk.λres.
    if k = n then res
    else sum a b n i j (k+1)
         (res + a.[i,k] * b.[k,j]) in

letrec matmul2 = λa.λb.
    imap |a| {_(iv): letrec i = iv.[0] in
                     letrec j = iv.[1] in
                     letrec n = |a|.[0] in
                     sum a b n i j 0 0
```

The first version uses a *reduce* operation to sum the result of multiplication of *i*-th row by *j*-th column; the second version implements sum as a tail-recursive function.

Our main concern is to estimate the sanity of our translation, rather than to optimise for best performance on a given system. We use two possible shapes of input arrays: 1000×1000, and 100×100 for the cases when the generated code is unreasonably slow for the former shapes.

Finally, we briefly discuss what would it take to extend each backend to support higher-order functions, partial applications and arrays of functions.

Setup We ran all experiments on an 'Intel (R) Core (TM) i7–6700HQ CPU @ 2.60GHz' CPU, under Gentoo Linux, GCC version 7.2.0, Python version 3.5.5, PyPy version 5.10.0, sac2c version 1.2, Julia version 0.6.2, and Dyalog APL version 16.0.32742. All the experiments ran on a single core in a sequential fashion.

3 Heh to SaC

Heh and SaC are closely related: Heh can be seen as SaC, generalized with higher-order functions, lambdas and arrays of functions. Both languages support multi-dimensional arrays the same way. The key data-parallel construct in SaC is called the *with-loop* [17]. A *with*-loop makes it possible to define index-value mapping in the same way as partitions in *imap*. However, a *with*-loop also provides a syntax that can perform a fold operation, using a two-argument function and a neutral element, as in this array generator:

```
// imap [5] {_(iv): 1
```

```
with {
  (. <= iv <= .): 1
}: genarray ([5], 0);
```

where `(. <= iv <= .)` is the syntax for the entire partition *e.g.* `_(iv)` in Heh. The `genarray` construct takes two arguments: the shape and a default element. The latter is needed when partitions of the *with*-loop do not cover the entire index space. In Heh, this would be a runtime error.

For non-scalar *imap*s with shapes s_1 and s_2, we have to generate the default element of shape s_2. Here is how we might do this:

```
// imap [5]|[3] {_(iv): [1,2,3]
s1 = [5]; s2 = [3];
with {
  (0*s1 <= iv < s1): [1,2,3];
}: genarray (s1, zero (s2));
```

where the function `zero` is defined as follows:

```
// Array of zeroes of shape s
int[*] zeroes (int[.] s)
{
  return with{}: genarray (s, 0);
}
```

We can express reductions in SaC as follows:

```
/* reduce (λx.λy.x+y) 0 [1,2,3,4,5] */
a = [1,2,3,4,5];
with {
  ([0] <= iv < [5]): a[iv];
}: fold (+, 0);
```

where `fold` has two arguments: the binary function and the neutral element. The above expression combines all the results produced by the mappings using the binary function and a neutral element; the latter is useful when the index set is empty or contains a single element.

SaC requires all functions to specify their argument and return types. The type system of SaC allows type declaration for arrays of any shape, with a fixed element-type, so we make all functions of type integer-array of any shape. For example:

```
// letrec plus = λx.λy.
//     imap |x| {_(iv): x.iv + y.iv
inline int[*] plus (int[*] x, int[*] y)
{
  s = shape (x);
  return with {
           (0*s <= iv < s): x[iv] + y[iv];
         }: genarray (s, 0);
}
```

The `shape` function is built in, working just as `|x|` does in Heh, being applicable to arrays of all shapes, including scalars. SaC specializes functions with known array ranks, often producing better performance. SaC uses heuristics to decide whether a function is suitable for inlining. We mark all generated functions with *inline*.

For more translation details, including its OCaml implementation, refer to https://github.com/ashinkarov/heh.

Evaluation Due to Heh and SaC similarities, we can easily express shape-invariant functions, *imap*s, non-scalar *imap*s and reductions. Runtime results in seconds are presented in Table 1.

We observed reasonable performance for *matplus* and *matmul1*. The SaC compiler fused reductions over *imap* into a single operator, avoiding unnecessary memory allocation and memory subsystem traffic. When *inline* annotations are not present, *matmul1* runs in about 1.4 seconds. This suggests that current inlining heuristics are not optimal at least for the given use case.

Benchmark	Time (*s*)
matplus 1000×1000	0.01
matmul1 1000×1000	0.58
matmul2 1000×1000	53

Table 1. SaC runtimes

The second version of matrix multiplication performs less well, suggesting that SaC does not optimise tail-recursive functions into loops. A manual rewrite of the function-call with a for-loop construct reduces the runtime to 0.6 seconds.

Extensions Support for higher-order functions and function arrays in SaC would require extensions to the SaC compiler, as there is no obvious way to emulate those constructs.

4 Heh to APL

APL[3] provides a rich set of array operations that we used to implement Heh primitives. APL arrays are first class, and a built-in shape function, `⍴ a`, gives the shape of the array *a*. Selections from array *a* at the index vector *iv* are expressed as: `iv ⌷ a`. Constants, scalar operations and array constructors are also built in.

APL has unusual treatment of functions: all the functions must be either binary or unary; higher-order functions are supported only if they are of order two. We chose to implement all our translation functions as unary ones. We pass multiple arguments as a nested-array argument, splitting them into components within the function. As an example, consider our translation of the scalar addition function $\lambda x.\lambda y.x+y$, where ω is a formal parameter to f that we bind to local variables x and y:

```
f ← {
  (x y) ← ⍵
  x + y
}
```

We used *guards*, similar to *return* statements in imperative programs, to represent Heh conditionals. The `p:e` expression returns the result of *e* if the predicate *p* is true. We translated a Heh conditional such as **if** `2 > 3` **then** `f a` **else** `g b` as a function with local functions for each leg of the original conditional, then use the guard expression to select the leg to execute:

[3] We use APL in the form of John Scholes' Dfns[10], a functional subset of Dyalog APL[9].

```
condf ← {
  tr_br ← { f a }
  fl_br ← { g b }
  ⍵ : tr_br 0
  fl_br 0
}
condf (2 > 3)
```

APL has no *imap* operation, as most APL primitives operate on the values of entire arrays, rarely requiring scalar functions and array selections as building blocks. We implemented *imap* as array updates to an all-zero array. For each *imap* partition, we generated an index array using this function:

```
indexset ← {(⊂⍺)+⍳⍺−⍵}
```

e.g. the *imap* partition index set for `[2,3] <= x < [4,6]` is a nested array of shape 2 3, each element of which is a 2-element vector:

```
 2 3 indexset 4 6
2 3  2 4  2 5
3 3  3 4  3 5
```

Finally, we use the helper function to implement the update of the array `res` for indices from `lb` to `ub`:

```
imap ← {
  res ← ⍺ ⋄ (lb ub) ← ⍵
  iv ← lb indexset ub
  empty ← 0≡×/⍴iv
  empty : res
  vals ← ⍺⍺ ¨¨ iv
  vals @iv ⊢ res
}
```

the ¨¨ (double dot) operation applies the function containing the body of the *imap* partition to every index within that partition; and the last line updates the `res` array. We had to introduce a special case for empty partitions due to the chosen semantics of the double dot operator in APL: if right-hand operand of ¨¨ is an empty array, the left operand is called with 0. The above function is used for both scalar- and non-scalar imaps.

As an example, consider the increment function generated from the $\lambda a.imap\ |a|$ {_(iv): a.iv + 1 expression:

```
inc ← {
  s ← ⍵
  f ← { 1 + (⍵ ⌷ s) }
  ss ← (⍴s)⍴0
  ⊃(ss f imap (0×(⍴s)) (⍴s))
}
```

APL offers built-in reduction operations; however, as reductions over empty arrays and arrays containing a single element are not well-defined for generic functions, we introduced special cases in our translation scheme. Further details including the OCaml code can be found[4] at the *apl-backend* branch of the Heh repository.

[4] Available at https://github.com/ashinkarov/heh/tree/apl-backend.

Evaluation We summarise the runtime of the generated codes for our benchmarks, including native APL matrix multiplication in seconds in Table 2.

Heh generates scalar-oriented code, and experienced APL programmers know that the appearance of index vectors in source code is almost always a bad omen; that is the case here.

In APL interpreters, all values are arrays, with descriptors specifying type, shape, and value. Even trivial expressions such as `iv+1` require memory allocation and initialization for results, so APL arrays, including index vectors, are heap-allocated, with concomitant performance loss due to memory management overheads.

Benchmark	Time (*s*)
matplus 1000 × 1000	4.2
matmul1 100 × 100	8.0
matmul2 100 × 100	8.3
`x+.×y` 1000 × 1000	2.3

Table 2. APL runtimes

With nested arrays, those overheads are multiplied by the number of elements in each array, because each nested element requires its own descriptor. Thus our code using nested arrays runs about 500× slower than a hand-coded APL equivalent.

The Heh-generated `imap` function generates an explicit index set for the array, as nested vectors, then performs an ¨¨ using that index set as an argument. Replacing the above expression with `f⍤¨0⊢s` [4] produced a more than three-fold speedup, much better, but still short of native code performance.

The brief array lifespan in APL means that runtime attempts to turn scalar-oriented APL into array operations are usually a net loss, in terms of overall application performance [2]. For this reason, APL implementations do not optimize, relying instead on special cases for some primitives and compositions thereof.

Recent JIT work by Foad [8] reduces some interpreter overhead, but does not support transformations such as turning an *imap*-based scalar function into an APL array expression, such as `x+y`.

Extensions Like SaC, today's APLs lack ways to emulate support for general higher-order functions or function arrays [1, 3].

5 Heh to Python/Julia

Translation to the Python and Julia languages is similar, as both languages provide similar abstractions. Python is a dynamic language that does not focus on arrays, but the NumPy [14] module offers native support for multi-dimensional arrays. It is touted as a package for scientific computing, so we chose to translate Heh to NumPy. Similarly, Julia [5] is claimed to be a "high-level, high-performance dynamic programming language for numerical computing" that treats multi-dimensional arrays as first-class citizens.

5.1 NumPy

The representation of multi-dimensional arrays in NumPy is similar to those found in SaC or Heh: an object with a shape vector (a tuple of length n), and a value array that can be indexed by tuples of n integers. Consider the following example:

```
# letrec a = [[1,2],[3,4]] in a.[0,1]
a = np.array ([[1,2],[3,4]])
a[(0,1)]
```

If we use `np.array` to construct an array of shape $(2,2)$ from a nested list, then select element $(0,1)$ from it, we obtain 2.

For the purposes of our translation, we need only a few of the NumPy-provided operations in APL, such as shape and rank, called `shape` and `dim`. For the array `a` from above:

```
a.shape # evaluates to (2,2)
a.dim   # evaluates to 2
```

To iterate over an array, NumPy provides the `nditer` construct, which creates an iterator over all valid indices of the argument array. For example, we can implement *reduce* as follows:

```
# reduce f neut a
res = neut
for x in np.nditer (a):
    res = f (res, x)
```

NumPy provides a function that *almost* implements the *imap* construct. Consider an example:

```
np.fromfunction (lambda i,j: i+j, (2,2),
                 dtype=int)
```

The higher-order function `fromfunction` has three arguments: a function from indices to values, the shape and the element type. The function is called for every index defined by the shape. The above expression evaluates to the array: $[[0,1],[1,2]]$.

Any multi-partition *imap* can be rewritten as a single-partition *imap* by using a conditional. This was our first implementation of *imap*. We have inlined the implementation of `fromfunction` and we defined a generic function `heh_imap`:

```
def heh_imap (s1, s2, f):
    res_sh = tuple (list (s1) + list (s2))
    res = np.empty (res_sh, dtype=int)
    for idx in np.ndindex (*tuple (s1)):
        res[idx] = f (np.array (idx))
    return res
```

We compute the result shape by concatenating shapes `s1` and `s2` (we also type-cast the array into a tuple), then create an empty array of that shape using `nd.empty`. This function creates a placeholder for that array, but does not set values. Then, we iterate over the index-range `s1`, at every index assigning the result of applying the function f to that index. This works correctly for non-scalar results, because partial selections are natively supported by NumPy.

After that, we generate a local function that chooses the right partition for this index, then executes the relevant code:

```
# imap [4] {[0] <= iv < [2]: 1,
#           [2] <= iv < [4]: 2
def f (iv):
    if (heh_inrange (iv, lb1, ub1)):
        return np.array (1)
    if (heh_inrange (iv, lb2, ub2)):
        return np.array (2)
    return None

heh_imap (np.array ([4]), np.array ([]), f)
```

Then, we implemented an alternative translation similar to that in APL and C. For every partition with a given *imap*, we generate a loop that iterates from lower bound to upper bound defined by the partition and updates elements of the result array:

```
# imap [4] {[0] <= iv < [2]: 1,
#           [2] <= iv < [4]: 2
s = np.array ([4])
t1 = np.array ([2])
t2 = np.array ([0])
res = np.empty (tuple ((list (s) + list ([]))),
      dtype=int)
for idx in (np.ndindex (*tuple ((t1 - t2)))):
    iv = (idx + t2)
    res[tuple (iv)] = np.array (1)
for idx in (np.ndindex (*tuple ((s - t1)))):
    iv = (idx + t1)
    res[tuple (iv)] = np.array (2)
```

Details on Heh to Python translation, including OCaml code, can be found in the *numpy-backend* branch[5] of the Heh repository.

Evaluation By default, shape-invariant functions do not work in NumPy. Zero-dimensional arrays are supported by NumPy, but scalar values are not promoted to zero-dimensional arrays, so selection from a scalar produces a runtime error. To solve this problem, we promote scalars to zero-dimensional arrays, whenever we generate a number, and at every selection. For example, the `2.[]` Heh expression will be translated as:

```
x_1 = np.array (2)
x_2 = np.array ([])
np.array (x_1[tuple (x_2)])
```

NumPy runs under both the Python interpreter CPython and PyPy [16] — a fast alternative implementation of Python, using Just-in-Time compilation.

The following table gives runtimes for these, in seconds; we use the *h*-prefix when *imap*s are translated using higher-order functions.

Benchmark	*h*CPython	*h*PyPy	CPython	PyPy
matplus 1000×1000	27	11	13.3	7.8
matmul1 100×100	18	12	12.1	11.1
matmul2 100×100	6.1	7.9	5.7	7.6

Performance is poor even on such trivial benchmarks. PyPy helps in some cases, but runtimes are still poor.

If we manually encode matrix multiplication in NumPy, using a built-in operation, we get 1.7 seconds for two $1000 \times$

[5] Available at https://github.com/ashinkarov/heh/tree/numpy-backend.

1000 matrices from CPython, and `NotImplementedError` from PyPy.

Extensions Higher-order functions, partial applications and arrays of functions are supported by Python/NumPy already, and extending the proposed translation scheme is straightforward.

5.2 Julia

Like Python, Julia is a dynamically typed programming language and interpreter. Its code generation is similar to Python, so we focus on areas where they differ. Julia supports multi-dimensional arrays as first-class objects with shape information:

```
# letrec a = [[1,2],[3,4]] in a.[0,1]
a = [1 2; 3 4]
a[1,2]
```

Julia uses origin 1 indices; also, arrays are stored in column-major order. Heh does not fix how arrays are mapped into the memory, so column-major representation can be used. This has performance implications if the source language uses a different mapping and fixes array traversal order. The difference in indexing is solved by incrementing all the index-vector elements by one at every selection. When we use built-in iterators to index an array, we decrement all the elements of the iterator by one.

Shape and rank information can be retrieved using the `size` and `ndims` functions, respectively. For example:

```
size(a)   # (2, 2)
ndims(a)  # 2
```

For array iteration, Julia provides a `eachindex` function, much like NumPy's `nditer` function, resulting in this `reduce` function:

```
# reduce f neut a
res = neut
for idx in eachindex(a)
    res = f(a[idx], res)
end
```

Implementation of the *imap* follows the approach we took in Python. First, we try the higher-order function approach:

```
# imap s1|s2 {_(iv): f iv
function heh_imap(s1::Array, s2::Array, f)
  res=Array{Array{Int, length(heh_arr2tup(s2))},
                       length(heh_arr2tup(s1))
        }(heh_arr2tup(s1))

  for idx in eachindex(res)
    res[idx] =
        f(heh_tup2arr(ind2sub(res,idx))-1)
  end
  flt = collect(Iterators.flatten(res))
  return reshape(flt, tuple(s1..., s2...))
end
```

As Julia does not support non-scalar updates natively, we create the nested array `res`; we iterate over the `s1`-generated index-space; finally we reshape `res` to the concatenation of `s1` and `s2`. Unfortunately, in Julia, reshape only works on the outermost array, and does not affect the shape of inner structures, therefore we first flatten `res` and then reshape it to the correct shape. Further complexity of the code has to do with the semantics of `eachindex` which generates linear scalar indices *e.g.* from 1 to $\prod s_1$ in our case. These numbers are valid indices into Julia arrays but they do not match the expectations of the Heh code.

We also implemented an inline *imap* solution, akin to that in Python. Partitions of the Heh *imap* are converted to inline for-loops, covering the index ranges as specified in the Heh code.

Further details of the Julia backend for Heh can be found in the *julia-backend* branch[6] of the Heh repository.

Evaluation We evaluate the performance of generated Julia code in Table 3 against the same benchmarks as Python; times are in seconds; the *h*-prefix indicates the use of higher-order functions.

Benchmark	*h*Julia	Julia
matplus 1000×1000	11.2	13.2
matmul1 100×100	17.4	16.6
matmul2 100×100	14.4	14.5

Table 3. Julia Runtimes (*s*)

Both of our *imap* variants have similar runtimes, and are comparable with the Python inlined-*imap* results. Yet, in absolute terms, this is still inadequate, suggesting that Julia's JIT facility is not strong enough to handle our generated code. A loop-based *imap* implementation did not outperform one using higher-order functions, perhaps due to unoptimized index vector arithmetic in Julia.

We implement native matrix multiplication of two 1000×1000 arrays using Julia built-in functions, and measure a runtime of 2.5 seconds.

Extensions Similarly to Python, higher-order functions and arrays of functions are supported natively.

6 Heh to C

We now look at translation for low-level languages, using C as typical of that class. The main differences, compared to our other backends, lie in C's need for explicit memory management and its weak support for multi-dimensional arrays.

Two simple ways to model multi-dimensional arrays in C are as a nested vector of pointers, or as a flat representation. Since a flat representation provides better performance, we used that.

C does not have support for multi-dimensional arrays in the Heh/SaC/APL sense, but it provides a convenient way to index flat multi-dimensional arrays. We might make an array as:

```
// stack-allocate a flat 5×6 array
int a[5][6];
// heap-allocate a flat 5×6 array
int (*b)[5][6] = malloc (5*6*sizeof(int));
```

[6] Available at https://github.com/ashinkarov/heh/tree/julia-backend.

We index these objects using square bracket syntax: `a[0][1]` and `(*b)[0][1]`. Internally, a sequence of square brackets are turned into a row-major linearised representation offset, and traversed using a loop nest. Array shapes and ranks must be programmer-managed.

The above representation leads to good performance, as loop nests over integer indices are recognised by a number of compiler optimisations, but it requires compile-time knowledge of all array ranks, and if we want to support shape-invariant functions, that knowledge may not be available, so we focus on the general case.

We define a structure to represent an array as a union of two structures, the first for scalars, the other for non-scalars:

```
struct value_scalar {
  enum value_kind kind; size_t val;
};
struct value_array {
  enum value_kind kind; size_t dim;
  size_t *shape; size_t *val;
};
union value {
  struct value_scalar val_scal;
  struct value_array val_arr;
};
```

We use a primitive memory model that directly follows the Heh code. All arrays are read-only; we always create a new array to store results. Hence, we always allocate a new array on the heap to store an *imap* or function result. The *value* unions are allocated on the stack and are passed by value as function arguments.

We implemented an *imap* expression, using a for-loop over a generic iterator. Below we show the structure of our translation:

```
// imap s1|s2 {l1 <= iv < u1: e1,
//                ...
value res = mk_array (vec_concat (s1, s2));
for (iter_t it = mk_iter (l1, u1); !it.done;
     it = next_iter (it)) {
  iv = it.idx; modarray (res, iv, e);
}
```

We allocate the array `res` to store the result of our *imap*. For every partition, we create an iterator object which internally keeps an index. The index is a vector of the same length as `s1`. At every iteration the loop we increment the index within the iterator until we reach the value `u1`, at which point the iterator is done. Also, at every iteration of the loop, we bind the variable `iv` to the current value of the iterator index. Then we evaluate the expression within the partition and we update `res` at the given index using `modarray` helper function. Note that `modarray` may involve some memory copying in case of non-scalar *imap*s.

We then promote scalar values to `value_scalar` type, and adjust arithmetic operations and selections so that the invariant holds.

We annotate all functions as *static inline*, and all arguments are annotated with *const*. For more details on translation and OCaml code, refer to the *c-backend* branch[7] of the Heh repository.

Evaluation With a C backend, we can express all the shape-invariant functions, non-scalar *imap*s and reductions.

We use the Boehm Garbage Collector [6], to ensure that local arrays are freed when a function goes out of local scope. Without this, the matrix multiplication would otherwise exhaust 16GB of memory in about 12 seconds.

Benchmark	Time (*s*)
matplus 1000×1000	0.03
vectorplus 1000000	0.03
matmul1 1000×1000	295
matmul2 1000×1000	246

Table 4. C Runtimes

The same runtimes for matrix and vector addition suggest that using iterators does not have substantial performance implications.

A *reduce* over an *imap* pattern is not optimized, but the major performance problem is that of repeated memory allocations, to store two-element index vectors $[i, k]$ and $[k, j]$. The C compiler leaves the allocations within the inner loop, with a detrimental effect on performance. If we manually rewrite the reduction, the program runs in about 2 seconds.

Extensions Higher-order function and function arrays support can be done by adding a closure representation, through a new value union member, although we have yet to implement that.

7 Heh to Futhark

Futhark [12] is touted as a "High-performance purely functional data-parallel array programming language"[8], having an array constructor identical to Heh and multi-dimensional array support.

```
[[1,2],[3,4]]
```

Futhark also provides a generic *reduce* operator identical to that in Heh. For example:

```
reduce (+) 0 [[1,2],[3,4]]
```

evaluates to 10. Futhark provides a *map* combinator that operates in a usual way on 1-dimensional arrays, so this expression evaluates to $[2, 3, 4]$.

```
map (\x -> x + 1) [1,2,3]
```

But, for multi-dimensional arrays, *map* iterates over only the outer dimension. So, incrementing all elements of a matrix would need:

```
map (\row -> map (\x -> x + 1) row) [1,2,3]
```

It in unclear how we might express a shape-invarient function in Futhark. One way would be to obtain an argument's shape, then recursively apply *map* until we get to vectors.

[7] Available at https://github.com/ashinkarov/heh/tree/c-backend.
[8] The official web-page https://futhark-lang.org/

The shape operator and recursive functions *are not* supported in Futhark. Access to shape components of an array must be done when defining function argument types, *e.g.*:

```
let sum_shape [m][n] (a: [m][n]i32) = m+n in
sum_shape [[1,2],[3,4]]
```

defines a function `sum_shape` that binds variables `m` and `n` to shape components of the function argument. That is, Futhark's shape is a type-level object that can be projected to values during function application. This permits expression of shape-polymorphic functions on fixed-rank arrays. In the above example, the `sum_shape` function operates only on rank-2 arrays.

Such restrictions pose challenges to our proposed translation. We could take the C route and generate all operations on a flat array representation, but this would undermine Futhark's design.

We can translate those Heh programs where all array ranks are known statically, rejecting the others, but this does not help with generic translation. How do we deal with the cases when the target language is strictly smaller than the intermediate language? Do we reject translations, adjust the intermediate language, or try to generate code even though it undermines the design of a language? This remains an open question.

8 Related Work

A typical intention of array intermediate languages is to capture a sufficiently precise representation of a single language, rather than a universal intermediate layer. In [11], the authors propose an intermediate typed language that can express a subset of APL. The type system captures array shapes, but it does not support operations where the shape of an expression depends on the shape of its arguments in a non-trivial way. The language introduces about 30 primitive operations, but it is not clear whether it is a complete or sufficient list to capture generic array programs.

Array languages like Qube [22] or Remora [19] use dependent types to capture array rank and shape dependencies, which require all programs to be type-checked statically; type inference in such systems may be challenging. An alternative approach, taken by SaC [17], uses a combination of sub-typing and intersection types, guaranteeing that type checking and type inference always terminate and can be always done statically. If typing information is imprecise, the compiler inserts runtime checks.

In [7, 15, 21], the authors explore specifications that capture algorithms and code transformation sequences that achieve good performance on certain hardware classes. Although this ability is useful in practice, it is not clear whether this is needed for a generic intermediate array language.

In [20], the authors introduce a generic array language and a set of rewrite rules for its core operators. Machine learning techniques make it possible to find a suitable sequence of rewrites that lead to the best performance. The authors show impressive performance results for a certain application class. However, the search space is incredibly large, so it is not clear whether our use of this approach could optimise generic translations for all backends of interest.

9 Conclusions and Future Work

We implemented translations of an array-based intermediate language into five supposedly universal array languages. We achieved semantically correct translation, but target compilers were fragile with respect to programming style: equivalent programs may have runtimes that are orders of magnitudes apart. This can be problematic when compilers are used in automated pipelines, as it undermines performance portability.

There are several actions that can be taken to mitigate this problem. For instance, we can treat a target compiler as a black box and improve the chosen translation scheme and/or the intermediate language. Or, we can require those compilers to recognise some array patterns. In the case of Python, Julia, and APL, it may be reasonable to recognise array expressions written as iterations over an index space. The SaC compiler would benefit from transforming tail-recursive functions into loops. Finally, C compilers could benefit from moving memory allocations around; unfortunately this is non-trivial in general. For some languages, improving memory reuse could be beneficial. This is non-trivial, but we can start with local optimisations, such as avoiding allocations within an *imap* body, as all data can be discarded at the end of every iteration.

It would also help to identify which *imap*s can be turned into generators. For high-level languages, code often can be left unmodified, as these languages typically apply advanced analysis already.

A proper type system would help to optimise special cases and improve the quality of translations. Despite many existing type systems for array languages, it is not clear what would an optimal choice that maximises static information, yet staying decidable.

Although improving compilers in the proposed ways is a very social/technical process, we believe that the goal to create a universal representation for array programs may stimulate such activity.

The question whether the design of an intermediate language that is based on data parallel combinators over scalar operations is a good choice remains open. An alternate approach might be to perform translation at a much higher semantic level, so that, for example, scalar functions on arrays are translated directly to the target language, without having to deal with *imap* and the like, echoing Mullin's early work [13]. Despite all these challenges, we believe that we got one step closer to the unified representation of array

programs and translations between array languages. Taking into the account this experience, we are working on the inverse functions that translate array languages into the unified intermediate form.

Acknowledgments

This work was supported in part by grants EP/L00058X/1 and EP/N028201/1 from the Engineering and Physical Sciences Research Council (EPSRC).

References

[1] Robert Bernecky. 1984. Function Arrays. *ACM SIGAPL Quote Quad* 14, 4 (June 1984), 53–56.

[2] Robert Bernecky. 1997. *APEX: The APL Parallel Executor.* Master's thesis. University of Toronto.

[3] Robert Bernecky and R.K.W. Hui. 1991. Gerunds and Representations. *ACM SIGAPL Quote Quad* 21, 4 (July 1991).

[4] Robert Bernecky and Kenneth E. Iverson. 1980. Operators and Enclosed Arrays. In *APL Users Meeting 1980.* I.P. Sharp Associates Limited, Toronto, Canada.

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[6] Hans-J Boehm and Paul F Dubois. 1995. Dynamic memory allocation and garbage collection. *Computers in Physics* 9, 3 (1995), 297–303.

[7] Albert Cohen, Sebastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann, Oleg Kiselyov, and David Padua. 2006. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming* 62, 1 (2006), 25 – 46. https://doi.org/10.1016/j.scico.2005.10.013 Special Issue on the First MetaOCaml Workshop 2004.

[8] Dyalog Limited 2017. *Compiler User Guide* (Dyalog version 16.0 ed.). Dyalog Limited, Bramley, Hampshire, UK.

[9] Dyalog Limited 2017. *Dyalog APL Language Reference Guide* (Dyalog version 16.0 ed.). Dyalog Limited, Bramley, Hampshire, UK. http://docs.dyalog.com/16.0/Dyalog%20APL%20Language%20Reference%20Guide.pdf

[10] Dyalog Limited 2017. *Dyalog APL Programming Reference Guide* (Dyalog version 16.0 ed.). Dyalog Limited, Bramley, Hampshire, UK. http://docs.dyalog.com/16.0/Dyalog%20Programming%20Reference%20Guide.pdf

[11] Martin Elsman and Martin Dybdal. 2014. Compiling a Subset of APL Into a Typed Intermediate Language. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14).* ACM, New York, NY, USA, Article 101, 6 pages. https://doi.org/10.1145/2627373.2627390

[12] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 556–571.

[13] L.M. Restifo Mullin. 1988. *A Mathematics of Arrays.* Ph.D. Dissertation. Syracuse University.

[14] Travis E Oliphant. 2006. *A guide to NumPy.* Vol. 1. Trelgol Publishing USA.

[15] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).* ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[16] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *OOPSLA Companion.*

[17] Sven-Bodo Scholz. 2003. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming* 13, 6 (2003), 1005–1059.

[18] Artjoms Šinkarovs and Sven-Bodo Scholz. 2017. A Lambda Calculus for Transfinite Arrays: Unifying Arrays and Streams. *CoRR* abs/1710.03832 (2017). arXiv:1710.03832 http://arxiv.org/abs/1710.03832

[19] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–46.

[20] M. Steuwer, T. Remmelg, and C. Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 74–85. https://doi.org/10.1109/CGO.2017.7863730

[21] Adilla Susungi, Norman A. Rink, Jerónimo Castrillón, Immo Huismann, Albert Cohen, Claude Tadonki, Jörg Stiller, and Jochen Fröhlich. 2017. Towards compositional and generative tensor optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 169–175. https://doi.org/10.1145/3136040.3136050

[22] Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming* 78, 7 (2009), 643 – 664. https://doi.org/10.1016/j.jlap.2009.03.002 The 19th Nordic Workshop on Programming Theory (NWPT 2007).