A Rosetta Stone for Array Languages

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

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

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

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 \texttt{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, \texttt{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:

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ARRAY’18, June 19, 2018, Philadelphia, PA, USA
© 2018 Copyright held by the owner/author(s).
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 \texttt{Julia} or \texttt{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:

$$T : \mathbb{X} \rightarrow \mathbb{X} \quad \mathbb{X} = \{X_1, \ldots, X_n\}$$

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

$$T = LT \circ TL \quad TL : \mathbb{X} \rightarrow \{L\} \quad LT : \{L\} \rightarrow \mathbb{X}$$

$T$ 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 exist 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 \( (m, n) \).

We implement\(^1\) these concepts in a strict version of the \( \text{HeH} \) programming language [18], an applied \( \lambda \)-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 index into that array. A valid index is a vector containing as many elements as the array has dimensions; otherwise it is undefined.

\[
\begin{align*}
[1, 2, 3, 4], [0] & = 1 & [1, 2], [3, 4], [1, 1] & = 4 & [1, 2], [3, 4], [1] & = \bot \\
\end{align*}
\]

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:

\[
|[[1, 2]]| = [2] & |[[1, 2]]| = [1, 0] & |true| = [] & |42| = [] & |\lambda x.x| = []
\]

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:

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

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 \([5]\):

\[
\text{imap} [5] \_ (iv): 1
\]

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:

\[
\]

\(^1\)Source code, documentation, and examples are available at https://github.com/ashinkarov/heh.
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 \texttt{imap} evaluates non-scalar results (objects of non-empty shape), we require a programmer to specify that shape using the following syntax:
\[
\texttt{imap} \ [3] [\{2\} \ {\_ (iv): [1, 2]}
\]
which evaluates to the array \([[[1, 2], [1, 2], [1, 2]], [2, 2]]\), 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 

[2] To simplify presentation, we assume that \(s1\) is a variable, therefore rewrites do not introduce code duplication.

\[
\texttt{imap} \ [1, 0, 5] \ {\_ (iv): 1}
\]
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:
\[
\begin{align*}
\texttt{letrec} \ &\texttt{matmul1 = } \lambda a. Ab. \\
&\texttt{imap} \ [a] \ {\_ (iv): a. iv + b. iv}
\end{align*}
\]
\[
\begin{align*}
\texttt{letrec} \ &\texttt{matmul2 = } \lambda a. Ab. \\
&\texttt{imap} \ [a] \ {\_ (iv): a. iv + b. iv}
\end{align*}
\]

The first version uses a \texttt{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: \(1000 \times 1000\), and \(100 \times 100\) 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 \texttt{with-loop} \cite{withLoop}. A \texttt{with-loop} makes it possible to define index-value mapping in the same way as partitions in \texttt{imap}. However, a \texttt{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:

\[
// \texttt{imap} \ [5] \ {\_ (iv): 1}
\]
where \((. \leq iv \leq .)\) is the syntax for the entire partition e.g. \(_{(iv)}\) in Heh. The \texttt{genarray} construct takes two arguments: the shape and a default element. The latter is needed when partitions of the \texttt{with-loop} do not cover the entire index space. In Heh, this would be a runtime error.

For non-scalar \texttt{imaps} 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]
// with {
//   (0 <= iv <= 5)
//   s1 = [5]; s2 = [3];
//   genarray (s1, zero (s2));
//}
```

where the function \texttt{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 \texttt{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 \texttt{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.

\texttt{SAC} requires all functions to specify their argument and return types. The type system of \texttt{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 <= iv < s): x[iv] + y[iv];
//   }: genarray (s, 0);
// }
```

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

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

### Evaluation

Due to \texttt{Heh} and \texttt{SAC} similarities, we can easily express shape-invariant functions, \texttt{imaps}, non-scalar \texttt{imaps} and reductions. Runtime results in seconds are presented in Table 1.

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

The second version of matrix multiplication performs less well, suggesting that \texttt{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 \texttt{SAC} would require extensions to the \texttt{SAC} compiler, as there is no obvious way to emulate those constructs.

## 4 Heh to APL

\texttt{APL} provides a rich set of array operations that we used to implement \texttt{Heh} primitives. \texttt{APL} arrays are first class, and a built-in shape function, \(ρ \ a\), 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.

\texttt{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 \(λx.λy. x+y\), where \(ω\) is a formal parameter to \(f\) that we bind to local variables \(x\) and \(y\):

```
f ∗ { (x y) ∗ ω x + y }
```

We used \texttt{guards}, similar to \texttt{return} statements in imperative programs, to represent \texttt{Heh} conditionals. The \(p: e\) expression returns the result of \(e\) if the predicate \(p\) is true. We translated a \texttt{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.

\footnote{We use \texttt{APL} in the form of John Scholes’ \texttt{Dfns}[10], a functional subset of \texttt{Dyadic} \texttt{APL}[9].}
condf ← {
  tr_br ← { f a }
  fl_br ← { g b }
  ω : tr_br 0
  fl_br 0
}
cond (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:

\[
\text{indexset} \leftarrow \{(\omega)+1:\alpha-\omega\}
\]

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:

\[
\begin{align*}
2 & 3 & \text{indexset} & 4 & 6 \\
2 & 3 & 2 & 4 & 2 & 5 \\
3 & 3 & 3 & 4 & 3 & 5 \\
\end{align*}
\]

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

\[
\text{imap} \leftarrow \{
\text{res} \leftarrow a \odot (lb \; ub) \odot \omega
\text{iv} \leftarrow lb \; \text{indexset} \; ub
\text{empty} \leftarrow 0 \equiv x \;/ \text{iv}
\text{empty} : \text{res}
\text{vals} \leftarrow a \alpha " \; \text{iv}
\text{vals} \; @ \text{iv} \; \text{res}
\}
\]

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 \(\lambda a.\text{imap} [a] \{ (iv) : a.iv + 1 \text{ expression} \} \):

\[
\text{inc} \leftarrow \{
\text{s} \leftarrow \omega
\text{f} \leftarrow \{ 1 + (\omega \; [] \; s) \}
\text{ss} \leftarrow (\rho s) \rho 0
\text{≠} (\text{ss} \; f \; \text{imap} \; (0 \times (\rho s)) \; (\rho s))
\}
\]

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.

---

### 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.

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 \(f^0Ωs\) \([4]\) 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.

---

\(^4\) Available at https://github.com/ashinkarov/heh/tree/apl-backend.
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:

```python
# 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:

```python
# 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:

```python
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 `hehimap`:

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

We compute the result shape by concatenating shapes $s_1$ and $s_2$ (we also type-cast the array into a tuple), then create an empty array of that shape using `np.empty`. This function creates a placeholder for that array, but does not set values. Then, we iterate over the index-range $s_1$, 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:

```python
# 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 ([1]), 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:

```python
# imap [4] [[0] <= iv < [2]: 1,
# [2] <= iv < [4]: 2

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

Details on HEH to PYTHON translation, including OCAML code, can be found in the `numpy-backend` branch 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:

```python
x_1 = np.array (2)
x_2 = np.array ([1])
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 `imaps` are translated using higher-order functions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>hCPython</th>
<th>hPyPy</th>
<th>CPython</th>
<th>PyPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>matplus 1000×1000</td>
<td>27</td>
<td>11</td>
<td>13.3</td>
<td>7.8</td>
</tr>
<tr>
<td>matmul1 100×100</td>
<td>18</td>
<td>12</td>
<td>12.1</td>
<td>11.1</td>
</tr>
<tr>
<td>matmul2 100×100</td>
<td>6.1</td>
<td>7.9</td>
<td>5.7</td>
<td>7.6</td>
</tr>
</tbody>
</table>

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 1000×1000 matrices.
1000 matrices from CPYTHON, and NotImplemented 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:

```julia
# 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 each 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:

```julia
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:

```julia
# 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:

```julia
# (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))
        )
        )
    for idx in eachindex(res)
        res[idx] = f(heh_tup2arr(ind2sub(res, idx))−1)
    end
    f1 = collect(iterators.flatten(res))
    return reshape(f1, 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 each index 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>hJULIA</th>
<th>JULIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>matplus 1000 × 1000</td>
<td>11.2</td>
<td>13.2</td>
</tr>
<tr>
<td>matmul1 100 × 100</td>
<td>17.4</td>
<td>16.6</td>
</tr>
<tr>
<td>matmul2 100 × 100</td>
<td>14.4</td>
<td>14.5</td>
</tr>
</tbody>
</table>

**Table 3. JULIA Runtimes (s)**

**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:

```c
// stack-allocate a flat 5×6 array
int b[5][6];
```

```c
// 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](https://github.com/ashinkarov/heh/tree/julia-backend).
We index these objects using square bracket syntax: \[a[0][1]\] and \((a+b)[0][1]\). 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:

```c
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_scalar;
    struct value_array val_array;
};
```

We use a primitive memory model that directly follows the HèH code. All arrays are read-only; we always create a new array on the heap to store an `imap` 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:

```c
// 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 iteration, 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 `imaps`.

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 of the HèH repository.

**Evaluation** With a C backend, we can express all the shape-invariant functions, non-scalar `imaps` 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.

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 HèH to Futhark

**Futhark** [12] is touted as a “High-performance purely functional data-parallel array programming language”

Futhark also provides a generic `reduce` operator identical to that in HèH. For example:

```futharkreduce (+) 0 \[1,2\],\[3,4\]\n```

evaluates to 10. FUTHKARK provides a `map` combinator that operates in a usual way on 1-dimensional arrays, so this expression evaluates to \([2,3,4]\).

```futharkmap (\x \rightarrow x + 1) \[1,2,3\]\n```

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

```futharkmap (\row \rightarrow map (\x \rightarrow x + 1) row) \[1,2,3\]\n```

It in unclear how we might express a shape-invariant function in FUTHKARK. One way would be to obtain an argument’s shape, then recursively apply `map` until we get to vectors.

---

1Available at https://github.com/ashinkarov/heh/tree/c-backend.

2The 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.:

\[
\text{let sum_shape \( m \)[\( n \)](a: \[m\][\( n \)]\( \text{int} \)) = m+n in}
\]

\[
\text{sum_shape \{[1,2],[3,4]\}}
\]

defines a function \text{sum_shape} that binds variables \( m \) and \( n \) to shape components of the function argument. That is, \text{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 \text{sum_shape} function operates only on rank-2 arrays.

Such restrictions pose challenges to our proposed translation. We could take the R route and generate all operations on a flat array representation, but this would undermine \text{Futhark}'s design. We can translate those \text{Hek} 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 \text{imap} body, as all data can be discarded at the end of every iteration.

It would also help to identify which \text{imaps} 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 be the 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 \text{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


The 19th Nordic Workshop on Programming Theory (NWPT 2007).