Compiling a Subset of APL Into a Typed Intermediate Language *

Martin Elsman Martin Dybdal
Department of Computer Science, University of Copenhagen, Denmark
dybber@dybber.dk, mael@di.ku.dk

Abstract

We present a compiler and a typed intermediate language for a subset of APL. The intermediate language treats all numeric data as multi-dimensional arrays and the type system makes explicit the ranks of arrays. Primitive operators are polymorphic in rank and in the type of the underlying data operated on.

The frontend of the APL compiler deals with much of the gory details of the APL language, including infix resolution, resolution of identity items for reduce operations, resolution of default element values for empty arrays, scalar extensions, and resolution of certain kinds of overloading.

We demonstrate the usefulness of the intermediate language by showing that it can be compiled efficiently, using known techniques, such as delayed arrays, into a C-like language. We also demonstrate that the language is sufficiently expressive that some primitive operators, such as APL's inner product operator, which works on arrays of arbitrary dimensions, can be compiled using more primitive operators.

Categories and Subject Descriptors D.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

Keywords APL, Lambda Calculus, Compilation, Optimization, Fusion

1. Introduction

In recent years, array languages have received renewed attention due, in large parts, to the profound promise of data parallelism and the use of array combinators as an important abstraction mechanism for operating, efficiently, on bulk data objects.

APL is an array programming language pioneered by Kenneth E. Iverson in the 60's [11]. The language is a dynamically typed, multi-dimensional language with a functional core supporting first and second-order functions. The language makes extensive use of special characters for denoting the many different special built-in functions and operators, such as array transposition (⍉), rotation (⌽), and a generalized multi-dimensional inner-product operator. Its concise syntax[1] and its rapid prototyping and debugging facilities are being considered harmful by some and fruitful by others. APL and its derivatives, such as J [3] and K [20], are still being used extensively in certain domains, such as in the financial industry, where large code bases are still operational and being actively developed.

Traditionally, APL is an interpreted language, although there have been many attempts at compiling APL into low-level code, both in online and offline settings. For instance, Guibas and Wyatt have demonstrated how a subset of APL can be compiled using a delayed representation of arrays [8]. Other attempts at compiling APL include APEX [1], which also contains a backend for targeting SaC [7].

Until recently, the programming language semantics community have paid only little attention to the APL language, and Kenneth E. Iverson never developed a semantics for APL in terms of a formal model. Apart from recent work by Slepak et al. [16], there have been few attempts at developing formal models or type systems for APL. On the other hand, recent development in data-parallel language implementations [5, 12, 14] have resulted in promising and scalable techniques for high-level programming of highly-parallel hardware, such as GPGPUs.

In this paper, we present a compiler that compiles a subset of APL into a typed intermediate representation, which should serve as a practical and well-defined intermediate format for targeting parallel-architectures through a large number of existing tools and frameworks. The intermediate language is conceptually close to the language Repa [12]. It supports shape-polymorphic functions and types that classify shapes. The compiler takes a simplified approach to certain aspects of APL. Following other APL compilation approaches, the compiler is based on lexical (i.e., static) identifier scoping and has no support for dynamic compilation (APL execute) [1, 2].

As a simple example of the compilation, consider the following signal processing program, derived from the APEX benchmark suite [1]:

```
diff ← {1↓⍵-¯1⌽⍵}
signal ← {¯50⌈50⌊50×(diff 0,⍵)÷0.01+⍵}
+/ signal 9 8 6 8 7 4 4 3 2 2 1 2 4 5 6
```

This program declares two one-parameter functions `diff` and `signal`, in which the formal parameter is referenced using `⍵`. In the `diff` function, the expression `¯1⌽⍵` specifies that the param-

* This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center HIPERFIT: Functional High Performance Computing for Financial Information Technology (`hiperfit.dk`) under contract number 10-092299.

[1] Conway's Game of Life can be written in one line—see youtube video: `https://www.youtube.com/watch?v=a9xAKttWgP4&fmt=18`

```
let v0:Sh(15) = [9,8,6,8,7,4,4,3,2,2,1,2,4,5,6] in
let v3:Sh(16) = consSh(0,v0) in
reduce(addd,0.00,
 each(fn v11:[double]0 => maxd(i2d(~50),v11),
  each(fn v10:[double]0 => mind(i2d(50),v10),
   each(fn v9:[double]0 => muld(i2d(50),v9),
    sum(divd,
     each(i2d,
      drop(1,sum(subi,v3,rotateSh(~1,v3)))),
       each(fn v2:[double]0 => addd(0.01,v2),
        each(i2d,v0)))))))
```

Figure 1. The result of compiling the example APL program into an explicitly typed intermediate representation. Notice the presence of shape types with explicit length attributes.

eter vector is rotated one entry to the right. This vector is then subtracted from the argument vector (point-wise), and the result of this subtraction is returned as the result with the first element dropped. The last line of the program calls the `signal` function on an input vector and sums (i.e., sum-reduces) the result of the call. In the compiled version of the program, which is presented in Figure 1, bulk operations are replaced with calls to the `each` function, which is much like map in other functional languages. Moreover, the compiler has inserted explicit integer-to-double coercions to assure that all arrays contain elements of the same type. Array types in the target language are annotated with explicit ranks; the type `[double]0`, for instance, is the type of double-precision scalar values, which are treated in the type system as arrays of rank zero. Also notice the special shape types (e.g., `Sh(16)`), which range over integer vectors[2] of a specific length. Finally, notice that the compiler has identified 0.0 as the neutral element for a reduction with the `addd` operation.

The contributions of this paper are threefold:

1. In Section 2, we present a statically typed intermediate array language with support for multi-dimensional arrays and operations on such arrays. We demonstrate that the language is suitable as the target for an inference algorithm for an APL compiler. The compiler is open source and available in source form at `https://github.com/melsman/aplcompile`.
2. We demonstrate the flexibility of the typing features, which include shape types and singleton types. Following Repa, the type system keeps track of the shape of an array in its type and contains a separate type for array shapes. Moreover, the type system also makes use of singleton types for singleton integers and for one-element integer vectors. As we shall see in Section 3, these typing features allow the compiler to treat complex operations, such as matrix-multiplication, and generalized versions thereof (inner products of higher-ranked arrays) as operations derived from the composition of other more basic operations.
3. Finally, we demonstrate, also in Section 3, that the intermediate language is useful as a language for further array-compilation, by demonstrating, as an example, that the array language can be compiled into a C-like language, using functional delayed arrays [5, 6], also called pull arrays.

2. A Typed Intermediate Array Language

We assume a denumerable infinite set of *program variables* (x). We use i and n to range over integers and d to range over doubles. Whenever z is some object, we write $\vec{z}$ to range over sequences of similar objects. When we want to be explicit about the size of a sequence $\vec{z} = z_0, \cdots, z_{(n-1)}$, we often write it on the form $\vec{z}^{(n)}$ and we write $z, \vec{z}$ to denote the sequence $z, z_0, \cdots, z_{(n-1)}$.

Shapes (δ), primitive operators (op), values (v), and expressions (e) are defined as follows:

$$
\begin{array}{rcll}
\delta & ::= & \langle \vec{n} \rangle & \text{(shapes)} \\
a & ::= & i \mid d & \\
op & ::= & \texttt{addi} \mid \texttt{subi} \mid \texttt{muli} \mid \texttt{mini} \mid \texttt{maxi} & \text{(operators)} \\
 & \mid & \texttt{addd} \mid \texttt{subd} \mid \texttt{muld} \mid \texttt{mind} \mid \texttt{maxd} & \\
 & \mid & \texttt{iota} \mid \texttt{each} \mid \texttt{reduce} \mid \texttt{i2d} & \\
 & \mid & \texttt{reshape0} \mid \texttt{reshape} \mid \texttt{rotate} & \\
 & \mid & \texttt{transp} \mid \texttt{transp2} \mid \texttt{zipWith} & \\
 & \mid & \texttt{shape} \mid \texttt{take} \mid \texttt{drop} \mid \texttt{first} & \\
 & \mid & \texttt{cat} \mid \texttt{cons} \mid \texttt{snoc} & \text{(derived ops)} \\
 & \mid & \texttt{shapeSh} \mid \texttt{catSh} \mid \texttt{consSh} \mid \texttt{snocSh} & \text{(shape ops)} \\
 & \mid & \texttt{iotaSh} \mid \texttt{rotateSh} & \\
 & \mid & \texttt{takeSh} \mid \texttt{dropSh} \mid \texttt{firstSh} & \\
v & ::= & [\vec{a}]^{\delta} \mid \lambda x.e & \text{(values)} \\
e & ::= & v \mid x \mid [\vec{e}] \mid e\ e' & \text{(expressions)} \\
 & \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid op(\vec{e}) &
\end{array}
$$

Notice that array expressions $[\vec{e}]$ are always one-dimensional, whereas array values $[\vec{a}]^{\delta}$ may be multi-dimensional with their dimensionality specified by the shape δ. We often write i and d to denote scalar values $[i]^{\langle\rangle}$ and $[d]^{\langle\rangle}$, respectively.

2.1 Type System with Shape Polymorphism

We assume denumerable infinite sets of *type variables* (α) and *shape variables* (γ).

$$
\begin{array}{rcll}
\kappa & ::= & \texttt{int} \mid \texttt{double} \mid \alpha & \text{(base types)} \\
\rho & ::= & i \mid \gamma \mid \rho + \rho' & \text{(shape types)} \\
\tau & ::= & [\kappa]^{\rho} \mid \texttt{Sh}\ \rho \mid \texttt{I}\ \rho \mid \texttt{VI}\ \rho \mid \tau \to \tau' & \text{(types)} \\
\sigma & ::= & \forall \vec{\alpha}\vec{\gamma}.\tau & \text{(type schemes)}
\end{array}
$$

Types are segmented into base types (κ), shape types (ρ), types (τ), and type schemes (σ). Shape types (ρ) are considered identical upto associativity and commutativity of $+$ and upto evaluation of constant shape-type expressions involving $+$.

In addition to the array type and shape type constructors, introduced in Section 1, the type system supports singleton integers ($\texttt{I}\ \rho$), single-element integer vectors ($\texttt{VI}\ \rho$), and function types. As special notation, we often write κ to denote the scalar array type $[\kappa]^0$.

A *type substitution* (S_t) maps type variables to base types. A *shape substitution* (S_s) maps shape variables to shape types. A *substitution* (S) is a pair (S_t, S_s) of a type substitution and a shape substitution. Applying a substitution S to some object B, written $S(B)$, has the effect of simultaneously applying S_t and S_s to objects in B (being the identity outside their domain). A type τ' is *an instance of* a type scheme $\sigma = \forall \vec{\alpha}\vec{\gamma}.\tau$, written $\sigma \geq \tau'$, if there exists a substitution S such that $S(\tau) = \tau'$.

A type τ is a *subtype* of another type τ', written $\tau \subseteq \tau'$, if the relation can be derived according to the following rules:

Subtyping $\boxed{\tau \subseteq \tau'}$

$$\frac{}{\tau \subseteq \tau}\ (1) \qquad \frac{\tau_1 \subseteq \tau_2 \quad \tau_2 \subseteq \tau_3}{\tau_1 \subseteq \tau_3}\ (2)$$

$$\frac{}{\texttt{VI}\ \rho \subseteq \texttt{Sh}\ 1}\ (3) \qquad \frac{}{\texttt{Sh}\ \rho \subseteq [\texttt{int}]^1}\ (4) \qquad \frac{}{\texttt{I}\ \rho \subseteq \texttt{int}}\ (5)$$

The subtyping relation is used for allowing known-sized vectors (one-dimensional arrays) to be treated as shape vectors with the number of dimensions being statically known. For instance, we shall see that the constant integer vector expression $[1, 2, 3]$ is given

[2] Vectors are arrays of rank 1.

APL	*op*(*s*)		TySc(*op*)
	`addi`, ...	:	$\texttt{int} \to \texttt{int} \to \texttt{int}$
	`addd`, ...	:	$\texttt{double} \to \texttt{double} \to \texttt{double}$
⍳	`iota`	:	$\texttt{int} \to [\texttt{int}]^1$
¨	`each`	:	$\forall\alpha\beta\gamma.(\alpha \to \beta) \to [\alpha]^\gamma \to [\beta]^\gamma$
/	`reduce`	:	$\forall\alpha\gamma.(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha]^{1+\gamma} \to [\alpha]^\gamma$
⍴	`shape`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to \texttt{Sh}\ \gamma$
⍴	`reshape0`	:	$\forall\alpha\gamma\gamma'.\texttt{Sh}\ \gamma' \to [\alpha]^\gamma \to [\alpha]^{\gamma'}$
⍴	`reshape`	:	$\forall\alpha\gamma\gamma'.\texttt{Sh}\ \gamma' \to \alpha \to [\alpha]^\gamma \to [\alpha]^{\gamma'}$
⌽	`reverse`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^\gamma$
⌽	`rotate`	:	$\forall\alpha\gamma.\texttt{int} \to [\alpha]^\gamma \to [\alpha]^\gamma$
⍉	`transp`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^\gamma$
⍉	`transp2`	:	$\forall\alpha\gamma.\texttt{Sh}\ \gamma \to [\alpha]^\gamma \to [\alpha]^\gamma$
↑	`take`	:	$\forall\alpha\gamma.\texttt{int} \to \alpha \to [\alpha]^\gamma \to [\alpha]^\gamma$
↓	`drop`	:	$\forall\alpha\gamma.\texttt{int} \to [\alpha]^\gamma \to [\alpha]^\gamma$
⊃	`first`	:	$\forall\alpha\gamma.\alpha \to [\alpha]^\gamma \to \alpha$
	`zipWith`	:	$\forall\alpha_1\alpha_2\beta\gamma.(\alpha_1 \to \alpha_2 \to \beta) \to [\alpha_1]^\gamma \to [\alpha_2]^\gamma \to [\beta]^\gamma$
,	`cat`	:	$\forall\alpha\gamma.[\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1}$
,	`cons`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1}$
,	`snoc`	:	$\forall\alpha\gamma.[\alpha]^{\gamma+1} \to [\alpha]^\gamma \to [\alpha]^{\gamma+1}$

Figure 2. Operator type schemes for standard operations.

APL	*op*(*s*)		TySc(*op*)
⍴	`shapeSh`	:	$\forall\gamma.\texttt{Sh}\ \gamma \to \texttt{VI}\ \gamma$
↑	`takeSh`	:	$\forall\gamma.\texttt{I}\ \gamma \to [\texttt{int}]^1 \to \texttt{Sh}\ \gamma$
↓	`dropSh`	:	$\forall\gamma\gamma'.\texttt{I}\ \gamma \to \texttt{Sh}\ (\gamma + \gamma') \to \texttt{Sh}\ \gamma'$
,	`consSh`	:	$\forall\gamma.\texttt{int} \to \texttt{Sh}\ \gamma \to \texttt{Sh}(1 + \gamma)$
,	`snocSh`	:	$\forall\gamma.\texttt{Sh}\ \gamma \to \texttt{int} \to \texttt{Sh}(1 + \gamma)$
⊃	`firstSh`	:	$\forall\gamma.\texttt{VI}\ \gamma \to \texttt{I}\ \gamma$
⍳	`iotaSh`	:	$\forall\gamma.\texttt{I}\ \gamma \to \texttt{Sh}\ \gamma$
⌽	`rotateSh`	:	$\forall\gamma.\texttt{Sh}\ \gamma \to \texttt{Sh}\ \gamma$
,	`catSh`	:	$\forall\gamma\gamma'.\texttt{Sh}\ \gamma \to \texttt{Sh}\ \gamma' \to \texttt{Sh}\ (\gamma + \gamma')$

Figure 3. Operator type schemes for operations on shapes.

the type `Sh` 3, but that the expression can also be given the type $[\texttt{int}]^1$ using the subtyping relation. Similarly, when asking for the shape of a shape vector with type `Sh` γ, we obtain a one-element shape vector containing the value γ. For typing this value, we can use the singleton vector type `VI` γ, which is a subtype of `Sh` 1, the type of one-element shape vectors.

Each operator, *op*, is given a unique type scheme, σ, as specified by the relation $\mathrm{TySc}(op) = \sigma$ defined in Figure 2 and Figure 3. For all operators *op*, such that $\mathrm{TySc}(op) = \forall\vec{\alpha}\vec{\gamma}.\tau_1 \to \ldots \to \tau_n \to \tau$, where τ is not a function type, we say that the *arity* of the operator *op*, written *arity*(*op*), is n.

Type assumptions Γ map variables to type schemes:

$$\Gamma \quad ::= \quad \Gamma, x : \sigma \mid \bullet$$

The type system allows inferences among sentences of the form $\Gamma \vdash e : \tau$, which are read: "under the assumptions Γ, the expression e has type τ."

Shape typing $\boxed{\vdash \delta : \rho}$

$$\frac{}{\vdash \langle \vec{n}^{(i)} \rangle : i} \ (6)$$

Value typing $\boxed{\Gamma \vdash v : \tau}$

$$\frac{\vdash \delta : \rho}{\Gamma \vdash [\vec{i}]^\delta : [\texttt{int}]^\rho} \ (7) \qquad \frac{\vdash \delta : \rho}{\Gamma \vdash [\vec{r}]^\delta : [\texttt{double}]^\rho} \ (8)$$

$$\frac{}{\Gamma \vdash [\vec{i}]^{\langle n \rangle} : \texttt{Sh}\ n} \ (9) \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \ (10)$$

$$\frac{}{\Gamma \vdash [n]^{\langle 1 \rangle} : \texttt{VI}\ n} \ (11) \qquad \frac{}{\Gamma \vdash n : \texttt{I}\ n} \ (12)$$

Expression typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \texttt{I}\ n}{\Gamma \vdash [e] : \texttt{VI}\ n} \ (13) \qquad \frac{\Gamma(x) \geq \tau}{\Gamma \vdash x : \tau} \ (14) \qquad \frac{\tau \subseteq \tau' \quad \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \ (15)$$

$$\frac{\Gamma \vdash e_i : \kappa \quad i = [0; n[}{\Gamma \vdash [\vec{e}^{(n)}] : [\kappa]^1} \ (16) \qquad \frac{\Gamma \vdash e_i : \texttt{int} \quad i = [0; n[}{\Gamma \vdash [\vec{e}^{(n)}] : \texttt{Sh}\ n} \ (17)$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \ (18) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \mathrm{fv}(\vec{\alpha}\vec{\gamma}) \cap \mathrm{fv}(\Gamma, \tau') = \emptyset \quad \Gamma, x : \forall\vec{\alpha}\vec{\gamma}.\tau \vdash e_2 : \tau'}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau'} \ (19)$$

$$\frac{\mathrm{TySc}(op) \geq \tau_0 \to \ldots \to \tau_{(n-1)} \to \tau \quad \mathrm{arity}(op) = n \quad \Gamma \vdash e_i : \tau_i \quad i = [0; n[}{\Gamma \vdash op(\vec{e}^{(n)}) : \tau} \ (20)$$

Notice that operators are required to be fully applied. In examples, however, when an operator *op* is not applied to any arguments (it is perhaps given as argument to a higher-order function), it is eta-expanded into the form $\lambda x_1.\cdots\lambda x_n.op(x_1, \cdots, x_n)$, where $n = \mathrm{arity}(op)$.

As indicated by the operator type schemes, certain limitations apply. For instance, in accordance with APL, the `each` operator operates on each base value of a multi-dimensional array. One may consider, instead, providing a `map` operator with the following type scheme:

$$\texttt{map} \quad : \quad \forall\alpha\beta\gamma.([\alpha]^\gamma \to [\beta]^\gamma) \to [\alpha]^{1+\gamma} \to [\beta]^{1+\gamma}$$

However, it is not possible, with the present type system, to express that the function returns arrays with the same extent for all arguments; the only guarantee the type system can give us is that result arrays have the same rank (number of dimensions). More expressive type systems, based on dependent types, allow for expressing more accurately, the assumptions of the higher-order operators [17–19].

Similarly, one may consider providing a `reduce'` operator with the following type scheme:

$$\texttt{reduce'} \quad : \quad \forall\alpha\gamma.([\alpha]^\gamma \to [\alpha]^\gamma \to [\alpha]^\gamma) \to [\alpha]^\gamma \to [\alpha]^{1+\gamma} \to [\alpha]^\gamma$$

The idea here is that the operator operates on entire subparts of the argument array. In a system supporting only `map` and `reduce'` (and not `each` and `reduce`), nested instances of `map`s can be used instead of `each` and nested instances of `map`s with an inner `reduce'` can be used instead of `reduce`. Additionally, one may consider providing a sequential `fold` operator that does not require associativity of the argument function:

$$\texttt{fold} \quad : \quad \forall\alpha\beta\gamma.([\alpha]^\gamma \to [\beta]^{\gamma'} \to [\beta]^{\gamma'}) \to [\beta]^{\gamma'} \to [\alpha]^{1+\gamma} \to [\beta]^{\gamma'}$$

As we shall see in the next section, the semantics of `reduce` is that it reduces the argument array along its last dimension, following the traditional APL semantics [11, 13].

The implementation of the APL compiler uses a hybrid approach of type inference and local context querying for resolving array ranks, scalar extensions, and identity items (neutral elements) during intermediate language program generation. The type inference is based on a simple unification algorithm using conditional unification for the implementation of the simple subtyping inference.

2.2 Example Programs

We now present a few example programs that utilizes the various operators. The dot-product of two integer arrays can be defined in the language as follows:

```
dotpi : ∀γ.[int]^{1+γ} → [int]^{1+γ} → [int]^γ =
  λx.λy.reduce(addi,0,zipWith(muli,x,y))
```

Notice that this function also works with integer matrices and integer arrays of higher dimensions. In case the extents of the argument arrays do not match up, the `zipWith` expression—and therefore the `dotpi` call—will result in a runtime error, as further specified in section 2.3. We can generalize the above function to be useful in a broader sense:

```
dotp : ∀γα.(α → α → α) → (α → α → α) → α
          → [α]^{1+γ} → [α]^{1+γ} → [α]^γ =
  λadd.λmul.λn.λx.λy.
      reduce(add,n,zipWith(mul,x,y))
```

2.3 Dynamic Semantics

Evaluation contexts, ranged over by E, take the following form:

$$\begin{array}{rcl} E & ::= & [\cdot] \mid [\vec{v} E \vec{e}] \mid E\ e \mid v\ E \\ & \mid & \texttt{let}\ x = E\ \texttt{in}\ e \mid op(\vec{v} E \vec{e}) \end{array}$$

When E is an evaluation context and e is an expression, we write $E[e]$ to denote the expression resulting from filling the hole in E with e.

The dynamic semantics is presented as a small step reduction semantics, which is explicit about certain kinds of errors that are not easily checked statically. Intuitively, a well-typed expression e is either a value or it can be reduced into another expression or the special token **err**. Errors that are treated explicitly include negative values passed to `iota`.

We first define a few helper functions for computations on shapes and for converting between flat indexing and multi-dimensional indexing. We assume a reverse operation (rev) on shapes and an operation, named product, that takes a shape of an array and returns the number of elements in the flattened version of the array. An expression $\mathrm{fromSh}_\delta \delta'$ takes a shape δ of an array and a multi-dimensional index δ' into the array and returns a corresponding index into the flattened version of the array.

$$\begin{array}{rcl} \mathrm{fromSh}_{\langle\rangle}\langle\rangle & = & 0 \\ \mathrm{fromSh}_{\langle n,\vec{n}\rangle}\langle i, \vec{i}\rangle & = & i * p + \mathrm{fromSh}_{\langle\vec{n}\rangle}\langle\vec{i}\rangle \\ \quad \text{where } p = \mathrm{product}(\vec{n}) & & \end{array}$$

An expression $\mathrm{toSh}_\delta i$ takes a shape δ and an index i into the flattened version of the array and returns the corresponding multi-dimensional index into the array.

$$\begin{array}{rcl} \mathrm{toSh}_{\langle\rangle} 0 & = & \langle\rangle \\ \mathrm{toSh}_{\langle n,\vec{n}\rangle} i & = & \langle i \operatorname{div} p, \vec{i}\rangle \\ \quad \text{where} \quad p & = & \mathrm{product}(\vec{n}) \\ \quad\quad \langle\vec{i}\rangle & = & \mathrm{toSh}_{\langle\vec{n}\rangle}(i \bmod p) \end{array}$$

The expression $\mathrm{exchange}_\delta \delta'$ exchanges the elements in the shape δ' according to δ:

$$\begin{array}{l} \mathrm{exchange}_{\langle\vec{p}^{(n)}\rangle}\langle\vec{q}^{(n)}\rangle = \langle q_{p_0}, \cdots, q_{p_{(n-1)}}\rangle \\ \quad \text{where } \forall ij.i \neq j \Rightarrow p_i \neq p_j \end{array}$$

Notice the partiality of the exchange function; if $\delta' = \mathrm{exchange}_{\langle\vec{i}^{(k)}\rangle}\delta$ then $\vec{i}^{(k)}$ is known to be a permutation of $0, \cdots, (k-1)$.

A majority of the dynamic semantics rules are given below. For space reasons, we have left out the rules for `rotate`, `cat`, `cons`, `snoc`, and `drop`. We have also left out the rules for the shape-versions of the operations (e.g., `takeSh`), which are all easily defined in terms of the non-shape versions.

Small Step Reductions $\boxed{e \hookrightarrow e'/\textbf{err}}$

$$\frac{e \hookrightarrow e' \quad E \neq [\cdot]}{E[e] \hookrightarrow E[e']}\ (21) \qquad \frac{e \hookrightarrow \textbf{err} \quad E \neq [\cdot]}{E[e] \hookrightarrow \textbf{err}}\ (22)$$

$$\frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \hookrightarrow e[v/x]}\ (23) \qquad \frac{}{(\lambda x.e)\ v \hookrightarrow e[v/x]}\ (24)$$

$$\frac{}{[\vec{a}^{(n)}] \hookrightarrow [\vec{a}^{(n)}]^{\langle n\rangle}}\ (25)$$

$$\frac{i = i_1 + i_2}{\texttt{addi}(i_1, i_2) \hookrightarrow i}\ (26) \qquad \frac{d = d_1 + d_2}{\texttt{addd}(d_1, d_2) \hookrightarrow d}\ (27)$$

$$\frac{n \geq 0}{\texttt{iota}(n) \hookrightarrow [1, \cdots, n]^{\langle n\rangle}}\ (28) \qquad \frac{n < 0}{\texttt{iota}(n) \hookrightarrow \textbf{err}}\ (29)$$

$$\frac{e = [v_f\ a_0, \cdots, v_f\ a_{(n-1)}]}{\texttt{each}(v_f, [\vec{a}^{(n)}]^\delta) \hookrightarrow \texttt{reshape0}(\delta, e)}\ (30)$$

$$\frac{\begin{array}{c}\delta = \langle\vec{n}, m\rangle \quad k = \mathrm{product}(\vec{n}) \quad i = [0; m[\\ e_i = v_f\ a_{(i*k)}\ (\cdots (v_f\ a_{(i*k+m-1)}\ v)\cdots)\end{array}}{\texttt{reduce}(v_f, v, [\vec{a}^{(n)}]^\delta) \hookrightarrow \texttt{reshape0}(\langle\vec{n}\rangle, [\vec{e}^{(k)}])}\ (31)$$

$$\frac{m = \mathrm{product}(\delta') \quad f(i) = i \bmod n \quad n > 0}{\texttt{reshape}(\delta', a, [\vec{a}^{(n)}]^\delta) \hookrightarrow [a_{f(0)}, \cdots, a_{f(m-1)}]^{\delta'}}\ (32)$$

$$\frac{m = \mathrm{product}(\delta') \quad a_i = a \quad i = [0; m[}{\texttt{reshape}(\delta', a, []^\delta) \hookrightarrow [a_0, \cdots, a_{(m-1)}]^{\delta'}}\ (33)$$

$$\frac{\delta' = \mathrm{rev}(\delta) \quad f = \mathrm{fromSh}_{\delta'}\ o\ \mathrm{rev}\ o\ \mathrm{toSh}_\delta}{\texttt{transp}([\vec{a}^{(n)}]^\delta) \hookrightarrow [a_{f(0)}, \cdots, a_{f(n-1)}]^{\delta'}}\ (34)$$

$$\frac{\begin{array}{c}\delta' = \mathrm{exchange}_{\delta_0}(\delta) \\ f = \mathrm{fromSh}_{\delta'}\ o\ \mathrm{exchange}_{\delta_0}\ o\ \mathrm{toSh}_\delta\end{array}}{\texttt{transp2}(\delta_0, [\vec{a}^{(n)}]^\delta) \hookrightarrow [a_{f(0)}, \cdots, a_{f(n-1)}]^{\delta'}}\ (35)$$

$$\frac{\neg\exists\delta'.\delta' = \mathrm{exchange}_{\delta_0}(\delta)}{\texttt{transp2}(\delta_0, [\vec{a}]^\delta) \hookrightarrow \textbf{err}}\ (36)$$

$$\frac{\begin{array}{c}m \geq 0 \quad \delta' = \langle m, \vec{n}\rangle \quad j = \mathrm{product}(\delta') \\ f(i) = \text{if } i < k \text{ then } a_i \text{ else } a\end{array}}{\texttt{take}(m, a, [\vec{a}^{(k)}]^{\langle n,\vec{n}\rangle}) \hookrightarrow [f(0), \cdots, f(j-1)]^{\delta'}}\ (37)$$

$$\frac{\begin{array}{c}m < 0 \quad \delta' = \langle -m, \vec{n}\rangle \quad j = \mathrm{product}(\delta') \\ f(i) = \text{if } i < k \text{ then } a_{(k-1-i)} \text{ else } a\end{array}}{\texttt{take}(m, a, [\vec{a}^{(k)}]^\delta) \hookrightarrow [f(0), \cdots, f(j-1)]^{\delta'}}\ (38)$$

$$\frac{k > 0}{\texttt{first}(a, [\vec{a}^{(k)}]^\delta) \hookrightarrow a_0}\ (39) \qquad \frac{}{\texttt{first}(a, []^\delta) \hookrightarrow a}\ (40)$$

The transitive, reflexive closure of $\hookrightarrow$, written $\hookrightarrow^*$, is defined by the following two rules:

$$\frac{e \hookrightarrow e' \quad e' \hookrightarrow^* e''}{e \hookrightarrow^* e''}\ (41) \qquad \frac{}{e \hookrightarrow^* e}\ (42)$$

We further define $e\uparrow$ to mean that there exists an infinite sequence $e \hookrightarrow e_1 \hookrightarrow e_2 \hookrightarrow \cdots$. The presented language does not support general recursion or uncontrolled looping, thus all pro-

grams represented in the intermediate language are guaranteed to terminate. The semantic machinery, however, does support the addition of recursion (e.g., for implementing APL's recursion operator ∇).

2.4 Properties of the Language

In the following, we give a few definitions before we present a unique decomposition proposition. This proposition is used for the proofs of type preservation and progress, which allow us to establish a type soundness result for the language [15].

A *redex*, ranged over by r, is an expression of the form

$$r \quad ::= \quad (\lambda x.e)\ v \ \mid\ \texttt{let}\ x = v\ \texttt{in}\ e \ \mid\ op(\vec{v}) \ \mid\ [\vec{v}]$$

The following unique decomposition proposition states that any well-typed term is either a value or the term can be decomposed into a unique context and a unique well-typed redex. Moreover, filling the context with an expression of the same type as the redex results in a well-typed term:

PROPOSITION 1 (Unique Decomposition). *If* $\vdash e : \tau$ *then either* e *is a value* v *or there exists a unique* E*, a unique redex* e'*, and some* τ' *such that* $e = E[e']$ *and* $\vdash e' : \tau'$*. Furthermore, for all* e'' *such that* $\vdash e'' : \tau'$*, it follows that* $\vdash E[e''] : \tau$*.*

PROOF By induction over the derivation $\vdash e : \tau$. □

The proofs of the following type preservation and progress propositions are then straightforward and standard [15].

PROPOSITION 2 (Type Preservation). *If* $\vdash e : \tau$ *and* $e \hookrightarrow e'$ *and* $e' \neq \textbf{\textit{err}}$ *then* $\vdash e' : \tau$.

PROOF By induction over the structure of the typing derivation $\vdash e : \tau$, using Proposition 1. □

PROPOSITION 3 (Progress). *If* $\Gamma \vdash e : \tau$ *then either*

1. e *is a value; or*
2. $e \hookrightarrow \textbf{\textit{err}}$*; or*
3. *there exists an expression* e' *such that* $e \hookrightarrow e'$.

PROOF By induction over the structure of the typing derivation. □

PROPOSITION 4 (Type Soundness). *If* $\vdash e : \tau$ *then either* $e\uparrow$ *or there exists* v *such that* $e \hookrightarrow^* v/\textbf{\textit{err}}$.

PROOF By induction on the number of machine steps using Proposition 2 and Proposition 3. □

3. Compiling the Inner Product

We now adopt the technique used by Guibas and Wyatt [8] for compiling away the "dot" operator in APL, by representing it by a sequence of simpler APL-operations. The APL source code is given in Figure 4, which consist of the definition of an APL dyadic operator `dot` and use of the operator.

We shall not go into discussing and explaining the details of the APL code for the `dot`-operator, except from mentioning that the left and right function argument to the operator is referenced within the definition of the `dot`-operator using `αα` and `ωω`, respectively. The intermediate language code resulting from compiling the `dot`-operator example is shown in Figure 5. The intermediate language code generated for the inner product of two matrices may seem a bit extensive. However, once traditional fusion techniques and other optimizations are applied, the code is simplified drastically, as can be seen in Figure 6, which contains the result of using Obsidian-style pull-arrays for compiling the program [5]. A backend compiler based on pull-arrays is of course only one out of many possibilities for compiling the intermediate language. We envision

```
dot ← {
  WA ← (1↓ρω),ρα
  KA ← (⊃ρρα)-1
  VA ← ⍳ ⊃ ρWA
  ZA ← (KA⍣¯1↓VA),¯1↑VA
  TA ← ZA⍉WAρα    ⍝ Replicate, transpose
  WB ← (¯1↓ρα),ρω
  KB ← ⊃ ρρα
  VB ← ⍳ ⊃ ρWB
  ZB0 ← (-KB) ↓ KB ⍴ ⍳(⊃ρVB)
  ZB ← (¯1↓(⍳ KB)),ZB0,KB
  TB ← ZB⍉WBρω    ⍝ Replicate, transpose
  αα / TA ωω TB   ⍝ Compute the result
}

A ← 3 2 ρ ⍳ 5   ⍝ Example input A
B ← ⍉ A         ⍝ Example input B
R ← A + dot × B
R2 ← ×/ +/ R

⍝       1  3  5
⍝       2  4  1
⍝
⍝ 1 2   5 11  7  -+->    23  |
⍝ 3 4  11 25 19  -+->    55  ×
⍝ 5 1   7 19 26  -+->    52  |
⍝                     65780  v
```

Figure 4. The definition of a general dot-operator in APL together with an application of the operator to functions `+` and `×` and two matrices `A` and `B`.

that the intermediate language can be compiled into other array languages, such as Futhark [9, 10], or into uses of array libraries such as Accelerate [4] or Repa [12].

4. Conclusion and Future Work

We have presented a statically typed intermediate language, used as a target for an APL compiler.

There are several directions for future work. First, several operations, including boolean operations, need to be added to the language and to the compiler in order to test the feasibility of the approach for larger APL programs. We do not see any problems extending the approach in this direction.

Second, it would be interesting to compare the performance of the generated code with various array libraries and languages, including Repa [12] and Futhark [9, 10], and compare the performance of this generated code with the performance of a state-of-the-art APL interpreter, such as Dyalog APL [13].

Finally, as mentioned earlier, it would be interesting to investigate the possibility for compiling the intermediate language into efficient code for multi-core CPUs or many-core GPUs using array libraries such as Repa and Accelerate.

References

[1] Robert Bernecky. APEX: The APL parallel executor. Master's thesis, Graduate Department of Computer Science University of Toronto, 1997.

[2] Robert Bernecky and Stephen B. Jaffe. ACORN: APL to C on real numbers. In *ACM SIGAPL Quote Quad*, pages 40–49, 1990.

[3] Chris Burke and Roger Hui. J for the apl programmer. *SIGAPL APL Quote Quad*, 27(1):11–17, September 1996.

[4] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with mul-

```
bash-3.2$ ../aplc -ml matmul2.apl
Reading file: matmul2.apl
Typed program:
let v0:[int]2  = reshape([3,2],iotaSh(5)) in
let v1:[int]2  = transp(v0) in
let v2:Sh(3)   = catSh(dropSh(1,shape(v1)),
                   shape(v0)) in
let v3:[int]0  = subi(firstSh(shapeSh(
                   shape(v0))),1) in
let v4:Sh(3)   = iotaSh(firstSh(shapeSh(v2))) in
let v5:Sh(3)   = catSh(rotateSh(v3,dropSh(~1,v4)),
                   takeSh(~1,v4)) in
let v6:[int]3  = transp2(v5,reshape(v2,v0)) in
let v7:Sh(3)   = catSh(dropSh(~1,shape(v0)),
                   shape(v1)) in
let v8:Si(2)   = firstSh(shapeSh(shape(v0))) in
let v9:Sh(3)   = iotaSh(firstSh(shapeSh(v7))) in
let v10:Sh(1)  = dropSh(negi(v8),rotateSh(v8,
                   iotaSh(firstSh(shapeSh(v9))))) in
let v11:Sh(3)  = catSh(dropSh(~1,iotaSh(v8)),
                   snocSh(v10,v8)) in
let v12:[int]3 = transp2(v11,reshape(v7,v1)) in
let v17:[int]2 = reduce(addi,0,sum(muli,v6,v12)) in
let v22:[int]0 = reduce(muli,1,reduce(addi,0,v17)) in
i2d(v22)
Evaluating program
Result is [](65780.0)
```

Figure 5. Intermediate language code for inner product of a 3×2 matrix and a 2×3 matrix with operations + and ×.

ticore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP'11, pages 3–14. ACM, 2011.

[5] Koen Claessen, Mary Sheeran, and Joel Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP'12. ACM, 2012.

[6] Conal Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

[7] Clemens Grelck and Sven-Bodo Scholz. Accelerating APL programs with SAC. In *Proceedings of the Conference on APL '99: On Track to the 21st Century*, APL'99, pages 50–57. ACM, 1999.

[8] Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'78, pages 1–8. ACM, 1978.

[9] Troels Henriksen and Cosmin E. Oancea. A T2 graph-reduction approach to fusion. In *2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, September 2013.

[10] Troels Henriksen and Cosmin E. Oancea. Bounds checking: An instance of hybrid analysis. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY'14. ACM, 2014.

[11] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, Inc, May 1962.

[12] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, and Simon Peyton Jones. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP'2010, September 2010.

[13] Bernard Legrand. *Mastering Dyalog APL*. Dyalog Limited, November 2009.

[14] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell'10, pages 67–78. ACM, 2010.


```
double kernel(int n10) {
  int* a5 = (int*)malloc(sizeof(int)*6);
  for (int n166 = 0; n166 < 6; n166++) {
    a5[n166] = ((n166%5)+1);
  }
  int* a6 = (int*)malloc(sizeof(int)*6);
  for (int n167 = 0; n167 < 6; n167++) {
    a6[n167] =
      ((((((n167==5) ? n167
                     : ((3*n167)%5))==5)
        ? ((n167==5) ? n167
                     : ((3*n167)%5))
                     : ((2*((n167==5)
        ? n167 : ((3*n167)%5)))%5))%5)+1);
  }
  int n9 = 1;
  for (int n168 = 0; n168 < 3; n168++) {
    int n40 = 0;
    for (int n180 = 0;
         n180 < min(3,max((9-(n168*3)),0));
         n180++) {
      int n49 = 0;
      for (int n183 = 0;
           n183 < min(min(2,max((6-
                      (((n180+(n168*3))
                        /3)*2)),0)),
                  min(2,max((6-(((n180+
                     (n168*3))%3)*2)),0)));
           n183++) {
        n49 += (a5[(n183+(((n180+(n168*3))/3)*2))]*
                a6[(n183+(((n180+(n168*3))%3)*2))]);
      }
      n40 = (n49+n40);
    }
    n9 = (n40*n9);
  }
  return i2d(n9);
}
```

Figure 6. Target C-like code resulting from using pull-arrays for computing the inner product of a 3×2 matrix and a 2×3 matrix with operations + and ×.

[15] Greg Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, USA, December 1995.

[16] Justin Slepak, Olin Shivers, and Panagiotis Manolios. An array-oriented language with static rank polymorphism. In *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 27–46. Springer-Verlag, 2014.

[17] Peter Thiemann and Manuel M. T. Chakravarty. Agda meets accelerate. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, IFL'2012, 2013. Revised Papers, Springer-Verlag, LNCS 8241.

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

[19] Kai Trojahner and Clemens Grelck. Descriptor-free representation of arrays with dependent types. In *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, IFL'08, pages 100–117. Springer-Verlag, 2011.

[20] Arthur Whitney. K. *The Journal of the British APL Association*, 10(1):74–79, July 1993.