Compiling a subset of APL to a Typed Array Intermediate Language

Martin Dybdal
dybber@dybber.dk

HIPERFIT
DIKU
University of Copenhagen

10 October 2015

Joint work with Martin Elsman and Michael Budde

Goals

- GPU programming for APL fingers
- Develop backend technology independently of APL. Other frontends could be J, K, some new Haskell vector-library or R-library.
- Bridging the PL and APL communities
- Performance on code written by non-programmers (e.g. biologist or quant code)

Why APL?

- Notation for non-programmers (biologist/chemist/quants etc.)
- APLs primitives have proven suitable for many applications
- **APL programs are inherently parallel**

  > *"Unlike other languages, the problem in APL is not determining where parallelism exists. Rather, it is to decide what to do with all of it."*
  > *- Robert Bernecky, 1993*

- Existing programs/benchmarks (this is not such a strong point as we originally thought)

Overview

- APL: Dynamically weakly typed array language
- TAIL: Statically strongly typed array language as target for APL and friends.
  - type inference
  - polymorphic shape-types (similar to Repa/Accelerate shapes)
  - singleton-types
  - no nested arrays
  - no heterogeneous arrays

Restrictions

- No nested arrays or nested parallelism
- Lexical scoping (ISO APL Standard specifies dynamic scoping)
- No efficient implementation of boolean arrays as bit-arrays
- No dynamic conversions between integer and doubles (APL normally does this e.g. if overflow would otherwise occur)
- Reshapes not possible when size of shape-vector is unknown
- Several built-ins are still missing
- No labels and gotos, only limited branching
- No recursion

Brief APL introduction

```
⍳5
     1 2 3 4 5

⍴⍳5
     5

2⍴⍳5
     1 2

2 4⍴⍳5
     1 2 3 4
     5 1 2 3

⍴2 4⍴⍳5
     2 4
```

Brief APL introduction

Function definitions:

```
add ← {⍺ × 100 + ⍵}
10 add 42
    1420
```

Implicit vectorisation:

```
7 9 13 = 3 9 15
    0 1 0
```

Brief APL introduction

Function definitions:

```
add ← {⍺ × 100 + ⍵}
10 add 42
    1420
```

Implicit vectorisation:

```
7 9 13 = 3 9 15
    0 1 0
```

Equality of all elements can be written:

```
7 9 13 {∧/,⍺=⍵} 3 9 15
    0

7 9 13 {∧/,⍺=⍵} 7 9 13
    1
```

(or you can use the built in match-operator: ≡)

Brief APL introduction

Function definitions:

```
add ← {⍺ × 100 + ⍵}
10 add 42
      1420
```

Implicit vectorisation:

```
7 9 13 = 3 9 15
      0 1 0
```

Try it yourself

`http://tryapl.org/`

Equality of all elements can be written:

```
7 9 13 {∧/,⍺=⍵} 3 9 15
      0

7 9 13 {∧/,⍺=⍵} 7 9 13
      1
```

(or you can use the built in match-operator: ≡)

Brief APL introduction

```
ι5
    1 2 3 4 5

(1 10 100)∘.+(ι5)
      2   3   4   5   6
     11  12  13  14  15
    101 102 103 104 105
```

Brief APL introduction

```
ι5
      1 2 3 4 5

(1 10 100)∘.+(ι5)
        2   3   4   5   6
       11  12  13  14  15
      101 102 103 104 105
```

Upper triangular matrix idiom:

```
(ι5)∘.≤(ι5)
      1 1 1 1 1
      0 1 1 1 1
      0 0 1 1 1
      0 0 0 1 1
      0 0 0 0 1
```

Brief APL introduction

```
⍳5
      1 2 3 4 5

(1 10 100)∘.+(⍳5)
        2   3   4   5   6
       11  12  13  14  15
      101 102 103 104 105
```

Upper triangular matrix idiom:

```
(⍳5)∘.≤(⍳5)
      1 1 1 1 1
      0 1 1 1 1
      0 0 1 1 1
      0 0 0 1 1
      0 0 0 0 1
```

Idioms library: `http://aplwiki.com/FinnAplIdiomLibrary`

TAIL

- **Make vectorisation and scalar extensions explicit**
- **Statically determine array ranks and shapes (when possible)**
- Insert numeric coercions
- Resolve overloading of numeric operations
- Resolve default arguments (e.g. for over takes)
- Resolve overloading of shape operations

Example: APL $\rightarrow$ TAIL

```
pi ← {
 x ← ?ωρ0
 y ← ?ωρ0
 dists ← (x*2) + y*2
 4×(+/1>dists)÷ω
}
pi 1000000
    3.142668
```

Example: APL → TAIL

```
pi ← {
 x ← ?ωρ0
 y ← ?ωρ0
 dists ← (x*2) + y*2
 4×(+/1>dists)÷ω
}
pi 1000000
    3.142668
  ⇓
let v3:[double]1 =
  each(fn v2:[int]0 => roll(v2), reshape([1000000],[0])) in
let v5:[double]1 =
  each(fn v4:[int]0 => roll(v4), reshape([1000000],[0])) in
let v11:[double]1 =
  each(fn v10:[double]0 => powd(v10,divd(1.0,2.0)),
       zipWith(addd, each(fn v7:[double]0 => powd(v7,2.0),v3),
                     each(fn v6:[double]0 => powd(v6,2.0),v5))) in
muld(4.0, divd(i2d(reduce(addi,0,
                     each(b2i,
                       each(fn v12:[double]0 => gtd(1.0,v12),
                            v11)))),
               1000000.0))
```

Note: each is just map in APL lingo

Implicit vectorisation

Most APL primitives are defined for a specific argument rank k, but in the case it is applied to any array with a rank higher than k it will be applied *independently* to each rank-k subarray.

Negation

```
−17
    ¯17

−⍳6
    ¯1 ¯2 ¯3 ¯4 ¯5 ¯6

−2 3⍴⍳6
    ¯1 ¯2 ¯3
    ¯4 ¯5 ¯6
```

Implicit vectorisation

In TAIL we make vectorisation explicit by inserting applications of `each` and `zipWith`:

$$\texttt{each} : \forall \alpha\beta\gamma.\ (\alpha \to \beta) \to [\alpha]^\gamma \to [\beta]^\gamma$$

$$\texttt{zipWith} : \forall \alpha_1\alpha_2\beta\gamma.\ (\alpha_1 \to \alpha_2 \to \beta) \to [\alpha_1]^\gamma \to [\alpha_2]^\gamma \to [\beta]^\gamma$$

Example

```
(2 3⍴1) + (2 3⍴⍳6)
```

$\Downarrow$

```
zipWith(addi, reshape([2,3], [1]),
              reshape([2,3], iota(6)))
```

Implicit vectorisation

In some cases, applying “each” is not what we want:

Reduction

```
+/1 2 3 4
    10

2 3⍴⍳6
    1 2 3
    4 5 6

+/2 3⍴⍳6    ⍝ sum each row
    6 15
```

Implicit vectorisation

Instead we make reductions work directly on arrays with *rank* > 0 (like Accelerate).

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

Reduction translation

```
+/2 3⍴⍳6    ⍝ sum each row
```

$\Downarrow$

```
reduce(addi, reshape([2,3], iota(6)))
```

It still holds that: *Most APL primitives are defined for a specific argument rank k, but in the case it is applied to any array with a rank higher than k it will be applied independently to each rank-k subarray.*

Built-ins

APL	*op*(*s*)		$\mathrm{TySc}(op)$
	`addi, ...`	:	$\mathtt{int} \to \mathtt{int} \to \mathtt{int}$
	`addd, ...`	:	$\mathtt{double} \to \mathtt{double} \to \mathtt{double}$
⍳	`iota`	:	$\mathtt{int} \to [\mathtt{int}]^1$
¨	`each`	:	$\forall\alpha\beta\gamma.(\alpha \to \beta) \to [\alpha]^\gamma \to [\beta]^\gamma$
	`zipWith`	:	$\forall\alpha_1\alpha_2\beta\gamma.(\alpha_1 \to \alpha_2 \to \beta) \to [\alpha_1]^\gamma \to [\alpha_2]^\gamma \to [\beta]^\gamma$
/	`reduce`	:	$\forall\alpha\gamma.(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha]^{1+\gamma} \to [\alpha]^\gamma$
/	`compress`	:	$\forall\alpha\gamma.[\mathtt{bool}]^\gamma \to [\alpha]^\gamma \to [\alpha]^\gamma$
/	`replicate`	:	$\forall\alpha\gamma.[\mathtt{int}]^\gamma \to [\alpha]^\gamma \to [\alpha]^\gamma$
\	`scan`	:	$\forall\alpha\gamma.(\alpha \to \alpha \to \alpha) \to [\alpha]^\gamma \to [\alpha]^\gamma$
⍴	`shape`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to \langle\mathtt{int}\rangle^\gamma$
⍴	`reshape`	:	$\forall\alpha\gamma\gamma'.\langle\mathtt{int}\rangle^{\gamma'} \to \alpha \to [\alpha]^\gamma \to [\alpha]^{\gamma'}$
⌽	`reverse`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^\gamma$
⌽	`rotate`	:	$\forall\alpha\gamma.\mathtt{int} \to [\alpha]^\gamma \to [\alpha]^\gamma$
⍉	`transp`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^\gamma$
⍉	`transp2`	:	$\forall\alpha\gamma.\langle\mathtt{int}\rangle^\gamma \to [\alpha]^\gamma \to [\alpha]^\gamma$
↑	`take`	:	$\forall\alpha\gamma.\mathtt{int} \to \alpha \to [\alpha]^\gamma \to [\alpha]^\gamma$
↓	`drop`	:	$\forall\alpha\gamma.\mathtt{int} \to [\alpha]^\gamma \to [\alpha]^\gamma$
,	`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`	:	$\forall\alpha\gamma.[\alpha]^{\gamma+1} \to [\alpha]^\gamma \to [\alpha]^{\gamma+1}$

(incomplete list)

Shape types

We need to know the length of the shape-vector, to be able to infer the rank of the resulting array of a reshape!

APL	*op*(*s*)		$\mathrm{TySc}(\mathit{op})$
ρ	`shape`	:	$\forall \alpha\gamma.[\alpha]^{\gamma} \to \langle \texttt{int} \rangle^{\gamma}$
ρ	`reshape`	:	$\forall \alpha\gamma\gamma'.\langle \texttt{int} \rangle^{\gamma'} \to \alpha \to [\alpha]^{\gamma} \to [\alpha]^{\gamma'}$

- $\langle \texttt{int} \rangle^{\gamma}$ is a length γ integer vector
- $[\alpha]^{\gamma}$ is an array with rank γ

Limitation wrt. APL: We must know the length of the shape-vector statically, e.g. it cannot be the result of a filter.

Type system

$$
\begin{array}{lr}
\kappa ::= \texttt{int} \mid \texttt{double} \mid \texttt{bool} \mid \texttt{char} \mid \alpha & \text{(base types)} \\
\rho ::= i \mid \gamma \mid \rho + \rho' & \text{(shape types)} \\
\tau ::= [\kappa]^{\rho} \mid \langle\kappa\rangle^{\rho} \mid \mathrm{S}_{\kappa}(\rho) \mid \mathrm{SV}_{\kappa}(\rho) & \\
\quad \mid \tau \to \tau' & \text{(types)} \\
\sigma ::= \forall\vec{\alpha}\vec{\gamma}.\tau & \text{(type schemes)}
\end{array}
$$

- Shape types are tree structured to support `drop` and `catenate` on vectors (unlike Accelerate)
- $\mathrm{S}_{\texttt{int}}(\rho)$ is the singleton integer ρ (rank-0)
- $\mathrm{SV}_{\texttt{int}}(\rho)$ is a singleton integer vector with element ρ (rank-1)
- We often write κ instead of $[\kappa]^0$

Shape operations

We need to be able to calculate on shapes, retaining length-information.

APL	*op*(*s*)		TySc(*op*)
ρ	`shapeV`	:	$\forall\alpha\gamma.\langle\alpha\rangle^{\gamma} \to \mathrm{SV}_{\mathtt{int}}(\gamma)$
↑	`takeV`	:	$\forall\alpha\gamma.\mathrm{S}_{\mathtt{int}}(\gamma) \to [\alpha]^{1} \to \langle\alpha\rangle^{\gamma}$
↓	`dropV`	:	$\forall\alpha\gamma\gamma'.\mathrm{S}_{\mathtt{int}}(\gamma) \to \langle\alpha\rangle^{(\gamma+\gamma')} \to \langle\alpha\rangle^{\gamma'}$
⍳	`iotaV`	:	$\forall\gamma.\mathrm{S}_{\mathtt{int}}(\gamma) \to \langle\mathtt{int}\rangle^{\gamma}$
⌽	`rotateV`	:	$\forall\alpha\gamma.\langle\alpha\rangle^{\gamma} \to \langle\alpha\rangle^{\gamma}$
,	`catV`	:	$\forall\alpha\gamma\gamma'.\langle\alpha\rangle^{\gamma} \to \langle\alpha\rangle^{\gamma'} \to \langle\alpha\rangle^{(\gamma+\gamma')}$

(incomplete list)

Subtyping rules

We might know the vector sizes or integer values statically, but want to use them where that information is not needed:

```
let v0:<int>100 = iotaV(100) in
reduce(addi,0,v0)
```

To make the singleton integers and vectors with known length compatible with functions taking general arrays, we add subtyping:

$$\frac{}{\tau \subseteq \tau} \qquad \frac{\tau_1 \subseteq \tau_2 \quad \tau_2 \subseteq \tau_3}{\tau_1 \subseteq \tau_3}$$

$$\frac{}{\langle\kappa\rangle^\rho \subseteq [\kappa]^1} \qquad \frac{}{\mathrm{S}_\kappa(\rho) \subseteq [\kappa]^0} \qquad \frac{}{\mathrm{SV}_\kappa(\rho) \subseteq \langle\kappa\rangle^1}$$

Example: APL → TAIL

```
diff ← {1↓ω-¯1⌽ω}
signal ← {¯50⌈50⌊50×(diff 0,ω)÷0.01+ω}
+/ signal ⍳ 100
```

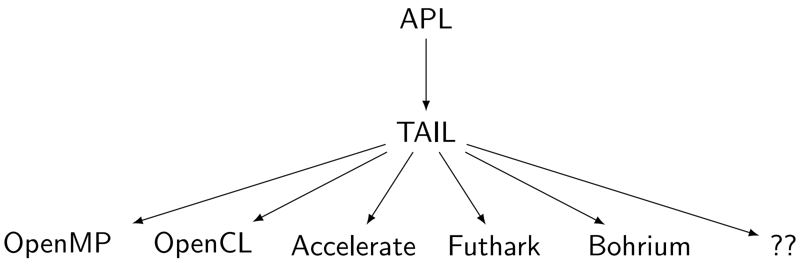
Example: APL → TAIL

```
diff ← {1↓ω-¯1⌽ω}
signal ← {¯50⌈50⌊50×(diff 0,ω)÷0.01+ω}
+/ signal ⍳ 100
```

⇓

```
let v0:<int>100 = iotaV(100) in
let v3:<int>101 = consV(0,v0) in
reduce(addd,0.00,
 each(fn v11:[double]0 => maxd(~50.00,v11),
  each(fn v10:[double]0 => mind(50.00,v10),
   each(fn v9:[double]0 => muld(50.00,v9),
    zipWith(divd,
     each(i2d,
       drop(1,zipWith(subi,v3,rotateV(~1,v3)))),
     eachV(fn v2:[double]0 => addd(0.01,v2),
       eachV(i2d,v0)))))))
```

APL $\rightarrow$ TAIL $\rightarrow$?



Why targeting Accelerate?

- Seemed like an obvious choice given the similarities with TAIL
- The Accelerate people had shown interest
- Using their segmented reductions and scans, we could potentially also perform NESL-like flattening and thus allow nested computations.

A lot of hurdles came up, mostly because the EDSL-nature of Accelerate. More on that later!

Example: TAIL → Accelerate

```
module Main where
import qualified Prelude as P
import Prelude ((+), (-), (*), (/))
import Data.Array.Accelerate
import qualified Data.Array.Accelerate.CUDA as Backend
import qualified APLAcc.Primitives as Prim

program :: Acc (Scalar P.Double)
program
  = let v0 = Prim.iotaV 100 :: Acc (Array DIM1 P.Int) in
      let v3
            = Prim.consV (constant (0 :: P.Int)) v0 :: Acc (Array DIM1 P.Int)
        in
        Prim.reduce (+) (constant (0.0 :: P.Double))
          (Prim.each (\ v11 -> P.max (constant (-50.0 :: P.Double)) v11)
             (Prim.each (\ v10 -> P.min (constant (50.0 :: P.Double)) v10)
                (Prim.each (\ v9 -> constant (50.0 :: P.Double) * v9)
                   (Prim.zipWith (/)
                      (Prim.each Prim.i2d
                         (Prim.drop (constant (1 :: P.Int))
                            (Prim.zipWith (-) v3
                               (Prim.transp
                                  (Prim.rotateV (constant (-1 :: P.Int)) (Prim.transp v3))))))
                      (Prim.eachV (\ v2 -> constant (1.0e-2 :: P.Double) + v2)
                         (Prim.eachV Prim.i2d v0))))))
main = P.print (Backend.run program)
```

Benchmarks

Benchmark	**Problem size**	**TAIL C**	**TAIL Acc**
Integral	N $=$ 10,000,000	46.90	3.10
Signal	N $=$ 50,000,000	209.03	16.1
Game-of-Life	40×40, N $=$ 2,000	28.70	2.30
Easter	N $=$ 3,000	33.96	-
Black-Scholes	N $=$ 10,000	54.0	-
Sobol MC π	N $=$ 10,000,000	4881.30	2430.30
HotSpot	1024×1024, N $=$ 360	6072.93	2.03

- Timings in milliseconds. Averages over 30 executions.

Difficulties using Accelerate

(I might be too honest here!)

- No easy to target AST-representation (yet) and generating Haskell code is not ideal
- Shapes in Accelerate have the outer-most dimension placed innermost in the the Shape-list. Making certain operations difficult to implement efficiently (e.g. vertical rotation)
- When using Accelerate looping constructs (e.g. `awhile`) it seems that sharing is not recovered (e.g. let bound variables outside the loop are inlined, into the loop)
- No real cost-model. We are unable to reason about memory layout (e.g. after a transposition).
- No control over memory allocations
- Hard to debug
- Hard to benchmark
- No mutable array updates

What I'm working on while at Chalmers

APL

$\Downarrow$

TAIL $\quad \Rightarrow \quad$ Accelerate

$\Downarrow$

...

$\Downarrow$

...

$\Downarrow$

$\boxed{\text{Some new functional low-level GPU intermediate language}}$

$\Downarrow$

OpenCL/CUDA

Requirements

- Ability to optimise
  - Consistent cost-model
  - Control over memory allocations
  - Control over when fusion/materialization occurs
- (Array updates in sequential code of kernel bodies)

Latest addition: Type annotations in APL code

```
equal ← { ∧/,α=ω }

pi ← { 4×(+/1>+/(?ω 2ρ0)*2)÷ω }

tc ← { ({ω∨ω∨.∧ω} ⍣ ≡) ω }
```

Latest addition: Type annotations in APL code

```
⍝ equal : [a]r → [a]r → bool
equal ← { ∧/,⍺=⍵ }

⍝ pi : int → double
pi ← { 4×(+/1>(+/(?⍵ 2⍴0)*2)*÷2)÷⍵ }

⍝ [bool]2 → [bool]2
tc ← { ({⍵∨⍵∨.∧⍵} ⍣ ≡) ⍵ }
```

Disclosure: still pretty buggy

Benefits of type annotations

- Easier debugging, through improved error messages
- Machine checked documentation
- A necessity for compiling certain programs - even parsing APL is undecidable!
- Will allow for introducing dependent types, with user-written types. (e.g. tracking complete shape information, not just ranks)
- First steps towards a module system for APL, supporting separate compilation of modules.

Other future work on TAIL

- Annotations for GPU vs. CPU execution
- Other annotations controlling code generation (not clear what they should be yet)
- Ability to drop down to a lower level (like inline assembler)
- Tracking complete shape information, using dependent types (like QUBE)
- Region-inference for memory management
- Additional primitives
- Integration with real APL interpreter (Dyalog)

References

Compiling a Subset of APL Into a Typed Intermediate Language.
Martin Elsman and Martin Dybdal, 2014
ARRAY'14

Compiling APL to Accelerate through a Typed Array Intermediate Language
Michael Budde, Martin Dybdal and Martin Elsman, 2014
ARRAY'15

Accelerating Haskell array codes with multicore GPUs
MMT Chakravarty, G Keller, S Lee, TL McDonell, V Grover, 2011
DAMP'11

APEX: The APL Parallel Executor
Robert Bernecky, 1997
Master Thesis

QUBE - Array Programming with Dependent Types
Kai Trojahner, 2011
Ph.D. thesis

Questions?