Typed Array Intermediate Language

Martin Dybdal

University of Copenhagen

dybber@dybber.dk

10. december 2014

Goals

- GPU programming for APL fingers
- Develop backend technology independently of APL. Other frontends could be J, K, some new Haskell vector-library or a NumPy/SciPy clone.
- 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)
- 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*

- **Many existing programs/benchmarks**
  - We don't have to write our own
  - Writing the benchmarks ourselves might not represent how it will be used in a production environment

Overview

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

TAIL

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

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, as it might lead to nested parallelism:

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 any array with *rank* > 0.

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

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.*

Shape types

When reshaping an array and the length of the shape-vector is statically known, we will always know the shape of the resulting array.

$$\texttt{reshape} : \forall \alpha \gamma \gamma'. \langle \texttt{int} \rangle^{\gamma'} \to \alpha \to [\alpha]^{\gamma} \to [\alpha]^{\gamma'}$$

- $\langle \texttt{int} \rangle^{\gamma'}$ 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

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

- Shape types are tree structured to support `drop` and `catenate`
- $\mathrm{S}_{\texttt{int}}(\rho)$ is the singleton integer ρ
- We sometimes write κ instead of $[\kappa]^0$

Shape operations

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

(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:

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

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

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

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

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)))))))
```

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)
```

User requirements

- Interpreted environment (e.g. APL, MATLAB, NumPy, R)
  - implies dynamic compilation (JIT)
  - implies dynamic garbage collection
- Ability to optimise
  - Consistent cost-model
  - Dropping down to underlying language (e.g. handwritten kernels from CUBLAS)
- Enough primitives
  - We can never cover all of APL
  - As a baseline we hope to support enough APL primitives to make most NumPy/SciPy primitives expressible.
- Optimised idioms (e.g. 100x100 identity matrix: $(\iota 100)\circ.{=}(\iota 100)$, should be represented as a sparse matrix)

Next steps

- Frontend
  - Add array indexing
  - Support DNA-application from Dyalog
  - Support benchmarks from various old APL-papers
- Accelerate backend
  - Convert TAIL shapes to Accelerate shapes correctly
  - Type-checker targetting their HOAS representation
  - Mersenne Twister in Accelerate
  - Support more primitives

Further in the future

- Bohrium/SNESL/Futhark backend
- Type annotations in APL (lightweight dependent types?)
- JIT compilation
- Support nested arrays (flattening or through SNESL?)
- Boolean array encode/decode?

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 IL
Michael Budde, 2014
7.5 ECTS project

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

Questions?