Controlled Array Fusion using Linear Types

Josef Svenningsson
Chalmers University of Technology, Gothenburg, Sweden

Joint work with Jean-Philippe Bernardy

Edinburgh University 2014-11-18

Parallel Functional Arrays

Pull Arrays

```haskell
data Pull a = Pull (Int -> a) Int
```

- Each element is computed independently
  - The consumer decide how to schedule the computations
  - Makes it easy to parallelize
- Well known, used in many different libraries and formalisms

Pull arrays

Some example functions

```
map :: (a -> b) -> Pull a -> Pull b
map f (Pull ixf l) = Pull (f. ixf) l

sum :: Num a => Pull a -> a
sum (Pull ixf l)
    = forLoop l 0 (\i sum ->
            sum + ixf i
      )

zipWith :: (a -> b -> c) -> Pull a -> Pull b -> Pull c
zipWith f (Pull ixfa la) (Pull ixfb lb)
   = Pull (\i -> f (ixfa i) (ixfb i))
              (min la lb)
```

Pull array fusion

Scalar product:

```
scProd :: Num a => Pull a -> Pull a -> a
scProd a b = sum (zipWith (*) a b)
```

Fusion:

```
sum (zipWith (*) (Pull ixfa la) (Pull ixfb lb)) =

sum (Pull (\i -> ixfa i * ixf b) (min la lb)) =

forLoop (min la lb) 0 (\i sum ->
  sum + (ixfa i * ixfb b)
)
```

Push Arrays

```
data Push a = Push ((Int -> a -> M ()) -> M ()) Int
```

- `M` is some monad which provides:
  - mutable updates
  - parallelism
- Intuition:
  - Push arrays are programs which write an array to memory,
  - Parameterized by how to write each element to memory
- The producer decides the scheduling order, consumers must be able to handle any order

Push arrays

`Push` arrays solve the shortcomings of `Pull` arrays

- Provides efficient concatenation

```
Push p l ++ Push q m = Push r (l+m)
  where r w = do p w
                 q (\i a -> w (i+l) a)
```

- Can write several elements at once

```
dup (Push p l) = Push q (2*l)
  where q w = p (\i a -> w (2*i) a >> w (2*i+1) a)
```

- Have the same strong fusion guarantees

Duality of Push and Pull

	Pull	Push
Permutations	Ok	Ok
Functor	Ok	Ok
Splitting	Ok	Nope
Zipping	Ok	Nope
Concatenating	Nope	Ok
Multiple writes in loop body	Nope	Ok

Ok Means operations are *efficiently* implementable.

Nope Means I'm fairly sure there are no *efficient* implementations.

Combining Pull and Push

- It is efficient to convert from Pull arrays to Push arrays
  It means coming up with a schedule for the Push array.

```
pullToPush :: Pull a -> Push a
pullToPush (Pull ixf l) = Push f l
  where f k =  parM l $ \i ->
                 k (ixf i) i
```

- Converting from Push array to Pull arrays requires memory
  Goes from a completely scheduled representation to a random access representation.

```
force :: Push a -> M (Pull a)
force (Push f l) =
  do marr <- newArray_ (0,l-1)
     f (\a i -> writeArray marr i a)
     arr <- freeze marr
     return (Pull (\i -> arr!i) l)
```

Problems

- Pull and Push arrays seem to be dual.
  - Are they actual duals?
  - There is no way to profit from the fact that they are duals in System F.
- We would like to ensure that the function is used at most one on each index to avoid recomputation.

  ```
  Pull (Int -> a) Int
  ```
- Each memory location should be written to exactly once.
  - No race conditions
  - No undefined elements

  ```
  Push ((Int -> a -> M ()) -> M ()) Int
  ```
- Hard to guarantee fusion in a turing complete language.

Classical Linear Logic

Judgments

We use one-sided judgments

- Assumptions on the left, terms on the right

$$\frac{}{x : A, y : A^{\perp} \vdash x \leftrightarrow y}\ \mathrm{Ax}$$

- Akin to Continuation Passing Style: only hypotheses, no result type

Types

$A \oplus B$	$A^\perp \,\&\, B^\perp$	additives
0	$\top$	additive units
$A \otimes B$	$A^\perp \parr B^\perp$	multiplicatives
1	$\bot$	multiplicative units
α	$\alpha^\perp$	atoms

Duality

- In our setting we only need elimination rules.
- Introduction rules are simply elimination rules for the dual construct.

Additive fragment

$$\frac{\Gamma, x : A \vdash a}{\Gamma, z : A \& B \vdash \text{let inl } x = z; a}$$

$$\frac{\Gamma, x : B \vdash a}{\Gamma, z : A \& B \vdash \text{let inr } x = z; a}$$

$$\frac{\Gamma, x : A, \Delta \vdash a \qquad \Gamma, y : B, \Delta \vdash b}{\Gamma, z : A \oplus B, \Delta \vdash \text{case } z \text{ of} \{\text{inl } x \mapsto a; \text{inr } y \mapsto b\}}$$

Multiplicative fragment

$$\frac{\Gamma, x : A, y : B \vdash a}{\Gamma, z : A \otimes B \vdash \text{let } x, y = z; a}$$

$$\frac{\Gamma, x : A \vdash a \qquad y : B, \Delta \vdash b}{\Gamma, z : A \parr B, \Delta \vdash \text{connect } z \text{ to}\{x \mapsto a; y \mapsto b\}}$$

Semantically, we consider par to introduce parallelism.

Arrays in Linear Logic

Types

We introduce two new types for arrays.

$$\begin{array}{cc} \bigotimes_n A & \text{Tensor arrays} \\ \bigparr_n A^\perp & \text{Par arrays} \end{array}$$

Intuition:

- Tensor arrays corresponds to Pull arrays
- Par arrays corresponds to Push arrays

Judgments

We extend judgments to be able to talk about n hypotheses simultaneously, without necessarily knowing n.

$$x : A^n \vdash a$$

x is really a family of variables, but we can think of it as referring to all hypotheses.

Rules

Eliminating tensor arrays means splitting it into n different assumptions

$$\frac{\Gamma, x : A^m \vdash a}{\Gamma, z : \bigotimes_m A \vdash \text{let } x = \text{slice } z; a}$$

Eliminating par arrays means running two different programs on two different parts of the array. Amounts to efficient concatenation.

$$\frac{\Gamma, x : A \vdash a \qquad \Delta, y : A \vdash b}{\Gamma^n, \Delta^m, z : \bigparr_{n+m} A \vdash \text{coslice } z\{x \mapsto_n a; y \mapsto_m b\}}$$

Semantically, the par rule also introduces parallelism.

Cut

$$\frac{\Gamma, x : A^n \vdash a \qquad y : A^\perp, \Delta \vdash b}{\Gamma, \Delta^n \vdash \mathrm{cut}\{x : A^n \mapsto a; y : A^\perp \mapsto b\}}$$

- Semantically, cut introduces allocation. If A is an array then the whole array is allocated.
- In order to achieve fusion we need to eliminate cuts

Writing programs in CLL

As we're used to program in some form of typed lambda calculus we may think of programs with types along the following:

$A \multimap B \multimap \cdots \multimap R$

Writing programs in CLL

As we're used to program in some form of typed lambda calculus we may think of programs with types along the following:

$A \multimap B \multimap \cdots \multimap R$

There are no result types in CLL. Instead we eliminate the dual of the result.

$x : A, y : B, \ldots, r : R^{\perp} \vdash a$

Example programs: map

Mapping a function over Pull arrays

$$xs : \bigotimes_n A, ys : \bigparr_n B^{\perp} \vdash \\ \text{let } xs_i = \text{slice } xs;\text{coslice } ys\{ys_i \mapsto_n f[xs_i, ys_i]\}$$

- The same program works just as well for mapping over a Push array

Example programs: zip

Zipping two arrays together using a function f.

$$\begin{array}{l} xs : \bigotimes_n A, ys : \bigotimes_n A, zs : \bigparr_n C^\perp \vdash \\ \text{let } xs_i = \text{slice } xs; \text{let } ys_i = \text{slice } ys; \\ \text{coslice } zs\{zs_i \mapsto_n f[xs_i, ys_i, zs_i]\} \end{array}$$

Folds

Folding using an associative operator:

$$\frac{u : B^{\perp}, \Delta \vdash a \quad x : B, y : B, w : B^{\perp} \vdash b \quad v : B^{\perp} \vdash e \quad z : B, \Gamma \vdash d}{\Gamma, \Delta^{n} \vdash \mathsf{foldmap}_{n}\ \Delta\{u, \Delta \mapsto a; v \mapsto e; x, y, w \mapsto b; z \mapsto d\}}\ \text{FOLD}$$

Example program: sum

Using fold we can write a program to sum the elements of an array:

$$
\begin{array}{l}
(+) : A \multimap A \multimap A, 0 : A; xs : \bigotimes_n A, r : A^{\perp} \vdash \\
\text{let } xs_i = \text{slice } xs; \\
\text{foldmap}_n\ xs_i \\
\quad \{x, xs_i \mapsto xs_i \leftrightarrow x;\ z \mapsto \mathit{0}[z];\ a, b, c \mapsto (+)[a, b, c];\ s \mapsto r \leftrightarrow s\}
\end{array}
$$

Putting it together: scalar product

Using map, zip and sum it is possible to write the scalar product.

$$
\begin{array}{l}
0 : A, (+) : A \multimap A \multimap A, (*) : A \multimap A \multimap A; xs : \bigotimes_n A, ys : \bigotimes_n A, r : A^{\perp} \vdash \\
\text{let } xs_i = \text{slice } xs; \\
\text{cut}\{v : \mathcal{R}_n A^{\perp} \mapsto \text{let } ys_i = \text{slice } ys; \text{coslice } v\{v_i \mapsto_n (*)[xs_i, ys_i, v_i]\} \\
\quad w : \bigotimes_n A \mapsto sum[w, r]\}
\end{array}
$$

Converting from Pull to Push

In order to be able to convert from Pull to Push we need to introduce two new terms:

$$\frac{\Gamma \vdash a \qquad \Delta \vdash b}{\Gamma, \Delta \vdash \mathsf{mix}\{a; b\}}\ \textsc{Mix} \qquad\qquad \frac{}{\vdash \mathsf{halt}}\ \textsc{Halt}$$

These are standard extensions to CLL and preserves cut-elimination.

Converting from Pull to Push

With the new extensions we can write the following program which converts from Pull to Push.

$$
\begin{aligned}
&\text{let } x_i = \text{slice } x; \text{let } y_i = \text{slice } y;\\
&\text{foldmap}_n\ x_i, y_i \{ a, x_i, y_i \mapsto \text{let } \diamond = a; x_i \leftrightarrow y_i\\
&\qquad\qquad\qquad z \mapsto \text{let } \diamond = z; \text{halt}\\
&\qquad\qquad\qquad l, r, b \mapsto \text{mix}\{\text{yield to } l;\ r \leftrightarrow b\}\\
&\qquad\qquad\qquad y \mapsto \text{yield to } y\}
\end{aligned}
$$

Converting from Push to Pull

- Recall that in the Functional Programming variant we could convert from Push to Pull only by means of allocating to memory.
- We would like to have the same possibility to express allocation in Linear Logic.

Converting from Push to Pull

We introduce the following rule

$$\frac{\Gamma, x : {A^{\perp}}^{n} \vdash a \qquad y : A^{n}, \Delta \vdash b}{\Gamma, \Delta \vdash \mathsf{sync}\{x : {A^{\perp}}^{n} \mapsto a; y : A^{n} \mapsto b\}}\ \textsc{Sync}$$

- This rule is akin to the n-cut rule in Linear Logic
- Unsound in general
- We require that communication in A is unidirectional i.e. all positive or negative types.
  - We say that A is a data type

Converting from Push to Pull

Now we can implement the conversion from Push to Pull:

$$
\begin{aligned}
&xs : \bigparr_n A, ys : \bigparr_n A^{\perp} \vdash \\
&\mathsf{sync}\{z : {A^{\perp}}^n \mapsto \mathsf{coslice}\ xs\{xs_i \mapsto_n z \leftrightarrow xs_i\} \\
&\qquad\quad \bar{z} : A^n \mapsto \mathsf{coslice}\ ys\{ys_i \mapsto_n \bar{z} \leftrightarrow ys_i\}\}
\end{aligned}
$$

Final example: FFT

The inner loop of an FFT:

$$
\begin{array}{l}
i : \bigotimes_{2n} C, o : \bigparr_{2n} C^{\perp} \vdash \\
\text{let } i_i = \text{slice } i; \text{let } x, y = \text{split}_n\, i_i; \\
\text{sync}\{v : {C^{\perp}}^{2n} \mapsto \text{let } x, y = \text{split}_n\, v; \\
\qquad \text{foldmap}_n\, x, y, x, y\{ \\
\qquad\qquad a, x, y, x, y \mapsto \text{let } \diamond = a; \mathit{bff}[x, y, x, y] \\
\qquad\qquad z \mapsto \text{let } \diamond = z; \text{halt} \\
\qquad\qquad l, r, b \mapsto \text{mix}\{\text{yield to } l;\ r \leftrightarrow b\} \\
\qquad\qquad y \mapsto \text{yield to } y\} \\
\quad w : C^{2n} \mapsto \text{coslice } o\{o_i \mapsto_{2n} w \leftrightarrow o_i\}\}
\end{array}
$$

Cut elimination and Fusion

Theorem

Every given instance of cut *can be eliminated.*

This means that we can fuse our programs to be free from allocation (modulo sync)

Size of a type

Definition

We define the size $|A|$ of a type A as follows:

$$
\begin{aligned}
|A \oplus B| &= 1 + max(|A|, |B|) & \qquad |0| &= 0 \\
|A \otimes B| &= |A| + |B| & |1| &= 0 \\
|\bigotimes_n A| &= n|A| & |P^\perp| &= |P|
\end{aligned}
$$

Cost of a program

Definition

We define the cost $|a|$ of a program a as follows

$$\begin{aligned}
|x \leftrightarrow y| &= |A| \\
|\text{cut}\{x : A^n \mapsto a; y : A^{\perp} \mapsto b\}| &= |A| + |a| + n|b| \\
|\text{mix}\{a; b\}| &= 1 + |a| + |b| \\
|\text{yield to } x| &= 1 \\
|\text{let } \diamond = x; a| &= 1 + |a| \\
|\text{halt}| &= 1 \\
|\text{dump } \Gamma \text{ in } x| &= 1 \\
|\text{let } x, y = z; a| &= 1 + |a| \\
|\text{connect } z \text{ to}\{x \mapsto a; y \mapsto b\}| &= 1 + |a| + |b| \\
|\text{case } z \text{ of}\{\text{inl } x \mapsto a; \text{inr } y \mapsto b\}| &= 1 + max(|a|, |b|)
\end{aligned}$$

Cost of a program

Definition

Continued..

$$
\begin{aligned}
|\text{let inl } x = z; a| &= 1 + |a| \\
|\text{let inr } x = z; a| &= 1 + |a| \\
|\text{let } x, y = \text{split}_n\, z; a| &= 1 + |a| \\
|\text{let } x = \text{slice } z; a| &= 1 + |a| \\
|\text{coslice } z\{x \mapsto_n a; y \mapsto_m b\}| &= 1 + n|a| + m|b| \\
|\text{sync}\{x : {A^{\perp}}^n \mapsto a; y : A^n \mapsto b\}| &= 10 + 0 + n|A| + |a| + |b|
\end{aligned}
$$

$$
\begin{aligned}
&|\text{foldmap}_n\ \Delta\{u, \Delta \mapsto a; v \mapsto e; x, y, w \mapsto b; z \mapsto d\}| \\
&\qquad = 1 + |e| + |d| + n|a| + (n-1)(5 + |b|)
\end{aligned}
$$

Guaranteed improvement

Theorem

For any two programs a and b communicating via type A:

$$|\mathrm{fuse}\{x : A^n \mapsto a; y : A^\perp \mapsto b\}| \leq |\mathrm{cut}\{x : A^n \mapsto a; y : A^\perp \mapsto b\}|$$

Using `fuse` means performing cut-elimination.

Notes on improvement

Fusion can make parallel runtime worse: `cut` introduces parallelism and removing it will decrease the amount of parallelism in the program.

Recovering Pull and Push arrays

It is possible to translate CLL into System F using a double negation translation, with a return type r.

$[\![A^{\perp}]\!] = [\![A]\!] \rightarrow r$

Recovering Pull and Push arrays

It is possible to translate CLL into System F using a double negation translation, with a return type r.

$$[\![A^{\perp}]\!] = [\![A]\!] \rightarrow r$$

We can choose to translate tensor arrays as follows:

$$[\![\bigotimes_n A]\!] = \mathsf{Int} \rightarrow [\![A]\!]$$

Recovering Pull and Push arrays

It is possible to translate CLL into System F using a double negation translation, with a return type r.

$[\![A^\perp]\!] = [\![A]\!] \to r$

We can choose to translate tensor arrays as follows:

$[\![\bigotimes_n A]\!] = \mathsf{Int} \to [\![A]\!]$

That gives us Pull arrays (modulo the length):

```
data Pull a = Pull (Int -> a) Int
```

Recovering Pull and Push arrays

The translation of par arrays can be derived as follows:

$$
\begin{aligned}
&[\![\parr_n A]\!] = \\
&[\![(\parr_n A)^{\perp\perp}]\!] = \\
&[\![(\bigotimes_n A^{\perp})^{\perp}]\!] = \\
&[\![\bigotimes_n A^{\perp}]\!] \to r = \\
&(\mathsf{Int} \to [\![A^{\perp}]\!]) \to r = \\
&(\mathsf{Int} \to [\![A]\!] \to r) \to r
\end{aligned}
$$

Recovering Pull and Push arrays

The translation of par arrays can be derived as follows:

$$
\begin{aligned}
&[\![\parr_n A]\!] = \\
&[\![(\parr_n A)^{\perp\perp}]\!] = \\
&[\![(\textstyle\bigotimes_n A^{\perp})^{\perp}]\!] = \\
&[\![\textstyle\bigotimes_n A^{\perp}]\!] \to r = \\
&(\mathsf{Int} \to [\![A^{\perp}]\!]) \to r = \\
&(\mathsf{Int} \to [\![A]\!] \to r) \to r
\end{aligned}
$$

Picking $r = \mathsf{M}\ ()$ gives us Push arrays.

```
data Push a = Push ((Int -> a -> M ()) -> M ()) Int
```

Summary

- A Curry-Howard correspondence for functional parallel arrays in Classical Linear Logic
- Cut-elimination = Fusion
  - Fusion is guaranteed
  - Cost is proven to decrease

Notes

- Easy to add quantification
  - The cost measure on programs needs some care to get right
- Exponentials (!A) can be added, but we lose either of these things:
  - No guaranteed improvement
  - No guaranteed fusion

Ongoing and Future work

- An implementation which generates efficient code
  - We're currently targeting OpenMP
- Extend to sequential arrays
  - A self-dual connective
  - Expressing scans etc.
- Polarized version
  - Help make sense of the `sync` rule