Composable Efficient Array Computations Using Linear Types

Jean-Philippe Bernardy   Víctor López Juan   Josef Svenningsson

Chalmers University of Technology and University of Gothenburg

bernardy@chalmers.se   victor@lopezjuan.com   josefs@chalmers.se

Abstract

Functional languages excel at describing complex programs by composition of small building blocks. Yet, it can be difficult to predict the performance properties of such compositions. Namely, whether parts of the computation are shared or duplicated is typically left to the compiler to decide heuristically.

We propose the use linear types as a tool to specify the desired behavior (sharing or duplication). This means that the programmer is able to predict the performance of composition solely from the types of the composed functions.

In this paper, we focus on the array-manipulation fragment of such a language. Technically, we extend Girard's linear logic (LL) with vector types and a synchronization primitive. These extensions allow expressing array-processing programs in a compositional fashion.

We illustrate the effectiveness of our language by writing a number of examples in compositional style, and show that they predictably compile to efficient code. Examples include kernels of classical algorithms (FFT, 1-dimensional stencil, QuickHull).

Keywords Array-Programming, Classical Linear Logic

1. Introduction

Consider the `diff` function, which computes the difference between each pair of consecutive elements in a vector. (This operation is useful when approximating derivatives or interpolating polynomials.) A straightforward implementation of `diff` in C is the following:

```
void diff(size_t n, const double a[], double b[]) {
  for(i = 0; i < n−1; i++)
     b[i] = a[i+1] − a[i];
}
```

To compute second order differences (for example, to approximate second derivatives), one can apply `diff` twice:

```
void diff2(size_t n; const double a[], double c[]) {
  double b[] = malloc( sizeof(double) ∗ (n−1) );
  diff(a,b);
  diff(b,c);
}
```

The two runtime instances of `diff` communicate via an intermediate array, which needs to be allocated in full. This allocation is undesirable; a C programmer aiming for performance would not make this allocation, and instead write `diff2` as follows:

```
void diff2_fused(size_t n, const double a[], double c[]) {
  for(i=0; i < n−2; i = i + 1)
     c[i] = a[i + 2] − 2∗a[i+1] + a[i];
}
```

In his famous position paper, Hughes [14] argues that a key advantage of functional programming is that programs can be written by composition of simple building blocks, while retaining good runtime behaviour. Trusting this *composability principle*, we would write diff2 as follows:

```
diff  (x:xs)  = zipWith (−)  (x:xs)  xs
diff  []      = []

diff2  =  diff  .  diff
```

Unfortunately, the above implementation of `diff2` still allocates the intermediate data. (In a lazy language, the intermediate list will be allocated piecewise, but one still pays the overhead of each individual thunk.)

Our aim is to show how the composability principle can be reliably extended to efficient array computations.

An intuitive approach to this goal, used in several functional DSLs [2, 6, 9, 17, 28], is to represent an immutable array of type A by a function $Int \to A$; and deferring the reification of such functions into actual arrays to a later phase. Using such a representation, we can define diff2 as follows:

```
type Array a = Int  →  a
diff ,  diff2  ::  Array a → Array a
diff  f  i  = f  (i  + 1)  − f  i
diff2  =  diff  .  diff
```

The above code does not allocate intermediate data structures. However, this comes at the price of duplicated computation. Indeed, `diff` accesses each index in the array twice; thus, when composing it with itself, the first set of differences will be computed twice. While a sufficently clever compiler may spot the duplication in the later code-generation phases, this leaves the programmer in an uncomfortable situation: the only way to predict the performance behaviour of the generated code is to have an intimate knowledge of the optimisation passes implemented in the current version of the compiler. Hence, in this paper, we will not be satisfied with relying on compiler-specific optimisations. We instead aim to give the programmer strong guarantees on the behaviour of the generated code.

Our contributions are as follows:

- We extend classical linear logic [11] with vector types ($\bigotimes_n A$, $⅋_n A$, $\S_n A$) and a synchronization primitive (Sec. 2.1). The resulting language is functional at its heart, because any computation can be made into a first-class value, passed as an argument, and composed with others. We call this language CLL^n.

- We guarantee that all well-typed function compositions do not allocate intermediate storage. Not even potential closures and thunks will be allocated. Linearity guarantees that the transformation will preserve the computational cost, thus avoiding the need for heuristics (Sec. 3). Yet, programmers are given the opportunity to explicity use allocation primitives, and the types indicate when such allocations are necessary.
- We provide a compiler from CLL^n programs to structured, imperative C code (Sec. 4).
- We implement kernels for FFT, 1-dimensional wave propagation and QuickHull in our language, and show how fusion significantly improves running time (Sec. 5).

2. Our language

In this section, we describe our language, CLL^n. We emphasize that we describe only the core calculus behind a functional language which would eventually be based on it. The syntax and typing rules of this calculus are summarized in Fig. 1.

Issues such as syntactic sugar, type inference, etc. are out of the scope of this paper. Even so, in long code examples, we let ourselves omit braces if they would fully encompass the remainder of the current code block.

2.1 CLL as a programming language

We begin with a quick review of the Linear Logic of Girard [11] (which we abbreviate CLL, for "classical", emphasizing the difference with intuitionistic linear logic). While we are interested only in the computational interpretation of CLL, via the Curry-Howard lens, we still use the logical lexicon where it seems more natural.

As per Curry-Howard, programs are derivations in the logic. The input and output arguments of the program are propositions at the root of the derivation; computation is performed by applying rules, yielding a derivation tree.

The defining characteristic of a linear logic is the lack of a weakening rule (no hypotheses can be discarded), or a contraction rule (conclusions must be proved as many times as they appear). Consequently, in linear logic it is valid to treat arguments and results dually. Technically every type A has a dual $A^\perp$, and returning a result of type A is equivalent to consuming an argument of type $A^\perp$. Further, dualization is involutive: $(A^\perp)^\perp = A$.

Duality makes for an economic design, particularly suitable to a core language, for two reasions. First, there is no need to distinguish inputs from outputs, so both can be written on any one side of the turnstile. Here, we chose the left side, to indicate that they are values to be consumed by the program. Second, it is enough to provide eliminators, as introduction rules can be implemented by dualizing the former. Effectively, duality cuts the number of constructions one has to deal with in half . We make further use of duality via the self-dual sequence-type operator (Sec. 2.4).

Consequently, our typing judgement has the form $\Gamma \vdash a$, where Γ is a context and a is a program discharging all the inputs in Γ. In general, contexts are ranged over by capital Greek letters. We follow the convention that when contexts are mentioned on either side of a comma, they must be disjoint. Contexts are order-agnostic.

2.2 Cut-Elimination as Evaluation

The composition of programs a and b via a type A is represented by cut:

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

As a first approximation, one can computationally interpret the above cut as first running a, storing its result into a memory area large enough for an A, and then running b. However, in CLL the type A may be a complicated protocol, therefore the flow of execution may jump back and forth several times between a and b. Hence, it is useful to think of the above cut as running a and b concurrently (as co-routines). Besides, it should be stressed that the only point of communication between a and b is A, which may be understood as a communication channel, whose ends are x and y.

One can assign a computational interpretation to CLL^n by its cut-elimination procedure: each reduction of a cut rule corresponds to a communication step between programs.

Let us show two examples of evaluation. (Typing derivations for the programs can be recovered from Fig. 1.)

First consider the additives, known as *plus* ($\oplus$) and *with* ($\&$):

$$\begin{aligned}&\mathrm{cut}\{z : A^\perp \& B^\perp \mapsto \mathrm{let\ inl}\ x = z;\ \mathrm{a}\\ &\qquad \bar{z} : A \oplus B \mapsto \mathrm{case}\ \bar{z}\ \mathrm{of}\{\mathrm{inl}\ \bar{x} \mapsto \mathrm{b};\ \mathrm{inr}\ \bar{y} \mapsto \mathrm{c}\}\}\end{aligned}$$

The above program represents the composition of a case statement (on the right) and of the production of a bit of data indicating that the left branch of the case should be chosen (on the left). Producing the bit is implemented by the eliminator of the dual of $A \oplus B$. The reduct is a composition on the left of the rest of the program, and on the right of the chosen branch.

$$\mathrm{cut}\{x : A^\perp \mapsto \mathrm{a};\ \bar{x} : A \mapsto \mathrm{b}\}$$

Consider now the multiplicatives *tensor* ($\otimes$), and *par* ($⅋$):

$$\begin{aligned}&\mathrm{cut}\{\bar{z} : A^\perp ⅋ B^\perp \mapsto \mathrm{connect}\ \bar{z}\ \mathrm{to}\{\bar{x} \mapsto \mathrm{a};\ \bar{y} \mapsto \mathrm{b}\}\\ &\qquad z : A \otimes B \mapsto \mathrm{let}\ x, y = z;\ \mathrm{c}\}\end{aligned}$$

The let instruction decomposes the tensor z and makes its constituents x and y available to the program c. Conversely, connect deals with the dual, by making each of the components available to independent programs a and b. The reduct is two compositions:

$$\mathrm{cut}\{\bar{x} : A^\perp \mapsto \mathrm{a};\ x : A \mapsto \mathrm{cut}\{\bar{y} : B^\perp \mapsto \mathrm{b};\ y : B \mapsto \mathrm{c}\}\}$$

Finally, *linear implication* ($\multimap$) is an arrow connective which models functions in linear logic. To eliminate $A \multimap B$, a value of type A is provided, and a value of type B is consumed, each in a disjoint part of the context. (If they were produced and consumed in the same context, the programmer might violate causality by using the output of the function to produce the input.) In CLL, $A \multimap B \equiv A^\perp ⅋ B$.

2.3 Cut as a Cost-Free Abstraction

The cut-elimination process has the additional property that any individual cut can be eliminated, regardless of the definition of the composed programs (they may themselves contain cuts). In this paper, we define an additional rule (fuse$\{x : A \mapsto \mathrm{a}; y : A^\perp \mapsto \mathrm{b}\}$), logically equivalent to CUT, but which is recursively defined instead of postulated, so it is guaranteed to disappear from the original program (Thm. 1). We call this rule FUSE: indeed, it effectively fuses the two programs that it connects, without any intermediate data structure. Furthermore, in our setting, cut-elimination does not worsen the runtime behaviour of the program (Thm. 2). This property means that the abstraction mechanism provided by fuse (composition and naming of intermediate results) is implemented in a manner that incurs no runtime cost. Technically, the definition of FUSE is based on the syntactic cut-elimination process for CLL, which we extend to arrays. Before discussing fuse systematically (Sec. 3), we introduce our array primitives and showcase their behaviour on some examples.

$$\frac{}{x : A, y : A^{\perp} \vdash x \leftrightarrow y}\ \textsc{Ax} \qquad \frac{\Gamma, x : A^n \vdash \mathrm{a} \qquad \Delta, y : A^{\perp} \vdash \mathrm{b}}{\Gamma, \Delta^n \vdash \mathrm{cut}\{x : A^n \mapsto \mathrm{a}; y : A^{\perp} \mapsto \mathrm{b}\}}\ \textsc{Cut}_n \qquad \frac{\Gamma \vdash \mathrm{a} \qquad \Delta \vdash \mathrm{b}}{\Gamma, \Delta \vdash \mathrm{mix}\{\mathrm{a}; \mathrm{b}\}}\ \textsc{Mix} \qquad \frac{}{x : \bot \vdash \mathrm{yield\ to}\ x}\ \bot$$

$$\frac{\Gamma \vdash \mathrm{a}}{\Gamma, x : 1 \vdash \mathrm{let} \diamond = x; \mathrm{a}}\ 1 \qquad \frac{}{\vdash \mathrm{halt}}\ \textsc{Halt} \qquad \frac{}{\Gamma, x : 0 \vdash \mathrm{dump}\ \Gamma\ \mathrm{in}\ x}\ 0 \qquad \frac{\Gamma, x : A, y : B \vdash \mathrm{a}}{\Gamma, z : A \otimes B \vdash \mathrm{let}\ x, y = z; \mathrm{a}}\ \otimes$$

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

$$\frac{\Gamma, x : A \vdash \mathrm{a}}{\Gamma, z : A \& B \vdash \mathrm{let\ inl}\ x = z; \mathrm{a}}\ \&_1 \qquad \frac{\Gamma, x : B \vdash \mathrm{a}}{\Gamma, z : A \& B \vdash \mathrm{let\ inr}\ x = z; \mathrm{a}}\ \&_2 \qquad \frac{\Gamma, x : A^n, y : A^m \vdash \mathrm{a}}{\Gamma, z : A^{n+m} \vdash \mathrm{let}\ x, y = \mathrm{split}_n\ z; \mathrm{a}}\ \textsc{Split}_n$$

$$\frac{\Gamma, x : A^m \vdash \mathrm{a}}{\Gamma, z : \bigotimes_m A \vdash \mathrm{let}\ x = \mathrm{slice}\ z; \mathrm{a}}\ \bigotimes \qquad \frac{\Gamma, x : A \vdash \mathrm{a} \qquad \Delta, y : A \vdash \mathrm{b}}{\Gamma^n, \Delta^m, z : \bigparr_{n+m} A \vdash \mathrm{coslice}\ z\{x \mapsto_n \mathrm{a}; y \mapsto_m \mathrm{b}\}}\ \bigparr$$

$$\frac{\Gamma, y : B, x : A \vdash \mathrm{a} \qquad \Delta, y : B, x : A \vdash \mathrm{b}}{\Gamma^n, \Delta^m, x_1 : \S_n A, x_2 : \S_m A, y_1 : \S_{n+m} B \vdash \mathrm{traverse}\{y_1\ \mathrm{as}\ y, x_1\ \mathrm{as}\ x \mapsto_n \mathrm{a}; y_1\ \mathrm{as}\ y, x_2\ \mathrm{as}\ x \mapsto_m \mathrm{b}\}}\ \S$$

$$\frac{\Gamma, x : D^{\perp n} \vdash \mathrm{a} \qquad \Delta, y : D^n \vdash \mathrm{b}}{\Gamma, \Delta \vdash \mathrm{sync}\{x : D^{\perp n} \mapsto \mathrm{a}; y : D^n \mapsto \mathrm{b}\}}\ \textsc{sync} \qquad \frac{\Gamma, x : D^{\perp} \vdash \mathrm{a} \qquad \Delta, y : \S_m(D \otimes (D^{\perp} \& 1)) \vdash \mathrm{b}}{\Gamma, \Delta \vdash \mathrm{loop}\{x : D^{\perp} \mapsto \mathrm{a}; y : \S_m(D \otimes (D^{\perp} \& 1)) \mapsto \mathrm{b}\}}\ \textsc{loop}$$

Figure 1. Typing rules. For concision, we show only the binary versions of coslice and traverse, but any arity is valid. The SYNC and LOOP rules are both restricted to data types, and we use the name D to highlight this fact. The rules are explained in pedagogical order in section 2.

$A \oplus B$	$A^{\perp} \& B^{\perp}$	additives
0	$\top$	additive units
$A \otimes B$	$A^{\perp} \parr B^{\perp}$	multiplicatives
1	$\bot$	multiplicative units
$\bigotimes_n A$	$\bigparr_n A^{\perp}$	arrays
$\S_n A$	$\S_n A^{\perp}$	sequential arrays
α	$\alpha^{\perp}$	atoms

Figure 2. List of type connectives. The types in the left column are dual to the types in the right column, and *vice versa*.

2.4 Arrays

While Girard's CLL offers strong runtime guarantees, few useful programs can be implemented directly in it. In this section, we show how to extend CLL with array primitives, and demonstrate that this extension is enough to describe useful programs.

Contexts We first extend the syntax of contexts with special support for n copies of a type. ($\Gamma ::= - \mid \Gamma,\ x : A^n$). (While A^n may be intuitively understood as n copies of A, the binding $x : A^n$ introduces a single variable x. Accessing individual copies is done using special-purpose rules, which we discuss below.) We omit the superscript when it is equal to 1. We also use the shorthand Γ^n for multiplying all superscripts in Γ by n. We stress that the superscript is part of the context, not part of the type; in particular superscripts have no dual.

Tensor Arrays Our first array operator is the n-way generalization of tensor, and written $\bigotimes_n A$. The tensor array eliminator (slice) consumes an array $z : \bigotimes_n A$ and yields n values of type A in the variable x. The continuation program a uses *all* the values in the order it pleases. That is, the consumer of a tensor array decides the order in which the elements of the array are processed.

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

Functional programmers may want to think of $\bigotimes_n A$ as a function from index to A ($Nat \to A$), with an additional linearity constraint.

Par Arrays The dual of the tensor array is the par array, written ($\bigparr_n A$). By duality, the consumer of a par array must be able to consume its elements in any given order. Hence, the eliminator of $\bigparr_n A$ (coslice) must ensure that each element can be handled independently. This is realized by the following typing rule, where the body a has access to a single element of the array, and an nth fraction of the context. At runtime, a will be executed n times.

$$\frac{\Gamma, x : A \vdash \mathrm{a}}{\Gamma^n, z : \bigparr_n A \vdash \mathrm{coslice}\ z\{x \mapsto_n \mathrm{a}\}}\ \bigparr$$

However, all elements need not be processed in the same way. In general the programmer can specify up to n different programs. Here, a is used for elements up to index n, and b for elements after index n.

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

People with functional programming background may want to approximate $\bigparr_n A$ by the type $(Nat \to (A \to \bot)) \to \bot$, with linear constraints. However, dualizing this approximative type does not yield $\bigotimes_n A^{\perp}$, because double-negations can only be removed in a classical setting.

Duals Constructing tensor and par arrays does not require additional language support. We reap the benefits of using CLL as a framework: by duality, slice can be used to construct par arrays (when constructing a par array one can decide the processing order), and coslice can be used to construct tensor arrays (when constructing a par array one must abide to the order imposed by the consumer).

Sequences We have seen that the consumer of a tensor array masters the processing order, while the consumer of a par array is a slave which must respond to any requested order. A third way exists: when the order of processing is fixed once and for all, by the programming language itself. We call such an array a sequence, and write it $\S_n\, A$. The self-duality of the new array operator is reflected in the logic: $(\S_n\, A)^\perp = \S_n\, A^\perp$. As in par arrays, elements from each sequence are processed one at a time, with the the processing of each element possibly depending on its index. In contrast, the advantage of sequences is that any number of such arrays can be processed simultaneously by traversing them in lockstep:

$$\frac{\Gamma, x' : A, y' : B \vdash \mathrm{a}}{\Gamma^n, x : \S_n\, A, y : \S_n\, B \vdash \mathrm{traverse}\{x \text{ as } x', y \text{ as } y' \mapsto_n \mathrm{a}\}}\ \S$$

2.5 Basics: map, zip, saxpy

Equipped with the above, we can implement simple functions such as $\mathrm{map(f)}$, lifting a conversion program $f[x,y]$ from $x : A$ to $y : B$ to tensor arrays. Recall that producing a result of a particular type is equivalent to consuming an argument of the dual of that type. Hence the conversion from $\bigotimes_n A$ to $\bigotimes_n B$ can be implemented by slice-ing $\bigotimes_n A$ and coslice-ing $\parr_n B^\perp$, as follows:

$$\begin{array}{l}
xs : \bigotimes_n A, ys : \parr_n B^\perp \vdash \\
\mathrm{let}\ xs = \mathrm{slice}\ xs;\ \mathrm{coslice}\ ys\{ys \mapsto_n \mathrm{f}[xs, ys]\}
\end{array}$$

(We remark in passing that lifting f between par arrays has the exact same implementation. Also, xs and ys are new variable names, which shadows xs and ys.)

Another implementable function is $\mathrm{zip(f)}$, which combines two tensor arrays indexwise, via a combination function f.

$$\begin{array}{l}
xs : \bigotimes_n A, ys : \bigotimes_n B, zs : \parr_n C^\perp \vdash \\
\mathrm{let}\ xs = \mathrm{slice}\ xs;\ \mathrm{let}\ ys = \mathrm{slice}\ ys; \\
\mathrm{coslice}\ zs\{zs \mapsto_n \mathrm{f}[xs, ys, zs]\}
\end{array}$$

This function serves as a basic building block for many examples presented below. One such example is the so called saxpy procedure, which multiplies the vector x by a scalar a and adds it to the vector y. In a linearly-typed language, we must indicate that we need n copies of a, by wrapping it in a vector. We can implement saxpy by gluing $\mathrm{zip}(+)$ with $\mathrm{zip}(*)$, as follows:

$$\begin{array}{l}
xs : \bigotimes_n A, ys : \bigotimes_n A, a : \bigotimes_n A, rs : \parr_n A^\perp \vdash \\
\mathrm{cut}\{z : \parr_n A^\perp \mapsto \mathrm{let}\ a = \mathrm{slice}\ a;\ \mathrm{let}\ xs = \mathrm{slice}\ xs; \\
\qquad\qquad\qquad\qquad \mathrm{coslice}\ z\{z \mapsto_n (*)[a, xs, z]\} \\
\qquad \bar{z} : \bigotimes_n A \mapsto \mathrm{let}\ \bar{z} = \mathrm{slice}\ \bar{z};\ \mathrm{let}\ ys = \mathrm{slice}\ ys; \\
\qquad\qquad\qquad\qquad \mathrm{coslice}\ rs\{rs \mapsto_n (+)[\bar{z}, ys, rs]\}\}
\end{array}$$

The above implementation uses an intermediate array containing the $a * x$. We can get rid of it by performing fusion:

$$\begin{array}{l}
\mathrm{let}\ a = \mathrm{slice}\ a;\ \mathrm{let}\ xs = \mathrm{slice}\ xs;\ \mathrm{let}\ ys = \mathrm{slice}\ ys; \\
\mathrm{coslice}\ rs\{rs \mapsto_n \\
\qquad\qquad \mathrm{cut}\{\bar{z} : A \mapsto (+)[\bar{z}, ys, rs];\ z : A^\perp \mapsto (*)[a, xs, z]\}\}
\end{array}$$

The above code still has a cut on the type A, because we have assumed that A is abstract. In general, cuts on atomic types remain. Indeed, such types are representable in a single CPU register, and cuts on them correspond to using registers as intermediate storage.

2.6 Higher-Order Functions and Linearity

One may be worried that the above programs disregard higher-order programming. Coming back to the example of map, the function $f : A \multimap B$ ought to be taken as an argument. A remedy exists, but, to preserve linearity, we need n copies of f in the context. The type of $\mathrm{map} : \bigotimes_n (A \multimap B) \multimap \bigotimes_n A \multimap \bigotimes_n B$ is inhabited as follows:

$$\begin{array}{l}
f : \bigotimes_n (A \multimap B), xs : \bigotimes_n A, ys : \parr_n B^\perp \vdash \\
\mathrm{let}\ f = \mathrm{slice}\ f;\ \mathrm{let}\ xs = \mathrm{slice}\ xs; \\
\mathrm{coslice}\ ys \\
\quad \{ys \mapsto_n \mathrm{connect}\ f\ \mathrm{to}\{\tau \mapsto xs \leftrightarrow \tau;\ \sigma \mapsto ys \leftrightarrow \sigma\}\}
\end{array}$$

This type reminds the programmer that f is used n times. This reminder is specially useful if f is the result of, for example, an expensive partial application. In this case, the programmer should consider caching this result to memory (using a cut marked as "not-to-eliminate"), and sharing it across all n copies of f. The tensor array $\bigotimes_n (A \multimap B)$ is not represented in memory as an array; instead, all positions alias to the same value.

While higher-order programming is well supported by our language, we will use the more concise form of the combinators in the upcoming examples.

2.7 Loops

As explained in Sec. 2.4, the producer and the consumer of a sequences agree in advance on a specific processing order. We specifically chose a left-to-right sequential order. This particular order enables operations where the computation on each element of the array depends on the values of some or all of the previous elements. This is embodied in the LOOP rule:

$$\frac{\Gamma, x : D^\perp \vdash \mathrm{a} \qquad \Delta, y : \S_m (D \otimes (D^\perp \,\&\, 1)) \vdash \mathrm{b}}{\Gamma, \Delta \vdash \mathrm{loop}\{x : D^\perp \mapsto \mathrm{a};\ y : \S_m (D \otimes (D^\perp \,\&\, 1)) \mapsto \mathrm{b}\}}\ \textsc{loop}$$

When applying the rule, the programmer choses a type A and a size m. Then, they can provide an initial value of type A, and, sequentially, m functions of type $A \multimap A \oplus 1$. (In many but not necessarily all cases, all these functions have the same implementation.) At each step, the current function will receive a value and may chose to provide a new one (by returning an A), or keep the old one (by returning a unit value). The produced value (starting with the initial one) becomes the input to the next function; the last produced value is discarded. In sum, the LOOP rule produces an obligation of processing values of type A in a left-to-right sequence.

As an example of LOOP, we implement a dot-product function, by composing $\mathrm{zip}(*)$ with a LOOP computing the sum of the result.

$$\begin{array}{l}
\mathrm{fuse}\{\tau : \parr_n \mathbb{R}^\perp \mapsto \tau \leftrightarrow \mathrm{zip}(*)[a, b] \\
\qquad \tau : \bigotimes_n \mathbb{R} \mapsto \\
\qquad\quad \mathrm{let}\ \tau = \mathrm{slice}\ \tau; \\
\qquad\quad \mathrm{loop}\{mw : \mathbb{R}^\perp \mapsto mw \leftrightarrow 0.0 \\
\qquad\qquad\quad mu : \S_{n+1}(\mathbb{R} \otimes (\mathbb{R}^\perp \,\&\, 1)) \mapsto \\
\qquad\qquad\qquad \mathrm{traverse} \\
\qquad\qquad\qquad\quad \{mu \text{ as } mu \mapsto_n \\
\qquad\qquad\qquad\qquad\quad \mathrm{let}\ \tau, \tau = mu;\ \mathrm{let\ inl}\ \tau = \tau;\ \tau \leftrightarrow +[\tau, \tau] \\
\qquad\qquad\qquad\qquad mu \text{ as } mu \mapsto_1 \mathrm{let}\ \tau, \tau = mu;\ \mathrm{let\ inr}\ \tau = \tau; \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{let}\ \diamond = \tau;\ r \leftrightarrow \tau\}\}\}
\end{array}$$

After cut-elimination, a single loop remains.

$$\begin{array}{l}
\mathrm{let}\ a = \mathrm{slice}\ a;\ \mathrm{let}\ b = \mathrm{slice}\ b; \\
\mathrm{loop}\{mw : \mathbb{R}^\perp \mapsto mw \leftrightarrow 0.0 \\
\qquad mu : \S_{n+1}(\mathbb{R} \otimes (\mathbb{R}^\perp \,\&\, 1)) \mapsto \\
\qquad\quad \mathrm{traverse}\{mu \text{ as } mu \mapsto_n \mathrm{let}\ \tau, \tau = mu;\ \mathrm{let\ inl}\ \tau = \tau; \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{cut}\{w : \mathbb{R} \mapsto \tau \leftrightarrow +[\tau, w] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad v : \mathbb{R}^\perp \mapsto v \leftrightarrow *[a, b]\} \\
\qquad\qquad\qquad mu \text{ as } mu \mapsto_1 \mathrm{let}\ \tau, \tau = mu;\ \mathrm{let\ inr}\ \tau = \tau; \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{let}\ \diamond = \tau;\ r \leftrightarrow \tau\}\}
\end{array}$$

2.8 Matrix Multiplication

We can build on dot product to construct the product of a matrix x of size $n \times m$ and a matrix y of size $m \times p$ to obtain a matrix of size $n \times p$. Each element of x is needed p times, each element of y is needed n times. We might then give matrix multiplication the type $\bigotimes_{mnp} A \multimap \bigotimes_{mnp} A \multimap \bigotimes_{np} A$. For a suitable layout of elements in memory, the implementation is straightforward:

$$\begin{array}{l}
xs : \bigotimes_{nmp} A, ys : \bigotimes_{nmp} A, rs : \parr_{np} A^\perp \vdash \\
\mathrm{let}\ xs = \mathrm{slice}\ xs;\ \mathrm{let}\ ys = \mathrm{slice}\ ys; \\
\mathrm{coslice}\ rs\{rs \mapsto_{np} \mathrm{dot\ product}[xs, ys, rs]\}
\end{array}$$

2.9 Conversions between array types

In this section we describe how to convert between various kinds of arrays.

Tensor to Sequence We want to implement a function of type $\bigotimes_n A \multimap \parr_n A$, that is, derive $\bigotimes_n A, \S_n A^\perp \vdash$.

This is implementable directly using slice and $\mathrm{traverse}$:

$$\begin{array}{l}\mathrm{tensorToSequence} \equiv \mathtt{a} : \bigotimes_n A,\ \mathtt{b} : \S_n A^\perp \vdash \\ \textbf{slice}\ \mathtt{a}\ \{\ \mathtt{a} : A^n \mapsto \textbf{traverse}\ \{\ \mathtt{b}\ \textbf{as}\ \mathtt{b'} \mapsto \mathtt{a} \leftrightarrow \mathtt{b'}\ \}\ \}\end{array}$$

Sequence to Par In this case, we want to inhabit $\S_n A \multimap \parr_n A$, or derive $\S_n A, \bigotimes_n A^\perp \vdash$. Hence, by duality, the implementation is the same as the previous conversion.

Par to Tensor We want to implement a function of type $\parr_n A \multimap \bigotimes_n A$. That is, derive $\parr_n A, \parr_n A^\perp \vdash$ We have two arrays which want to control the order of computation. What we need is an intermediate synchronization mechanism, which presents an array to two programs, pretending to both of them that they are in control:

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

The conversion is then implemented as

$$\begin{array}{l}xs : \parr_n A, ys : \parr_n A^\perp \vdash \\ \mathrm{sync}\{z : A^{\perp n} \mapsto \mathrm{coslice}\ xs\{xs \mapsto_n z \leftrightarrow xs\} \\ \qquad \bar{z} : A^n \mapsto \mathrm{coslice}\ ys\{ys \mapsto_n \bar{z} \leftrightarrow ys\}\}\end{array}$$

The sync construction is in general unsound. Indeed, one program may start by demanding data from element at index 1 and block until that element is produced, while the other program starts by providing the element at index 2 and block until it gets demanded; causing a deadlock. However, if communication via A is unidirectional (say from left to right), then A is a *data type*, and sync is safe. Because the left hand side may not read data from the array (it may only write to it), it is possible to implement sync by allocating an intermediate array of size n, and running the left and right sides of the program in sequence.

Hence, sync can be understood as the allocation primitive of our language, together with the unfused CUT rule. While there are other ways to implement sync (e.g. allocation on demand), wholesale array allocation is a sensible strategy for array-based languages.

Summary The conversion from tensor to sequence and sequence to par correspond to traversals, while the conversion from par to tensor corresponds to allocation (explicit sharing of intermediate results).

$$\begin{tikzcd} & \S_n A \arrow[dr, "\mathrm{traverse}"] & \\ \bigotimes_n A \arrow[ur, "\mathrm{traverse}"] & & \parr_n A \arrow[ll, "\mathrm{sync}"] \end{tikzcd}$$

The other conversions can be implemented by compositions of the above. A noteworthy case is the conversion from tensor to par, via an intermediate sequence. If one eliminates the cut which implements the composition, then the intermediate sequence is gone.

$$\begin{array}{l}x : \bigotimes_n A, y : \bigotimes_n A^\perp \vdash \\ \mathrm{let}\ x = \mathrm{slice}\ x;\ \mathrm{let}\ y = \mathrm{slice}\ y;\ \mathrm{traverse}\{\mapsto_n y \leftrightarrow x\}\end{array}$$

2.10 Difference operator using tensor arrays

The difference operators described in the introduction are instances of one-dimensional stencil computations. In the array transformations, each element y_i in the target is a linear combination of the elements $x_{i-k} \ldots x_{i+k}$ from the source, where k is fixed by the algorithm. Stencil computations are common in numerical methods and image processing (when generalized to 2D).

We demonstrate how stencil operations can be programmed in our language via the difference example.

We first implement the difference operator using tensor arrays. Doing so poses two difficulties.

First, in our example each element of the input of size n is accessed twice. Because we work with linear types, we need two copies of the input array; that is, the input should have type $\bigotimes_2 \bigotimes_n A$. The implementation can alias the two rows to the same value.

Second, the stencil can be applied only on a sufficiently central portion of the array: the borders must be treated specially. We will use use a wrap-around strategy: the template for our stencil computation is

$$\mathrm{diff} = \mathrm{zipWith}(-)\ (\mathrm{rotate1}\ x)\ y$$

where x and y are the two copies of the input array and $\mathrm{rotate1}$ is a function which rotates elements in an array by one position to the left, so that the first one ends up in the last place.

Implementing $\mathrm{rotate1}$ is done by splitting off the first element (with the SPLIT rule), and then appending it to the end of the array (using coslice):

$$\begin{array}{l}i : \bigotimes_{n+1} A, o : \parr_{n+1} A^\perp \vdash \\ \mathrm{let}\ i = \mathrm{slice}\ i;\ \mathrm{let}\ x, y = \mathrm{split}_1\ i; \\ \mathrm{coslice}\ o\{o \mapsto_1 x \leftrightarrow o;\ o \mapsto_n y \leftrightarrow o\}\end{array}$$

After fusion, the implementation of diff looks like this:

$$\begin{array}{l}z : \bigotimes_2 \bigotimes_{n+1} A, o : \parr_{n+1} A^\perp \vdash \\ \mathrm{let}\ z = \mathrm{slice}\ z;\ \mathrm{let}\ x, y = \mathrm{split}_1\ z;\ \mathrm{let}\ x = \mathrm{slice}\ x; \\ \mathrm{let}\ a, b = \mathrm{split}_1\ x;\ \mathrm{let}\ y = \mathrm{slice}\ y;\ \mathrm{let}\ c, d = \mathrm{split}_1\ y; \\ \mathrm{coslice}\ o\{o \mapsto_n (-)[b, d, o];\ o \mapsto_1 (-)[a, c, o]\}\end{array}$$

One issue with $\mathrm{coslice}$ is that handling each element requires branching on its index to chose which program to run. Such a test within a tight loop is bad for performance. Fortunately, this test does not need to be run! Indeed, on the top level, a stencil computation will ultimately output a *par* array. This array will be stored to memory and thus its elements can be written in any order. The stencil computation which writes its result to memory is thus obtained by composing our original stencil operation with the tensor to par conversion given in Sec. 2.9. After fusion, the $\mathrm{coslice}$ on the intermediate par array is gone. We obtain the following program, which shows that each element can be computed independently, and without any test depending on the index.

$$\begin{array}{l}xs : \bigotimes_2 \bigotimes_{n+1} A, ys : \bigotimes_{n+1} A^\perp \vdash \\ \mathrm{let}\ xs = \mathrm{slice}\ xs;\ \mathrm{let}\ x, y = \mathrm{split}_1\ xs;\ \mathrm{let}\ x = \mathrm{slice}\ x; \\ \mathrm{let}\ a, b = \mathrm{split}_1\ x;\ \mathrm{let}\ y = \mathrm{slice}\ y;\ \mathrm{let}\ c, d = \mathrm{split}_1\ y; \\ \mathrm{let}\ ys = \mathrm{slice}\ ys;\ \mathrm{let}\ v, w = \mathrm{split}_1\ ys; \\ \mathrm{mix}\{(-)[a, c, v];\ \mathrm{traverse}\{\mapsto_n (-)[b, d, w]\}\}\end{array}$$

In most programming languages, lifting a condition out of a loop is implemented as a special purpose optimization. In our framework, this lifting comes automatically, as part of cut-elimination. Furthermore, no heuristic is necessary to detect whether it actually improves the program or not: it is guaranteed to always be the case.

Diff2 using tensor arrays Now, let us implement diff2 as the composition of the above function with itself. We can compose either version of the stencil operator: either the latter one (writing to memory, and of type $\bigotimes_2 \bigotimes_n A \multimap \parr_n A$) or the former one (outputting a tensor array and of type $\bigotimes_2 \bigotimes_n A \multimap \bigotimes_n A$).

```
diff  ≡ a : §n ℝ, b : §n ℝ⊥ ⊢
  loop { x' ↦ x' ↔ 0 }  x : §n(ℝ ⊗ (ℝ⊥ & ⊥)) ↦
  traverse { a as a0, b as b0, x as x0 ↦
    let x0, x0' = x0
    sync { a' ↦ a' ↔ a0 } a1 a2 ↦
    mix { let inl a2 = a1; a2 ↔ x0'
        ; b0 ↔ a2 − x0 } }

diff2  ≡ a : §n ℝ, b : §n ℝ⊥ ⊢
  cut { x' ↦  diff a x' }
      { x ↦  diff x b  }
```

Figure 3. First and second order differences in CLL^n.

- Direct composition. In the former case, the polarities match, so the composition does not need an extra array. However, the first instance of the stencil will have to be computed twice: its computation has to be duplicated. Fortunately, the programmer will be informed of this duplication by looking at the types: the type of the composition is $\bigotimes_4 \bigotimes_n A \multimap \bigparr_n A$.
- Via memory. In the latter case, the array polarities (tensor/par) do not match, so we must convert the output of the first pass into a tensor array. This conversion can be done only by allocating an intermediate array which must be allocated using sync suitably.

Even though the behaviour (sharing or duplication) of the composition is predictable, neither situation is quite satisfying. What we would like to obtain is an efficient loop, as shown in the C code in the introduction. The source of inefficiency is the that diff is made to access and produce elements in any arbitrary orders. Before showing an alternative, we make a detour by the MIX and HALT rules.

2.11 Scheduling with MIX and HALT

The MIX and HALT rules are extensions of CLL which preserve cut-elimination.

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

The HALT rule states that a program can terminate even if there is no other program to yield to. The MIX rule is similar to cut, in the sense that it glues two programs together. However, in the MIX case, there is no communication whatsoever occurring between the programs. Hence, they can be executed in any order: there is total freedom in the order of computation, which MIX has to decide.

While there are several operational interpretations of MIX, one for each relevant scheduling strategy, here we simply assume that MIX executes the programs sequentially. Under this interpretation, both MIX and HALT can be implemented in terms of traverse applied to 0 sequences.

2.12 Diff using sequences

Using the sequence type, we can implement the example we described in the introduction, without duplicating computation nor allocating any intermediate array.

To guarantee that each element is consumed only once, we can implement `diff` by traversing the array once (Fig. 3). We use zero-padding to adjust the size of the input array; other approaches such as producing a smaller sequence or wrapping around are also possible.

After fusing, we obtain a single traversal of the input array, with constant allocation of memory (Fig. 4). Furthermore, source and destination arrays are guaranteed to be accessed in a cache-friendly way (Fig. 5).

```
diff2  ≡ a : §n ℝ, c : §n ℝ⊥ ⊢
  loop { x' ↦ x' ↔ 0 } x : §n(ℝ ⊗ (ℝ⊥ & ⊥)) ↦
  loop { y' ↦ y' ↔ 0 } y : §n(ℝ ⊗ (ℝ⊥ & ⊥)) ↦
  traverse { a as a0, c as c0, x as x0, y as y0 ↦
    let x1, x1' = x0
    let y1, y1' = y0
    sync { a' ↦ a' ↔ a0 }          a1 a2 ↦
    sync { b' ↦ b' ↔ a1 − x1 } b1 b2 ↦
    mix  { let inl x2' = x1' ; x2' ↔ x0'
         ; let inl y2' = y1' ; y2' ↔ b2
         ; c ↔ b1 − y1 } }
```

Figure 4. Fused second-order differences

```
void diff2(size_t n, double* a, double* c) {
    double x = 0, y = 0;
    for (size_t i = 0; i < n; i++) {
        double a1 = a[i];
        double b = a1 − x;
        c[i] = b − y;
        x = a1;
        y = b;
    }
}
```

Figure 5. Compiled code for fused, second-order differences. Redundant, static single assignments to local variables have been removed for readability.

3. Cost-free abstractions

When writting a more complex algorithm, the programmer would build combinators that encapsulate basic patterns, and combine them using cut, as illustrated in the previous section. We prove in this section that 1. cuts can always be eliminated and 2. when they are, the performance of the program does not get worse. Together, these properties mean that abstraction is free, in the sense that it does not cost anything to structure a program as a composition of small building blocks.

3.1 Guaranteed fusion

Our cut-elimination algorithm is based on structural cut-elimination in CLL [26], with a one-sided sequent presentation [32].

Theorem 1. *Every given instance of* cut *can be eliminated.*

Proof. We define an admissible rule

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

by structural induction on the intermediate type A and the programs a and b. The definition does not introduce any use of cut. □

The complete set of cases for fuse is given in the supplementary material. The general outline of the definition, as well as an argument for its termination, follows. As a preliminary, we justify why fuse must be generalized to an n-ary version. Consider fuse on an array type:

$$\dfrac{\dfrac{\Gamma, A^{\perp} \vdash \mathrm{a}}{\Gamma^{n}, \parr_{n} A^{\perp} \vdash}\parr \qquad \dfrac{\Xi, A^{n} \vdash \mathrm{b}}{\Xi, \bigotimes_{n} A \vdash}\otimes}{\Gamma^{n}, \Xi \vdash}\textsc{Fuse}$$

The result should be a series of n fuses on A. However, because n is abstract, we cannot expand this series syntactically. Hence we extend the syntax to n-ary fuses to obtain the following reduct, which can be understood as running the program b and n copies of the program a.

$$\dfrac{\Gamma, x : A^{\perp} \vdash \mathrm{a} \qquad \Xi, \bar{x} : A^{n} \vdash \mathrm{b}}{\Gamma^{n}, \Xi \vdash \mathrm{fuse}\{x : A^{\perp} \mapsto \mathrm{a}; \bar{x} : A^{n} \mapsto \mathrm{b}\}}\textsc{Fuse}_{n}$$

We proceed by mutual induction on the programs being fused and the intermediate type. When both programs start by eliminating the type being fused (as in the above example) we say that the fuse is *ready*. The definition of fuse has then two main cases: ready or not.

Ready In the ready case, fusion reduces to fusion on the subcomponents (as shown above). The intermediate types to eliminate are then strictly smaller than the original. Compared to standard cut-elimination, the new cases that we introduce are when types are either tensor/par arrays or sequences (on both sides of the cut),

- The tensor/par case has been discussed above, but there is a complication if one of the programs uses a version of coslice with multiple branches. In such a case, one obtains multiple arrays, which must be merged together. The binary case looks like this:

$$\begin{array}{l}
\mathrm{fuse}\{z : \parr_{n+m} A^{\perp} \mapsto \mathrm{coslice}\ z\{x \mapsto_{n} \mathrm{a};\ y \mapsto_{m} \mathrm{c}\}\\
\quad\ \bar{z} : \bigotimes_{n+m} A \mapsto \mathrm{let}\ \bar{x} = \mathrm{slice}\ \bar{z};\ \mathrm{b}\} \qquad\qquad \Longrightarrow\\[1em]
\mathrm{fuse}\{x : A^{\perp} \mapsto \mathrm{a}\\
\quad\ x : A^{n} \mapsto\\
\qquad\quad \mathrm{fuse}\{y : A^{\perp} \mapsto \mathrm{c};\ y : A^{m} \mapsto \mathrm{let}\ \bar{x} = \mathrm{merge}\ x, y;\ \mathrm{b}\}\}
\end{array}$$

  The merge instruction is also defined by structural induction on its argument. Most rules do not interact with n-ary types, so merge merely commutes with those. The remaining rules are SPLIT, TRAVERSE and COSLICE. In the SPLIT case, merge commutes with just one side of the SPLIT. If the relationship between the sizes is not known statically, then a dynamic test must be introduced.

  Both coslice and traverse will use the merged array in exactly one of the branches. This branch can be split into two, each of them consuming one of the arguments of merge.

- In the sequence case, we have a traverse on both sides of the cut. Then we combine both instances of traverse into a single one which consumes all the sequences that where used by any of the sides. In the following example, we show the case of two-sequence traversals. Observe that all variables, of arity $n + m$, in Γ (used by the program a) are split into two contexts with arities n and m.

$$\begin{array}{l}
\mathrm{fuse}\{x : \S_{n+m} A^{\perp} \mapsto \mathrm{traverse}\{x\ \mathrm{as}\ x_1 \mapsto_{n+m} \mathrm{a}[\Gamma]\}\\
\quad\ y : \S_{n+m} A \mapsto \mathrm{traverse}\{y\ \mathrm{as}\ y_1 \mapsto_{n} \mathrm{b};\ y\ \mathrm{as}\ y_2 \mapsto_{m} \mathrm{c}\}\} \Longrightarrow\\[1em]
\quad\ \mathrm{let}\ \Gamma, \Gamma = \mathrm{split}_{n}\ \Gamma;\\
\quad\ \mathrm{traverse}\{\mapsto_{n} \mathrm{fuse}\{x_1 : A^{\perp} \mapsto \mathrm{a}[\Gamma];\ y_1 : A \mapsto \mathrm{b}\}\\
\qquad\qquad\quad \mapsto_{m} \mathrm{fuse}\{x_1 : A^{\perp} \mapsto \mathrm{a}[\Gamma];\ y_2 : A \mapsto \mathrm{c}\}\}
\end{array}$$

Not ready Ignoring for a moment that fuse can be n-ary, we then have the following situation

$$\dfrac{\Gamma, x : A^{\perp} \vdash \mathrm{a} \qquad w : C, \Delta, y : A \vdash \mathrm{b}}{\Gamma, w : C, \Delta \vdash \mathrm{fuse}\{x : A^{\perp} \mapsto \mathrm{a}; y : A \mapsto \mathrm{b}\}}\textsc{Fuse}$$

where b begins by eliminating $w : C$. It can then be shown that the eliminator of C can be commuted with fuse, for every type C. The commuting step makes the subprogram smaller, which guarantees that fuse eventually becomes ready.

Let us now consider the commuting conversion in the general (n-ary) case:

$$\dfrac{\Gamma, x : A^{n} \vdash \mathrm{a} \qquad w : C, \Delta, y : A^{\perp} \vdash \mathrm{b}}{\Gamma, w : C^{n}, \Delta^{n} \vdash \mathrm{fuse}\{x : A^{n} \mapsto \mathrm{a}; y : A^{\perp} \mapsto \mathrm{b}\}}\textsc{Fuse}_{n}$$

where the program b begins by eliminating $w : C$. This situation is problematic, because commuting the rule which eliminates C is impossible. Indeed, in the conclusion of fuse, C has arity n, and there is no rule which works on a whole array in one shot. The solution however is simple: it suffices to first commute rules on the left-hand (arity-preserving) side of fuse. Proceeding in this order ensures that, when considering the right-hand side, the left-hand-side is ready, and therefore $n = 1$. Then, the right-hand-side is arity-preserving as well, and commuting poses no problem.

Commutation also works if the rule is not an eliminator, such as a cut, sync or LOOP. This means that allocations of intermediate data structures in subprograms do not compromise fusion at an outer level.

3.2 Guaranteed improvement

We proceed to prove that fusion cannot increase the runtime of programs. To this effect, we first give a measure of the size of types (an upper bound of how much memory they need), and then give a measure of programs, which is an upper bound on the computation time used to execute them (ie. no parallelism is assumed).

Definition 1. *We define the size $|A|$ of a type A as follows, where $|A| = |A^{\perp}|$:*

$$\begin{array}{ll}
|A \oplus B| = 1 + max(|A|, |B|) & |0| = 0\\
|A \otimes B| = |A| + |B| & |1| = 0\\
|\bigotimes\limits_{n} A| = n \cdot |A| & |\S\limits_{n} A| = n \cdot |A|
\end{array}$$

Definition 2. *We define the cost* $|\mathrm{a}|$ *of a program* a *according to Fig. 6.*

Because we have a concurrent system, one may wonder if we should account for hidden costs due to synchronization. The answer is no: the type-system guarantees that synchronization needs to happen only when handling the additive fragment, and costs are already accounted for in the corresponding eliminators.

One can then state and prove the property of interest:

Theorem 2. *For any two programs a and b communicating via type A:*

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

Proof. By case analysis. The proof is straightforward because the measure precisely fits the bill. □

Alternate measure: code size Instead the total consumed CPU time, one could be interested in program size. Fusion comes close to be an improvement of the size of programs, but not quite: the size of a program is increased when case commutes with fuse.

$$
\begin{aligned}
|x \leftrightarrow y| &= |A| \\
|\mathrm{cut}\{x : A^n \mapsto \mathrm{a}; y : A^\perp \mapsto \mathrm{b}\}| &= |A| + |\mathrm{a}| + n \cdot |\mathrm{b}| \\
|\mathrm{mix}\{\mathrm{a}; \mathrm{b}\}| &= 1 + |\mathrm{a}| + |\mathrm{b}| \\
|\text{yield to } x| &= 1 \\
|\text{let } \diamond = x; \mathrm{a}| &= 1 + |\mathrm{a}| \\
|\mathrm{halt}| &= 1 \\
|\text{dump } \Gamma \text{ in } x| &= 1 \\
|\text{let } x, y = z; \mathrm{a}| &= 1 + |\mathrm{a}| \\
|\text{connect } z \text{ to}\{x \mapsto \mathrm{a}; y \mapsto \mathrm{b}\}| &= 1 + |\mathrm{a}| + |\mathrm{b}| \\
|\text{case } z \text{ of}\{\text{inl } x \mapsto \mathrm{a}; \text{inr } y \mapsto \mathrm{b}\}| &= 1 + max(|\mathrm{a}|, |\mathrm{b}|) \\
|\text{let inl } x = z; \mathrm{a}| &= 1 + |\mathrm{a}| \\
|\text{let inr } x = z; \mathrm{a}| &= 1 + |\mathrm{a}| \\
|\text{let } x, y = \mathrm{split}_n\ z; \mathrm{a}| &= 1 + |\mathrm{a}| \\
|\text{let } x = \text{slice } z; \mathrm{a}| &= 1 + |\mathrm{a}| \\
|\text{coslice } z\{x \mapsto_n \mathrm{a}; y \mapsto_m \mathrm{b}\}| &= 1 + n \cdot |\mathrm{a}| + m \cdot |\mathrm{b}| \\
|\text{traverse}\{x \text{ as } x', y \text{ as } y' \mapsto_n \mathrm{a}\}| &= 1 + n \cdot |\mathrm{a}| \\
|\mathrm{sync}\{x : D^{\perp n} \mapsto \mathrm{a}; y : D^n \mapsto \mathrm{b}\}| &= 1 + |\mathrm{a}| + |\mathrm{b}| + n \cdot |D| \\
|\mathrm{loop}\{x : D^\perp \mapsto \mathrm{a}; y : \S_m(D \otimes (D^\perp \& 1)) \mapsto \mathrm{b}\}| &= 1 + |\mathrm{a}| + |\mathrm{b}| + |D|
\end{aligned}
$$

Figure 6. Measure of sequential program execution cost. Most rules can be implemented as a fixed set of machine instructions, which we assign an arbitrary cost of 1. The AX rule corresponds to forwarding data from one process to another and *vice versa*, and hence takes an amount of type proportional to the size of the data. Similarly, cut and sync, which allocate data, have a cost proportional to the size of this data.

4. Code Generation

We have implemented a tool which turns CLL^n derivations into structured C code. In this section we give a sketch of its implementation. The transformation process is divided into the following phases: 1. Cut-elimination; 2. Polarization of sequences; 3. Focusing; 4. Generation of C code.

The cut-elimination phase works as described in the previous section. The main idea of the phases 1 to 3 is to interpret a negative type N as a function taking $N^\perp$. For example, $A \parr B$ is implemented as $A^\perp \otimes B^\perp \rightarrow \perp$. However, taking this interpretation literally means that the generated code would contain an excess function calls.

Fortunately, LL can be focused, which, computationally, reduces the number of jumps due to control flow to a minimum. The focused proof can be compiled to similar C code to what one would write by hand.

4.1 Polarization of sequences

The remaining difficulty for code generation is that sequences are self-dual, and as such can not be focused with the standard algorithm [19].

The simplest solution is not to polarize sequences, and resort to function calls at runtime to access the elements. (After all, most of them would be eliminated by the first phase: cut-elimination.)

However, in many cases, one can statically assign a polarity to sequences, such that in every occurence of traverse contains at most one negative sequence. Then, polarisation can handle sequence, and the generated code is maximally efficient.

Theorem 3. *For every derivation in which all sequences appear with arity one and are not nested inside other connectives, there exists an assignment of polarities to each appearance of a sequence type such that:*

- *Whenever one or more sequences are eliminated (either with the* AX *or* traverse*), there will be* at most one *negative sequence in the context.*
- *Sequences introduced with* cut *have opposite signs at each end.*
- *Sequences introduced by* LOOP *and* SYNC *have positive signs.*

Proof. We strengthen the condition in the theorem, by requiring all contexts to have at most one negative sequence. Then proceed by structural induction on derivations. □

5. Examples and benchmarks

We implement and benchmark kernels for computing FFT, 1-dimensional PDE for wave propagation, and the convex hull of a finite set of points on the plane. In all three cases, we achieve a high degree of abstraction and compositionality without compromising execution speed.

5.1 Methodology

The generated code is compiled using GCC 4.9.2. The programs are run and timed on a Fedora 21, 2×8GB 1333MHz RAM, Intel i7 2630QM machine.

For each of the examples, two versions of the C code are generated by alternatively enabling and disabling cut elimination in the prototype compiler.

The object file for each variant is statically linked to the benchmarking program. The running time (userspace) is averaged over between 10 and 100 repetitions; the value is chosen for each benchmark to minimize variance.

We have taken care to ensure that the non-fused code is not unfairly disadvantaged. Optimization level -O3 is used to control for domain-agnostic transformations that state-of-the-art compilers can already perform. Hand-optimized C versions of the algorithms are provided for comparison and measured using the same method.

5.2 FFT kernel

A ubiquitous algorithm in high performance computing is the Fast Fourier Transform. There is a plethora of ways to implement FFT. We choose here to implement the conventional radix-2 decimation in time Cooley-Tukey. At the core of this algorithm is the size-two discrete Fourier transform: shown below:

```
bff ≡ (x0,x1) : ℂ⊗ℂ, (y0,y1) : ℂ⊥⊗ℂ⊥ ⊢
```

$$
\begin{aligned}
y_0 &= x_0 + x_1 w_n^k \\
y_1 &= x_0 - x_1 w_n^k
\end{aligned}
$$

The above function is sometimes called "butterfly", in reference to the shape of the dependency graph between inputs and outputs, both pairs of complex numbers. The w_n^k constants are known as twiddle factors.

As is conventional, our FFT consists of a loop which iterates $logn$ times where n is the number of elements in the input array. We will describe a single iteration: their chaining offers no particular insight to understand par or tensor arrays. We can program each iteration as the chaining of a series of smaller steps, shown below. Thanks to duality, zip and halve can be applied backwards by exchanging the roles of argument and result.

$$\begin{tikzcd}
\bigotimes_{2n}\mathbb{C} \arrow[r, "\text{halve}"] & \bigotimes_n\mathbb{C}\otimes\bigotimes_n\mathbb{C} \arrow[r, "\text{zip}"] & \bigotimes_n(\mathbb{C}\otimes\mathbb{C}) \arrow[r, "\text{map(bff)}"] & \bigotimes_n(\mathbb{C}\parr\mathbb{C}) \arrow[r, "\text{schedule}"] & \bigparr_n(\mathbb{C}\parr\mathbb{C}) & \bigparr_n\mathbb{C}\parr\bigparr_n\mathbb{C} \arrow[l, "\text{zip}"'] & \bigparr_{2n}\mathbb{C} \arrow[l, "\text{halve}"']
\end{tikzcd}$$

The above description decomposes the FFT algorithm into a pipeline of simple functions which are easy to implement. This pattern is typical of functional programming; and is the style which we advocate.

We implement this in our framework. After fusion we obtain an efficient implementation which does not compute nor allocate any intermediate arrays. The generated code (with some variables renamed for readability) consists on one single loop which both reads and performs almost identically to the hand-optimized version.

```
void fftStep (size_t n, double complex *a, double complex *b) {
    double complex z = _Complex_I * 2.0 * M_PI / (double)(n);
    for (size_t i = 0; i < n; i = i + 1) {
        double complex x = a[i];
        double complex y = cexp(((double)i) * z) * arg_aS[n + i];
        b[i] = x + y;
        b[n + i] = x − y;
    }
}
```

5.3 Wave propagation stencil

A numerical solution for wave propagation can be described as a stencil computation [13]. Both time and space are discretized. At a timestep t the displacement of the medium at each point can be computed from the displacements at times $t - 1$ and $t - 2$.

If the stencil computation is simple enough, it is possible to improve performance by fusing several steps together. This is possible in CLL^n if the input and the output of each step have the same type. To achieve this, we will have each of the steps will consume the two vectors corresponding $t - 1$ and $t - 2$, and produce vectors t and $t - 1$ that can be fed to the next step. As in Sec. 2.12, we use a sequence type to implement this stencil operation. Because there are two arrays involved, we use the type $\S_n(\mathbb{R}\otimes\mathbb{R}) \multimap \S_n(\mathbb{R}\otimes\mathbb{R})$.

We decompose the algorithm using a linear version of standard arrow combinators [15]. With them, streaming computations can be expressed as morphisms and products in a suitable category. The resulting data flow is shown in Fig. 7.

We measure the time to compute 600 simulation steps with an input vector of $6{\cdot}10^6$ elements ($6{\cdot}10^4$ for the unfused version). The time is averaged over 3 repetions, and divided by the number of computed wave displacements (both in time and space).

5.4 QuickHull for convex hull computation

Computing the convex hull of a finite set of points is a common operation in computational geometry. The quickhull algorithm [3] has an average complexity of O(n log n) for n points uniformly distributed over the space. We demonstrate how the 2-dimensional case can be implemented in CLL^n.

As is shown by Lippmeier et al. [22], the heart of this algorithm is a `filterMax`-like operation, which produces both the set of points above a line segment, and, among those, the one furthest away from it. For best performance, this operation should be made in with only one pass over the array.

We decompose the algorithm into three kernels which are fused internally. A driver program written in C calls them recursively, yielding a full implementation.

(***) : $\S_n(A \multimap B),\ \S_n(C \multimap D),\ (\S_n(A\otimes C) \multimap B\otimes D)^\perp \vdash$
(&&&) : $\S_n(D \multimap B),\ \S_n(D \multimap C),\ (\S_n(D \multimap B\otimes C))^\perp \vdash$
(>>>) : $\S_n(A \multimap B),\ \S_n(B \multimap C),\ \S_n(A \multimap C) \vdash$
(++) : $\S_n A,\ \S_m A,\ \S_{m+n} A \vdash$
arrowSeq n : $(A \multimap B)^n,\ (\S_n(A \multimap B))^\perp \vdash$
delaySeq k n : $\bigotimes_{k-1} D,\ (\S_n(D \multimap D))^\perp \vdash$
windowSeq k n : $\bigotimes_{k-1} D,\ (\S_n(D \multimap \bigotimes_k D))^\perp \vdash$
coapplySeq : $\S_n(A \multimap B),\ \S_n B^\perp,\ \S_n A \vdash$
applySeq : $\S_n(A \multimap B),\ \S_n A,\ \S_n A^\perp \vdash$

waveSymm ≡prev : $\S_n(\mathbb{R}\otimes\mathbb{R})$, next : $\S_n(\mathbb{R}^\perp \parr \mathbb{R}^\perp) \vdash$
 cut { arrowSeq 1 "waveL" } wL ↦
 cut { arrowSeq (n − 2) "waveC" } wC ↦
 cut { arrowSeq 1 "waveR" } wR ↦
 cut { arrowSeq (n + 1) { (_,p1) ↦ p1 } } snd ↦
 cut { arrowSeq n { a b ↦ a ↔ b } } identity ↦

 cut { wL ++ wC ++ wR } waveF : $\S_n(\bigotimes_3 \mathbb{R}\otimes\mathbb{R}) \multimap \mathbb{R}$ ↦
 cut { identity *** waveF } waveStep ↦
 cut { coapplySeq waveStep next } next' ↦
 cut { delaySeq 1 (n+1) ⊗[0] } delay ↦
 cut { windowSeq 3 (n+1) ⊗[0,0] } window ↦
 cut { delay *** window } stencil ↦
 cut { snd &&& stencil } pipeline ↦
 applySeq pipeline (prev ++ [0]) ({x ↦ dump x} ++ next)

Figure 7. Wave stencil as a data flow. Cuts where the variable of the left side is immediately used as the last argument of a call to another derivation are omitted for conciseness, and literals are used in the place of certain types.

- Select leftmost and rightmost points in the array (1 pass).
- Split points into above and below the line defined by the points in step 1. Select the highest and lowest points among each of the collections defined above (1 pass).
- Given three points (A, B, and C), select the points in the array above the line AB, and among those, the one furthest away from AB; the same is done for line BC (1 pass).

All three kernels are implemented using combinators expressible in pure CLL^n. These linear building blocks can find a maximum (e.g. `best1`), map a sequence point-wise (e.g. `map1`), split it on a predicate (`split`), or duplicate it in constant-space (e.g. `copy1`). We use elements of type $A\oplus 1$ for the sequences so that their lengths can be known before they are traversed (the possibility of bounded sizes is discussed in Sec. 6.1). The overhead of using additive types is reduced by two features of the code generator:

- When chaining several sequence operations, all intermediate values of type $A \oplus 1$ are eliminated thanks to fusion.
- Once synchronized to memory, run-length encoding is used for the tags whenever the elements of a sequence are of an additive type. This representation achieves good cache performance, and can be read back as a contiguous array.

We measure the time required to compute the convex hull of a subset of 10^6 points of a Lissajous curve (10^5 for the unfused version). The time is averaged over 100 repetions, and then divided by the number of elements in the input.

5.5 Summary

The run times for the examples are sumarized in Fig. 9. By applying cut elimination and a straightforward compilation scheme, it is pos-

```
map1 $\equiv \bigotimes_n ((A \multimap B) \& 1),\ \S_n (A \oplus 1),\ (\S_n (B \oplus 1))^\perp \vdash$
comap1 $\equiv \bigotimes_n ((A \multimap B) \& 1),\ (\S_n (B \oplus 1))^\perp,\ \S_n (A \oplus 1) \vdash$
splitS $\equiv \bigotimes_n ((D \multimap 1 \oplus 1) \& 1), \S_n (A \otimes B \oplus 1)$, a, b: $(\S_n (A \oplus 1))^\perp \vdash$
best1 $\equiv \bigotimes_n ((D \multimap D \multimap 1 \oplus 1) \& 1), \S_n (D_1 \otimes D \oplus 1), (D_1 \oplus 1)^\perp \vdash$
copy1 $\equiv \S_n (D \oplus 1),\ (\S_n (D \oplus 1))^\perp,\ (\S_n (D \oplus 1))^\perp \vdash$
withDistTo $\equiv$ a, b : $\mathbb{R} \otimes \mathbb{R},\ (\bigotimes_n ((\mathbb{R} \otimes \mathbb{R} \multimap \mathbb{R} \otimes \mathbb{R} \otimes \mathbb{R}) \& 1))^\perp \vdash$
withDupl : $\bigotimes_n ((A \otimes D \multimap A \otimes D \otimes D) \& 1)$
fst : $\bigotimes_n ((A \otimes D \multimap A) \& 1)$
$(\geq 0)$ : $\bigotimes_n ((\mathbb{R} \multimap 1 \oplus 1) \& 1)$
$(\geq)$ : $\bigotimes_n ((\mathbb{R} \multimap \mathbb{R} \multimap 1 \oplus 1) \& 1)$

splitTopAbove : all : $\S_n (\mathbb{R} \otimes \mathbb{R} \oplus 1)$,
    top' : $(\mathbb{R} \otimes \mathbb{R} \oplus 1)^\perp$, above', rest' : $(\S_n (\mathbb{R} \otimes \mathbb{R} \oplus 1))^\perp \vdash$
  cut { comap1 fst rest' } $\text{rest}_0$' $\mapsto$
  cut { comap1 fst above' } $\text{above}_0$' $\mapsto$
  cut { withDistTo left right } $\text{withDistance}_1$ $\mapsto$
  cut { map1 $\text{withDistance}_1$ all } $\text{all}_0$ $\mapsto$
  cut { map1 withDupl        $\text{all}_0$ } $\text{all}_1$ $\mapsto$
  cut { awd' $\mapsto$ splitS $(\geq 0)$ $\text{all}_1$ awd' rest' } awd $\mapsto$
  cut { $\text{above}_1$' $\mapsto$ copy1 awd $\text{above}_0$' $\text{above}_1$' }
      { $\text{above}_1$ $\mapsto$ best1 $(\geq)$ $\text{above}_2$ top' }
```

Figure 8. QuickHull kernel. The set of points above the line containing segment AB, and the one furthest away from this line are computed.

Algorithm	Unfused	Fully fused	Hand-optimized C
FFT kernel	91 ns	62.8 ns	62.1 ns
Wave propagation	213 ns	9.14 ns	7.20 ns
QuickHull	658 ns	58.0 ns	34.9 ns

Figure 9. Benchmark summary. Code generated before and after applying fusion is compared to a reference hand-optimized C program. Time is average CPU time divided by the number of input elements (QuickHull), or output elements (wave propagation, FFT). This element count is the same for both unfused, fused and hand-optimized code.

sible to run programs rich in abstraction in times resembling those of hand-optimized C code. These results show how the guaranteed fusion enables compositional programming in practical settings.

The compiler and the code used for the benchmarks can be obtained from: `https://lopezjuan.com/limestone/`.

6. Discussion

Duality Much of the literature devoted to the computational interpretation of linear logic is based on the intuitionistic variant (ILL): the duality aspect of LL is given secondary status. In this paper, duality is central, as both tensor and par arrays have their utility. Further, we have shown that memory allocation arises as the conversion from par to tensor, while flow control structures (schedule) arise from the conversion from tensor to par.

Array of Arrays vs. Matrices In usual programming languages, it is common to represent a matrix as an array of arrays. In our language, it is inadvisable to do so, because one cannot convert from an array of rows ($\bigotimes_m \bigotimes_n A$) to a "true" matrix ($\bigotimes_{mn} A$). Indeed, while the input type guarantees that each row is produced independently from the others, it may be that producing a row requires consuming some data B. Now, the output type requires every element of the matrix to be produced independently. If one demands the elements of the matrix by column, instead of by row; producing the first column would require consuming n times B, once for each row that an element is sourced from. But, when the program execution would reach the second column, it will require *the same n* elements of type B to be produced again. There are then two possibilities: either the array of B is saved to memory, or the program producing each of its elements must run multiple times. We must reject the former option, because we guarantee that the intermediate array of B can be eliminated. We must also reject the latter one, because we guarantee that the eliminator of the array of rows does not increase the runtime of the program.

The inequivalence between $\bigotimes_m \bigotimes_n A$ and $\bigotimes_{mn} A$ may come as a surprise, as the tensor operator in CLL is commutative and associative. The reason why the above reasoning does not hold in the binary case is that fusing away the intermediate pair of Bs requires then only a constant amount of data.

Quantifiers For conciseness, this paper presents a version of LL without second order quantification: there is no polymorphism. Adding quantification over types poses no fundamental problem, and is used internally by the code generator. One technical complication concerns the occurrence of types in cost measures, which makes the cost depend on a map from type variables to costs.

Exponentials Most versions of linear logic feature exponentials: the type $!A$ corresponds to n copies of A, where n can be chosen arbitrarily by the program. While adding exponentials to our calculus is straightforward, following previous work on CLL, it means to forego either guaranteed fusion (Thm. 1) or guaranteed improvement (Thm. 2). To remain consistent, logical systems keep cut-elimination, so they give up guaranteed improvement.

In our case, the best choice appears to be the opposite. That is, fusion should remain cost-free, but the programmer may opt-out from fusion by means of annotating a type with exponentials. Exponentials are then interpreted computationally as a thunk in a lazy language; they are evaluated at most once, but may be used many times, by means of storage.

Sequences and parallelism The type of sequences forces a linear traversal of arrays. When converting between tensor- and par arrays it acts as a way of scheduling traversals. But there are more ways to schedule traversals of arrays than to visit each element sequentially. For instance, a parallel schedule is likely to improve the execution time. It is possible to extend our language with more self-dual types, similar to sequences, but which provide other kinds of schedules.

6.1 Related Work

Fusion as Cut-Elimination Framing fusion as cut-elimination in a sequent calculus was first proposed by Marlow [23] in Chapter 3 of his thesis. Marlow identified the important role of commuting conversions, which allow to evaluate any cut, even if the composed programs do not immediately deal with the variable introduced by the cut. Marlow also notes that, by expressing fusion as cut-elimination, he obtains an algorithm simpler than Wadler [30] did in his seminal paper. By basing our work on a linear sequent calculus, we get additional benefits, such as the improvement guarantee.

Bounded Linear Logic and Implicit Complexity The idea of controlling time complexity by using linear types originates with the seminal paper of Girard et al. [12], describing Bounded Linear Logic (BLL). Together with the work of Leivant [20] and Bellantoni and Cook [5], BLL has seeded a sub-field of computer science and logic dedicated to place structural complexity bounds on programs *implicit computational complexity* [8]. Implicit complexity is about enforcing complexity bounds by the using a programming language which enforces those bounds. In particular, the idea of

BLL is to place an upper bound on the number of times each variable can be used, in order to get polynomial time complexity of program execution. (With the added *tour de force* proof that all polynomial functions can be expressed in BLL.)

The present paper describes an even more stringent version of LL, enforcing that each variable is used exactly once. The non-array fragment of our system is called Rudimentary Linear Logic (RLL) by Girard et al. [12], and dismissed with the following words: "a fantastic medicine with respect to problems of complexity, except that the patient is dead! Without contraction the expressive power of logic is so weak that one can hardly program more than programs permuting the components of a pair."

We show that some programming power is recovered by providing suitable array primitives (one could alternatively see the SPLIT rule is a form of contraction). In fact, we believe that, when extended with arrays, RLL is the language of choice for expressing efficient vectorized computations. We have given concrete evidence to this end, by means useful programs written in our language. Yet, the extension to arrays that we propose is compatible with BLL, and we believe that a complete language should incorporate bounded variables.

Another difference with the field of implicit complexity is the goals we aim to achieve. While Girard et al. [12] care about bounding the absolute complexity of programs, what matters for us is that any single cut can be eliminated without worsening the runtime. For us, it does not matter if exponentials are used locally within a function f. As long as they do not show up in the type of f, fusion of f with anything else is guaranteed to be improving the runtime.

Resource Management Linear types can be used to manage usage of all kinds of resources, in addition to CPU time as in BLL.

One kind of resource used by computers is memory, and the above idea can be specialized to it. Wadler [31] has proposed to use linear types to keep track of the number of references to a value. Doing so allows to safely update a value which is referenced only once, even in a functional language without observable side effects.

This idea has been integrated in full fledged programming languages, such as Clean [4], ATS [33], and more recently Mezzo [27].

The above goal is synergistic with ours, and ultimately we believe that programming languages should be striving for both. Our focus on arrays with independent processing of elements is justified by their natural fit in a world where parallelism is becoming ubiquitous. Indeed, programs typed in our language give the desired guarantee of independence between various subprograms: there is no hidden synchronization cost lurking.

Functional Array Programming In this paper we have used the terms par and tensor to refer to the two dual forms of fusible arrays. The functional high performance parallel array community uses the terms push and pull arrays with a similar meaning. In this subsection we will first give the definitions of pull and push arrays as used in that community before comparing them it to our work. First, pull arrays, also known as delayed arrays [16], define operations on arrays as functions from index to element. This kind of array representation can be traced at least as far back as Abrams [1], who defines operations on arrays by how to index into them and also introduces several transformations which achieve fusion. Using Haskell syntax we can define them as follows:

```
type Pull a = Int -> a
```

Fusion will happen if the compiler inlines the function stored in the push array. Besides, it is easy to compile to parallel code because each element can be computed independently.

More recently, Claessen et al. [7] have proposed push arrays as a complement to pull arrays.

```
type Push a = (Int -> a -> P) -> P
```

In the above, the type `P` represent programs which can write to memory, loop, and spawn parallel computations. The final result of a push array can be seen as a program which writes the contents of the array to memory. In the type shown above, that program is parameterized by the function `(Int -> a -> P)` which can be understood as the computation which actually performs the writing to memory, given an index and a value to write to memory. Fusion is also supported by push arrays and they have an efficient parallel implementation.

The functional representation of push and pull arrays is similar par and tensor arrays. Both representations support fusion, and show a duality between the two array types. In both representations, some functions are efficiently implementable using pull/tensor arrays but not push/par arrays and vice versa. In fact, the functional representation corresponds to the continuation-based translation of linear types [18], if one chooses `P` as the type of effects.

Yet, the functional representation suffers from infelicities.

In the case of pull arrays, if an element is accessed several times then it will be recomputed each time — something one would like to avoid in a high performance setting. When applying fusion, the computation for each element becomes bigger each time, exacerbating the problem. Push arrays suffer from a similar problem. Recall that they are defined in terms of a small program which writes to memory. It is important that the push array writes to all memory locations of the array once and only once; otherwise, some elements would be undefined, or, in a parallel setting, subject to race conditions.

The problems of pull and push arrays share one trait: they are caused by a lack of linearity. This is the reason why, in this paper, we have chosen to use linear logic as a starting point for the type system, via the Curry-Howard correspondence.

We believe that the choice of a classical, linear framework reveals the essential duality of push and pull arrays, whereas the functional representation is cluttered with artifacts due to the encoding. For example, we have seen that, in our framework, the cost introduced by pull array concatenation (as in Sec. 2.10) goes away when fusion is performed. Using the functional representation, the test is baked into an opaque function, and thus will remain even after the compiler has inlined functions. The end result is that pull array concatenation is inefficient using the functional representation, while it behaves well in our framework.

Repa The library Repa [16] also makes use of pull arrays, as functions from index to element. When faced with the shortcomings of pull arrays they developed a more elaborate version, tailored to efficiently compute stencil computations [21]. Their new representation, although still similar to pull arrays, solves some of their problems, such as the inefficient concatenation. Yet, their representation does not exhibit any duality, and it is not clear to us whether it solves all the problems that push arrays solve, in particular sharing computations between array elements when writing to an array, as we do in the FFT example.

MoA Mathematics of Arrays [24] is a framework for reasoning about array computations. It is tailored towards calculating array programs. The framework provides a way for producing efficient programs from calculations, by means of Ψ-reductions, which effectively performs fusion. Semantically Mathematics of Arrays considers arrays as functions from index to value, just as pull arrays, which makes the two frameworks very similar.

Data-flow Fusion Lippmeier et al. [22] implements branching data-flow fusion for Haskell by internally tagging each sequence with a rate, and applying fusion only when the rates match. This allows the programmer to use the usual idioms for filtering, zipping, mapping, and unzipping lists while removing redundant intermediate arrays and loop counters.

However, the information on whether fusion can be performed is hidden from the programmer. In our language, we instead rely on the core linearity properties of the system, and encode the composability of the data-flow in the same framework.

Generalized optimizations The fusion technique presented in this paper, and the related work mentioned so far, hails from the work on shortcut fusion [10]. The central idea is to pick a particular representation for the type which should be eliminated, in our case arrays, so that it becomes amenable to fusion. However, there are many techniques which can achieve the effect of fusion, including supercompilation [29], deforestation [30] and fixed point promotion [25]. These methods have the advantage that they can remove intermediate structures even when the programmer has not been careful to use a fusion-enabled type. Their downside is that they can be unreliable; it is hard to predict when fusion fail. Furthermore, they typically rely on rewriting whole functions, whereas shortcut fusion and its decendants relies on local rewrite rules which can be easily incorporated into an optimizing compiler.

6.2 Conclusion

We have shown that classical linear types are a suitable framework for describing composable programs with predictable fusion properties. The resulting calculus can form the basis for a functional language for predictable high-performance computations. Indeed, our examples show how it is possible to write code in a functional style (immutable data, function composition) while having the certainty that these abstractions will not translate into reduced performance.

Programming in a functional language may appear restrictive: the absence of side effects is a straitjacket which constrains creativity. Yet, to the initiated, using a functional language is a liberating experience: one may combine functions at will, without any risk of side effects getting in the way.

We feel the same about programming in a linear language: at first, the discipline of linearity appears daunting, severely constraining the kind of programs one can write; but in time, one enjoys the truly cost-free abstractions granted by guaranteed fusion.

Acknowledgments

Víctor López Juan is funded by a "la Caixa" scholarship. Other authors are partially funded by SSF. We thank Cezar Ionescu, Nicolas Gainot, Peter Jonsson, Nicolas Pouillard and anonymous reviewers for useful feedback on previous versions of this paper.

References

[1] P. Abrams. *An APL Machine*. PhD thesis, Stanford University, 1970.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.

[3] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, Dec. 1996. ISSN 0098-3500. .

[4] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Foundations of Soft. Tech. and Theor. Comp. Sci.*, pages 41–51. Springer, 1993.

[5] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational complexity*, 2(2):97–110, 1992.

[6] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[7] K. Claessen, M. Sheeran, and B. J. 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*, pages 21–30. ACM, 2012.

[8] U. Dal Lago and M. Hofmann. Bounded linear logic, revisited. In *Typed Lambda Calculi and Applications*, pages 80–94. Springer, 2009.

[9] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *Journal of functional programming*, 13(03):455–481, 2003.

[10] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. of FPCA*, pages 223–232. ACM, 1993.

[11] J.-Y. Girard. Linear logic. *Theor. Comp. Sci.*, 50(1):1–101, 1987.

[12] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theor. Comp. Sci.*, 97(1):1 – 66, 1992.

[13] A. Hoekstra, P. Sloot, et al. Lecture notes modelling and simulation, master programme computational science, a case study, the guitar string. 2014.

[14] J. Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[15] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1):67–111, 2000.

[16] B. L. G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, and S. L. P. Jones. Regular, shape-polymorphic, parallel arrays in haskell. In *ICFP*, pages 261–272, 2010.

[17] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. L. P. Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *International Conference on Functional Programming*, pages 261–272, 2010. .

[18] O. Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, 2002.

[19] O. Laurent. A proof of the focalization property of linear logic. Note available on the author's web page., 2004.

[20] D. Leivant. A foundational delineation of computational feasibility. In *Logic in Comp. Sci., 1991. LICS'91., Proc. of Sixth Annual IEEE Symposium on*, pages 2–11. IEEE, 1991.

[21] B. Lippmeier and G. Keller. Efficient parallel stencil convolution in haskell. In *Haskell*, pages 59–70, 2011.

[22] B. Lippmeier, M. M. Chakravarty, G. Keller, and A. Robinson. Data flow fusion with series expressions in haskell. In *ACM SIGPLAN Notices*, volume 48, pages 93–104. ACM, 2013.

[23] S. D. Marlow. *Deforestation for Higher-Order Funct. Programs*. PhD thesis, University of Glasgow, 1995.

[24] L. M. R. Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, December 1988.

[25] A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In *ACM SIGPLAN Notices*, volume 42, pages 143–154. ACM, 2007.

[26] F. Pfenning. Structural cut elimination. In *Logic in Comp. Sci., Symposium on*, page 156. IEEE, 1995.

[27] F. Pottier and J. Protzenko. Prog. with permissions in Mezzo. In *Proc. of the 2013 ACM SIGPLAN International Conf. on Funct. Prog. (ICFP'13)*, pages 173–184, 2013.

[28] J. Svensson. *Obsidian: GPU Kernel Programming in Haskell*. PhD thesis, Chalmers University of Technology, 2011.

[29] V. F. Turchin. Program transformation by supercompilation. In *Programs as Data Objects*, pages 257–281. Springer, 1986.

[30] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comp. Sci.*, 73(2):231–248, 1990.

[31] P. Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *Prog. Concepts and Methods*. North-Holland, 1990.

[32] P. Wadler. Propositions as sessions. In *Proc. of ICFP 2012*, ICFP '12, pages 273–286. ACM, 2012.

[33] D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *Practical Aspects of Declarative Languages*, pages 83–97. Springer, 2005.