Composable Efficient Array Computations Using Linear Types

Jean-Philippe Bernardy
bernardy@chalmers.se

Víctor López Juan Josef Svenningsson
Chalmers University of Technology
lopezv@chalmers.se josefs@chalmers.se

Abstract

Functional languages excel at describing complex programs as the composition of small building blocks. Yet, the performance properties of such compositions depend on compiler heuristics, whose behavior is difficult to predict.

In this paper, we introduce an array-programming calculus which can guarantee elimination of intermediate arrays when composing two programs, eliminating the cost of composition. The calculus has linear types and is an extension of Girard's linear logic with vector types and a synchronization primitive.

We illustrate the effectiveness of our language by implementing a number of classical algorithms in a compositional style, and then compiling these examples to efficient code.

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 (a useful operation for approximating derivatives or interpolating polynomials). A straightforward implementation of diff in C is as follows:

```
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 calls to `diff` communicate via an intermediate array, which needs to be allocated in full. This allocation is undesirable; a C programmer aiming for performance would likely avoid it, and instead rewrite `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]; }
```


Hughes [14] has notoriously argued that a key advantage of functional programming is that programs can be written by composition of simple building blocks, while retaining good run-time behavior. Trusting this *composability principle*, we could write diff2 as follows:

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

diff2 = diff . diff
```

Unfortunately, most compilation strategies will allocate intermediate data in the above implementation of `diff2`. Even in a lazy language, the intermediate list will be allocated piece-wise, 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. A possible approach, used in several functional DSLs [2, 6, 9, 18, 28], is to represent an immutable array of type A by a function $\mathrm{Int} \to A$; 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 gain comes at the price of duplicated computations. 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 sufficiently smart compiler may spot the duplication in the later code-generation phases, this leaves the programmer in the dark: the only way to predict the performance behavior of the generated code is to have an intimate knowledge of the optimization passes implemented in the compiler, which may even vary between versions. Instead, we want a logical calculus in which gives strong guarantees to the programmer, so that they can reliably predict the behavior of generated code from the types of the arrays involved. Our contributions are as follows:

- We extend classical linear logic [11] with vector types ($\bigotimes_n A$, $\parr_n A$, $\S_n A$) 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 every composition of well-typed functions can be fused (in the sense of Wadler [31]), thus removing the allocation of any intermediate storage. Fusion techniques that currently rely on rewrite rules can instead be modeled as type-directed transformations. By enforcing linearity we guarantee that these transformations preserve the computational cost, thus avoiding the need for heuristics (Sec. 4).

- Yet, programmers are given the opportunity to explicitly use allocation and scheduling primitives. Indeed, memory allocation and sequential scheduling correspond to the conversion between different array types (Sec. 2.8). In practice, this means that in any given function composition, the types indicate exactly if an allocation (or a sequential traversal) can be avoided.
- We provide a cost model using an interpretation of CLL^n programs into the λ-calculus. This same interpretation is used to provide a compiler from CLL^n programs to structured, imperative C code.
- 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. Syntax and Examples

In this section we describe CLL^n and motivate its design using a number of examples. The syntax and typing rules of this calculus are summarized in Fig. 1. We leave out issues such as syntactic sugar, type inference, etc.: CLL^n is a core calculus for a full-fledged language.

2.1 CLL as a programming language

In the Linear Logic of Girard [11] (which we abbreviate CLL, for "classical"), proof derivations can be interpreted as programs (as per Curry-Howard). The propositions at the root of the derivation are the input and output arguments of the program; computation is performed by applying proof 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 input values and results dually. Technically every type A has a dual $A^\perp$, and returning a result of type A is equivalent to consuming a value of type $A^\perp$ (Fig. 2). Furthermore, dualization is involutive: $(A^\perp)^\perp = A$.

Duality makes for an economic design, particularly suitable to a core language. Indeed, 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. Consequently our typing judgment has the form $\Gamma \vdash a$, where Γ is a context and a is a program discharging all the inputs in Γ. Such a presentation means that it is enough to have eliminators, as introduction is expressed as the elimination of the dual. Effectively, duality cuts in half the number of constructions one has to deal with. We make further use of duality via the self-dual sequence-type operator (Sec. 2.3).

We adopt the convention that contexts are ranged over by capital Greek letters. Further, when contexts are mentioned on either side of a comma, they must be disjoint; and contexts are order-agnostic.

2.2 Cut-Elimination as a Cost-Free Abstraction

The cut rule represents the composition of programs a and b. As a first approximation, one can computationally interpret the above cut as first running b, storing its result into y, a memory area large enough for a value of type A, and then running a, which accesses the data via x. 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).

In CLL (and indeed CLL^n), every individual cut can be eliminated regardless of the definition of the composed programs (they may themselves contain cuts). Cut elimination is traditionally used to give a computational interpretation to the logic. In this paper, we realize this property by defining an additional rule, logically equivalent to CUT, but which is recursively defined instead of postulated, so it is guaranteed to disappear from the program (Thm. 5). 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 run-time behavior of the program (Thm. 6). This property means that the abstraction mechanism provided by fuse (composition and naming of intermediate results) is implemented in a manner that incurs no overhead. Consider as an example a fuse on the multiplicatives: *tensor* ($\otimes$), and *par* ($\parr$) :

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

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

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

Finally, remark that *linear implication* ($\multimap$) is an arrow connective which models functions in linear logic. In CLL, we can define it in terms of par: $A \multimap B \equiv A^\perp \parr B$.

2.3 Arrays

While Girard's CLL offers strong computational 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.

Sizes and predicates First we extend the type system with size variables. Size variables live in an implicit, intuitionistic side of the context, unaffected by linearity constraints. Thus, they can be copied and used any number of times.

Type-level size terms are constructed from size variables according to the following grammar (where α is a non-negative size variable, and c is an integer constant).

$$n, m ::= \alpha \quad | \quad n + m \quad | \quad n - m \quad | \quad c \cdot n \quad | \quad c$$

Predicates about sizes are expressed as single inequalities between two size expressions ($\geq$, $\leq$, $>$, $<$, $=$, $\neq$). The resulting expressions are entirely within the theory of Quantifier-Free Linear Integer Arithmetic, which Barrett et al. [4] shows as efficiently decidable. The decidability allows us to check, for instance, that all size expressions must be provably ≥ 0 in the context in which they are used. In Sec. 6 we discuss the merits and limitations of this approach.

Both sizes and predicates have their corresponding existential and universal quantifiers. A function f of type $f : \forall \alpha : \mathbb{N}.A[\alpha] \multimap B[\alpha]$ can be applied to any size n, yielding $f@n : A[n] \multimap B[n]$. Conversely, to implement f one eliminates of a variable of the dual type, $f : \exists \alpha : \mathbb{N}.A[\alpha] \otimes B[\alpha]^\perp$.

Preconditions work in a similar way. We can restrict the above function to work on sizes $\alpha \geq 1$ by introducing a precondition p:

$$f_1 : \forall \alpha : \mathbb{N}.\forall p : \alpha \geq 1.A[\alpha] \multimap B[\alpha]$$

The implementation of f_1 will eliminate an existential, thus obtaining a witness of the fact that $\alpha \geq 1$. This witness will be carried in the context, and can be used as an additional assumption about α when eliminating $A[\alpha] \otimes B[\alpha]^\perp$.

$$f_1 : \exists \alpha : \mathbb{N}.\exists p : \alpha \geq 1.A[\alpha] \otimes B[\alpha]^\perp$$

By contrast, the user of f_1 can only use the function in those contexts where $\alpha \geq 1$ holds. Whether this is the case is decided by the type-checker based on the laws of arithmetic, and the predicates already in the context.

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

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

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

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

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

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

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

$$\frac{\Gamma, x : A[n] \vdash \mathrm{a}}{\Gamma, z : \forall \alpha : \mathbb{N}.A[\alpha] \vdash \text{let } x = z@n; \mathrm{a}}\ \forall \qquad \frac{\Gamma, x : A[\beta] \vdash \mathrm{a}}{\Gamma, z : \exists \alpha : \mathbb{N}.A[\alpha] \vdash \text{let } x\langle\beta\rangle = z; \mathrm{a}}\ \exists \qquad \frac{\Gamma, x : A \vdash \mathrm{a} \qquad n \le m}{\Gamma, z : \forall p : n \le m.A \vdash \text{let } x = z@\top; \mathrm{a}}\ \forall$$

$$\frac{\Gamma, x : A \vdash \mathrm{a}}{\Gamma, z : \exists p : n \le m.A \vdash \text{let } x\langle p\rangle = z; \mathrm{a}}\ \exists \qquad \frac{\Gamma \vdash \mathrm{a} \qquad \Gamma \vdash \mathrm{b}}{\Gamma \vdash \mathrm{compare}\ n, m\{\geq \ \mapsto \mathrm{a}; \leq \ \mapsto \mathrm{b}\}}\ \textsc{compare}$$

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
$\bigotimes_n A$	$\parr_n A^\perp$	arrays
$\S_n A$	$\S_n A^\perp$	sequential arrays
$\mathbb{A}$	$\mathbb{A}^\perp$	primitive atoms
$\exists \alpha : \mathbb{N}.A[\alpha]$	$\forall \alpha : \mathbb{N}.A[\alpha]^\perp$	size quantifiers
$\exists p : n \le m.A$	$\forall p : n \le m.A^\perp$	predicate quantifiers

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

As can be seen (or rather, not seen) in Fig. 1, size variables and predicates about them are carried implicitly in the context. One can introduce new predicates by branching dynamically with compare. If one branches on $n \le m$, then the left branch will can make use of $n \le m$ as an assumption, whereas the right branch can make use of the negation, $m < szN$.

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 of size n. The size n can contain symbolic variables in general.

Accessing individual copies is done using special-purpose rules:

- We omit the superscript when it is equal to 1. A variable can be eliminated only if its size is equal to 1 in the current context. This is decidable (see Sec. 2.3).
- The shorthand Γ^n multiplies all superscripts in Γ by n. Note that, because one can only multiply by a constant in sizes, if n is not a constant, then all the variables in Γ must have constant size.

We stress that superscripts are part of the context, not part of the type. In particular, they 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 consumes *each* of the n values exactly once in the order it pleases.

Par Arrays The dual of the tensor array is the par array, written ($\parr_n A$). By duality, the producer of a par array must be able to produce its elements in any given order. Hence, the eliminator of $\parr_n A$ (coslice) must ensure that each element is handled independently. This is realized by the $\parr$ rule, where the body a has access to a single element of the array, and an nth fraction of the context. However, all elements need not be processed in the same way. In general the programmer can specify as many different programs as the length of the array. In Fig. 1, a is used for elements up to index n, and b for the m elements thereafter.

Sequences We have seen that the consumer of a tensor array dictates the processing order, while the consumer of a par array must accept any requested order. There is a middle way: to use a canonical order of processing which is fixed by the programming language. We call a fixed-order array a sequence, and write it $\S_n A$. The fixed order is reflected in the logical self-duality of the type operator: $(\S_n A)^\perp = \S_n A^\perp$. As in par arrays, elements from each

sequence are processed one at a time; but, as with tensor arrays, a computation can involve elements from more than one sequence.

2.4 Basic functions: zip and saxpy

One example of a basic building block is $\mathrm{zipWith(f)}$, which combines two tensor arrays index-wise, via a combination function f.

$$\begin{array}{l}\mathrm{zipWith(f)} \equiv xs : \bigotimes_n A, ys : \bigotimes_n B, zs : \parr_n C^\perp \vdash \\ \text{let } xs_1 = \text{slice } xs;\ \text{let } ys_1 = \text{slice } ys; \\ \text{coslice } zs\{zs_1 \mapsto_n \mathrm{f}[xs_1, ys_1, zs_1]\}\end{array}$$

Using zipWith we can implement 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:

$$\begin{array}{l}\mathrm{saxpy} \equiv xs : \bigotimes_n A, ys : \bigotimes_n A, a : \bigotimes_n A, rs : \parr_n A^\perp \vdash \\ \mathrm{fuse}\{\tau : \parr_n \mathbb{R}^\perp \mapsto \tau \leftrightarrow \mathrm{zipWith}(*)[a, xs] \\ \qquad \tau_1 : \bigotimes_n \mathbb{R} \mapsto rs \leftrightarrow \mathrm{zipWith}(+)[\tau_1, ys]\}\end{array}$$

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

$$\begin{array}{l}\text{let } a_1 = \text{slice } a;\ \text{let } xs_1 = \text{slice } xs;\ \text{let } ys_1 = \text{slice } ys; \\ \text{coslice } rs\{rs_1 \mapsto_n \mathrm{cut}\{\bar{z} : A \mapsto (+)[\bar{z}, ys_1, rs_1] \\ \qquad\qquad\qquad\qquad z : A^\perp \mapsto (*)[a_1, xs_1, z]\}\}\end{array}$$

The above code still has a cut on the type A, because we have assumed that A is abstract. See end of Sec. 4.1 for a discussion.

2.5 Scheduling with MIX and HALT

The MIX and HALT rules are extensions of CLL which preserve cut-elimination (Fig. 1). 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.

2.6 Recovering weakening and contraction

One may worry that linearity requires us to define functions and values once for each time that they are used. For example, in the examples from Sec. 2.4, if one wants to apply a function to an array of length n, seems to need to define the same function n times. The rule $\textsc{Cut}_n$ allows using a value of any type (e.g. a function) a bounded number of times (n). This does not fully solve the problem, because, in order to use $\textsc{Cut}_n$, the values in the context used in the left side of the cut must themselves have arity n.

Furthermore, observe that any number values of type $1 \oplus 1$ can be discarded and duplicated just with the eliminators $\&_1$, $\&_2$ and $\oplus$. For our system, any value constructed only from atomic types ($\mathbb{A}$) and positive connectives ($\oplus$, $\otimes$, $\bigotimes$, 1, $\exists$) is considered a data type D. By applying $\textsc{sync}^k$ such a value can be computed only once and used up to k times.

Conversely, whether a value is used or not may depend on the inputs to the program. For data types, one may make 0 copies with $\textsc{sync}^0$, which effectively discards the value. In general, weakening is simulated with types of the form $A \,\&\, 1$, which can either be used as a value of A, or ignored as a value of type 1. Note that allowing weakening for general types would allow constructing values of any type by discarding their dual, thus rendering the calculus unsound.

In Sec. 6 we discuss how this approach differs from the more traditional notion of exponentials.

$$\begin{array}{l}\mathrm{fuse}\{\tau : \parr_n \mathbb{R}^\perp \mapsto \tau \leftrightarrow \mathrm{zipWith}(*)[a, b] \\ \quad \tau_1 : \bigotimes_n \mathbb{R} \mapsto \\ \quad\quad \text{let } \tau_2 = \text{slice } \tau_1; \\ \quad\quad \mathrm{loop}\{mw : \mathbb{R}^\perp \mapsto mw \leftrightarrow 0.0 \\ \quad\quad\quad mu : \S_{n+1}(\mathbb{R} \otimes (\mathbb{R}^\perp \,\&\, 1)) \mapsto \\ \quad\quad\quad\quad \mathrm{traverse}\{mu \text{ as } mu_1 \mapsto_n \\ \quad\quad\quad\quad\quad\quad \text{let } \tau_3, \tau_4 = mu_1;\ \text{let inl } \tau_5 = \tau_4; \\ \quad\quad\quad\quad\quad\quad \tau_5 \leftrightarrow +[\tau_3, \tau_2] \\ \quad\quad\quad\quad\quad mu \text{ as } mu_2 \mapsto_1 \\ \quad\quad\quad\quad\quad\quad \text{let } \tau_6, \tau_7 = mu_2;\ \text{let inr } \tau_8 = \tau_7; \\ \quad\quad\quad\quad\quad\quad \text{let } \diamond = \tau_8;\ r \leftrightarrow \tau_6\}\}\}\end{array}$$

$$\begin{array}{l}\text{let } a_1 = \text{slice } a;\ \text{let } b_1 = \text{slice } b; \\ \mathrm{loop}\{mw : \mathbb{R}^\perp \mapsto mw \leftrightarrow 0.0 \\ \quad mu : \S_{n+1}(\mathbb{R} \otimes (\mathbb{R}^\perp \,\&\, 1)) \mapsto \\ \quad\quad \mathrm{traverse}\{mu \text{ as } mu_1 \mapsto_n \text{let } \tau, \tau_1 = mu_1;\ \text{let inl } \tau_2 = \tau_1; \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathrm{cut}\{w : \mathbb{R} \mapsto \tau_2 \leftrightarrow +[\tau, w] \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad v : \mathbb{R}^\perp \mapsto v \leftrightarrow *[a_1, b_1]\} \\ \quad\quad\quad\quad\quad mu \text{ as } mu_2 \mapsto_1 \text{let } \tau_3, \tau_4 = mu_2;\ \text{let inr } \tau_5 = \tau_4; \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad \text{let } \diamond = \tau_5;\ r \leftrightarrow \tau_3\}\}\end{array}$$

Figure 3: Dot product, before and after cut elimination.

2.7 Loops

As explained in Sec. 2.3, the producer and the consumer of a sequence 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 possibility is realized by the LOOP rule. When applying the rule, the programmer chooses a type D and a size m. Then, they can provide an initial value of type D, and, sequentially, m functions of type $D \multimap D \oplus 1$. (In many but not necessarily all cases, all these functions have the same implementation.) At each step, the current function receives a value and may chose to provide a new one (by returning an D), 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 D in a left-to-right sequence.

As an example of LOOP, we implement a dot-product function, by composing $\mathrm{zipWith}(*)$ with a LOOP computing the sum of the result. After cut-elimination, a single loop remains (Fig. 3).

2.8 Conversions between array types

In this section we explain how array conversions correspond to traversal or allocation.

Tensor to Sequence We want to implement a function of type $\bigotimes_n A \multimap \parr_n A$, that is, derive $\bigotimes_n A, \S_n A^\perp \vdash$. The derivation is a direct application of slice and $\mathrm{traverse}$:

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

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

Par to Tensor We want to implement a function of type $\parr_n D \multimap \bigotimes_n D$. That is, derive $\parr_n D, \parr_n D^\perp \vdash$ We have two arrays which want to control the order of computation. What we need is an intermediate synchronization mechanism ($\textsc{sync}_n$)), which presents

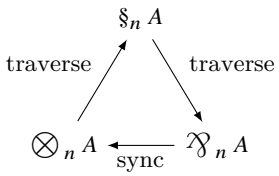
a tensor-like array to two programs, pretending to both of them that they are in control (Fig. 1). The conversion is then implemented as

$$\begin{aligned}&\text{freeze} \equiv xs : \parr_n D, ys : \parr_n D^\perp \vdash \\ &\text{sync}\{z : D^{\perp n} \mapsto \text{coslice } xs\{xs_1 \mapsto_n z \leftrightarrow xs_1\} \\ &\qquad \bar{z} : D^n \mapsto \text{coslice } ys\{ys_1 \mapsto_n \bar{z} \leftrightarrow ys_1\}\}\end{aligned}$$

Note that this derivation is only valid if D is a data type (Sec. 2.6). In particular, the rule SYNC_2 corresponds to a conversion $D \parr D \multimap D \otimes D$.

Observe that a general rule $A \parr B \multimap A \otimes B$ would be unsound, since it allows the construction of a value of the empty type $0 \otimes \top$. In a model where SYNC_n is interpreted as allocating a thunk to be filled later, the resulting program would block at runtime because of an implicit cyclic dependency.

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



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: and the conversion is a traversal involving no sequences, but only the vectors in context.

$$\begin{aligned}&\text{tensorToPar} \equiv x : \bigotimes_n A, y : \bigotimes_n A^\perp \vdash \\ &\text{let } x_1 = \text{slice } x;\ \text{let } y_1 = \text{slice } y;\ \text{traverse}\{\mapsto_n y_1 \leftrightarrow x_1\}\end{aligned}$$

2.9 Difference operator using linear arrays

In this section we demonstrate the interaction between different array types with the diff operator example from Sec. 1. We will also see some shortcomings of programming with par and tensor arrays, which justify the introduction of sequences.

Implementing the difference operator using tensor arrays 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 $\bigotimes_2 \bigotimes_n A$. Second, the stencil can be applied only on a sufficiently central portion of the array: the borders must be treated specially. Here we will use use a wrap-around strategy: the template for our stencil computation is

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

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

Implementing 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):

$$\begin{aligned}&\text{rotate} \equiv i : \bigotimes_{n+1} A, o : \parr_{n+1} A^\perp \vdash \\ &\text{let } i_1 = \text{slice } i;\ \text{let } x, y = \text{split}_1\ i_1; \\ &\text{coslice } o\{o_1 \mapsto_1 x \leftrightarrow o_1;\ o_2 \mapsto_n y \leftrightarrow o_2\}\end{aligned}$$

After fusion, the implementation of diff looks like this:

```
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′ : ℝ ↦ a′ ↔ a0 } { d : ℝ² ↦
      split1 d { a1 a2 ↦ mix { let inl a2 = a1; a2 ↔ x0′
                                ; −[a2,x0,b0] } } }

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

Figure 4: First and second order differences in CLL^n.

$$\begin{aligned}&\text{diff} \equiv z : \bigotimes_2 \bigotimes_{n+1} A, o : \parr_{n+1} A^\perp \vdash \\ &\text{let } z_1 = \text{slice } z;\ \text{let } x, y = \text{split}_1\ z_1;\ \text{let } x_1 = \text{slice } x; \\ &\text{let } a, b = \text{split}_1\ x_1;\ \text{let } y_1 = \text{slice } y;\ \text{let } c, d = \text{split}_1\ y_1; \\ &\text{coslice } o\{o_1 \mapsto_n (-)[b, d, o_1];\ o_2 \mapsto_1 (-)[a, c, o_2]\}\end{aligned}$$

One issue with coslice is that handling each element requires branching on its index to chose which program to run. Such a test within a tight loop performs badly. In most programming languages, lifting a conditional out of a loop is a special purpose optimization. In our framework, this lifting comes automatically, as part of cut-elimination.

Diff2 using tensor arrays Now, let us implement diff2 as the composition of the above function with itself. We have two options for the composition. Either i) we allocate an intermediate array to store the intermediate result and obtain two copies of it (SYNC^2) to feed to the next application of diff, or ii) we require 4 copies of the array as input, and compute the intermediate result twice.

Even though the behavior (sharing or duplication, respectively) 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 that diff is able to access and produce elements in any arbitrary order.

Diff2 using sequences By restricting the access order using the sequence type, we can implement diff2 without duplicating computation or allocating any intermediate arrays. To guarantee that each element is consumed only once, we can implement diff by traversing the array once (Fig. 4). We use zero-padding to adjust the size of the input array; producing a smaller sequence or wrapping around are other supported approaches.

After fusing, we obtain a single traversal of the input array, with constant allocation of memory (Fig. 5). This matches closely the diff2_fused example in Sec. 1. Furthermore, source and destination arrays are accessed in a sequential, cache-friendly way (Fig. 6).

3. Semantics of CLL^n

We give a computational interpretation of CLL^n into λ-terms. This demonstrates a correspondence between CLL^n and sequential, single-threaded programs.

3.1 Double-negation translation

While the CLL can be interpreted as concurrent processes [32], a literal interpretation of this semantics would mean that the generated code would not be very efficient: running concurrent, communicating processes is expensive. Fortunately, CLL can be embedded in the lambda calculus by means of a double-negation translation. This embedding has the effect of assigning an order of evaluation, which in turns means that single threaded, efficient code

```
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 a_0, c as c_0, x as x_0, y as y_0 ↦_n
    let x_1, x_1′ = x_0
    let y_1, y_1′ = y_0
    sync { a′ : ℝ ↦ a′ ↔ a_0 } { d_0 : ℝ^2 ↦
      split_1 d_0 { a_1 a_2 ↦ sync { b′ : ℝ ↦ −[a_1,x_1,b′] }
        { d_1 : ℝ^2 ↦ split_1 d_1 { b_1 b_2 ↦ mix {
                  let inl x_2′ = x_1′ ; x_2′ ↔ x_0′
                  ; let inl y_2′ = y_1′ ; y_2′ ↔ b_2
                  ; −[b_1,y_1,c] } }
```

Figure 5: 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 6: Compiled code for fused, second-order differences. Redundant, static single assignments to local variables have been removed for readability.

can be generated. This translation is standard [15], thus we recall only the translation of types.

We assume the existence of a type of effects which we write as $\bot$. he interpretation of a CLL^n_p term will always have type $\bot$, representing the effects or results of the computation. The effect type can be any monoid, whose operations interpret MIX (composition, $\gg$) and HALT (unit, nop).

Definition 1. *We define the translation of a type A to a polymorphic lambda calculus type $A^\bullet$ as follows:*

$$\begin{aligned}
\mathbb{A}^\bullet &= (\mathbb{A} \to \bot) \to \bot & \mathbb{A}^{\bot\bullet} &= \mathbb{A} \to \bot \\
0^\bullet &= (\varnothing \to \bot) \to \bot & \top^\bullet &= \varnothing \to \bot \\
1^\bullet &= \bot \to \bot & \bot^\bullet &= \bot \\
(A \oplus B)^\bullet &= (A^\bullet + B^\bullet \to \bot) \to \bot & (A \,\&\, B)^\bullet &= A^{\bot\bullet} + B^{\bot\bullet} \to \bot \\
(A \otimes B)^\bullet &= (A^\bullet \times B^\bullet \to \bot) \to \bot & (A \parr B)^\bullet &= A^{\bot\bullet} \times B^{\bot\bullet} \to \bot \\
(\bigotimes_n A)^\bullet &= ((\mathbb{N} \to A^\bullet) \to \bot) \to \bot & (\parr_n A)^\bullet &= (\mathbb{N} \to A^{\bot\bullet}) \to \bot \\
(\S^+_n A)^\bullet &= ((\mathbb{N} \to A^\bullet) \to \bot) \to \bot & (\S^-_n A)^\bullet &= (\mathbb{N} \to A^{\bot\bullet}) \to \bot \\
(\exists n : \mathbb{N}.A[n])^\bullet &= (\mathbb{N} \times A[n]^\bullet \to \bot) \to \bot & (\forall n : \mathbb{N}.A[n])^\bullet &= \mathbb{N} \times A[n]^\bullet \to \bot \\
(\exists p : n \le m.A)^\bullet &= A^\bullet & (\forall p : n \le m.A)^\bullet &= A^\bullet
\end{aligned}$$

The translation of tensor and par arrays is a straightforward extension. However, the translation of sequences requires work. Indeed, the double negation translation works only for connectives which have a distinct dual. This restriction means that sequences must be assigned a polarity before the standard double-negation translation can apply: every neutral sequence operator must be refined into either a positive sequence $\S^+_n A$ or a negative sequence $\S^-_n A$, where $(\S^+_n A)^\bot = \S^-_n A^\bot$. The next section explains how to do this polarity assignment.

3.2 Polarized sequences: CLL^n_p

In terms of process semantics, the self-duality of the sequence operators means that neither producer nor the consumer of a sequence controls the program flow. In this section we show how to impose a polarization to sequences; which semantically corresponds to statically assign control to a side of a cut on a sequence.

The reason not to use polarized sequences from the beginning is that cut elimination can already be done on unpolarized sequences. This way, we get two complementary ways of understanding the calculus: one operational, and one denotational. Furthermore, if sequences are used only as an the type of intermediate values, which are later fused away, then the issue of polarization can be side-stepped altogether.

We call the calculus with polarized sequences CLL^n_p. It differs from CLL^n in the following respects:

- The axiom rule only applies to atomic types. One may obtain the equivalent of the axiom rule for other types by applying the eliminators for the positive and negative types in succession.
- The type $\S_n A$ is replaced by the types $\S^+_n A$ and $\S^-_n A$. For both $\S^+_n A$ and $\S^-_n A$, the elements of the sequence are meant to be consumed (respectively produced) in a pre-agreed order. However, when consuming a value of type $\S^-_n A$, it is the producer value that dictates the pace at which the elements are processed (when to move to the next element); as opposed to $\S^+_n A$, where the consumer is the one in control.
- The TRAVERSE rule may eliminate at most one sequence of type $\S^-_n A$, because the sequence controls the rate at which elements are produced.
- Values can be read and written from memory at any pace, therefore, sync and LOOP introduce sequences of type $\S^+_n A$.

In all other respects, the rules for CLL^n_p match those of CLL^n. (Thm. 5 on cut-elimination theorem generalizes directly to CLL^n_p) We now explain the correspondence between both calculi.

Theorem 1. *Every CLL^n_p proof corresponds to a CLL^n proof*

Proof. Replace each occurrence of $\S^-_n A$ or $\S^+_n A$ in the proof by $\S_n A$. The result is a valid CLL^n proof. □

Unfortunately, the map given by Thm. 1 is not surjective: for certain CLL^n proofs, it is not enough to annotate sequences with polarities to obtain an equivalent CLL^n_p proof. Indeed, the alternatives of a case may each place non-local requirements on the polarities which cannot be reconciled.

A sequence polarization scheme There exists an embedding of CLL^n into CLL^n_p, albeit a non-trivial one. The underlying idea is that, when a multiplicative connective is eliminated (a product $A \parr B$ or a $\parr$-array $\parr_n A$), the producer and the consumer will exchange information on which element of the product will ultimately have control of the execution flow. This flow will ultimately depend on the types in the context, and how they are split among the different branches of the $\parr$ and coslice rules.

Definition 2. *We define the positive A^+ and negative A^- translations of a type as follows.*

$$\begin{aligned}
(A \parr B)^+ &= (A^- \parr B^+) \,\&\, (A^+ \parr B^-) \\
(\parr_n A)^+ &= \parr_n A^- \,\&\, (\forall i : \mathbb{N}.\, \parr_i A^- \parr A^+ \parr \parr_{n-i-1} A^-) \\
(\S_n A)^+ &= \S^+_n A^+
\end{aligned}$$

The translation of all other connectives is merely structural, e.g. $(A \otimes B)^+ = A^+ \otimes B^+$. *Additionally, we define* $A^- = A^{\perp +\perp}$

Theorem 2 ((Sequence polarization))**.** *If* $\Delta \vdash$ *holds in* CLL^n*, then*

- $\Delta^+ \vdash$ *holds in* CLL^n_p*, and*
- *for every* Γ *and* X *such that* $\Delta = \Gamma, X$ *then* $\Gamma^+, X^- \vdash$ *holds in* CLL^n_p*.*

Proof. The translation proceeds by induction on the proof derivation. For each elimination rule there will be two cases, depending on whether the name being eliminated is in Γ (positive name, e.g. z^+) or in X (negative name, e.g. z^-).

When the translation operator commutes with a connective, the rule can be applied as-is. When the translated type *offers* a choice, it is resolved in such way that the invariants are preserved. In the following example, because the left branch has negative name of type X, the negative name is introduced in the right branch:

$$\left(\frac{\Gamma, g : X, x : A \vdash \mathrm{a} \qquad \Delta, y : B \vdash \mathrm{b}}{\Gamma, g : X, z : A \parr B, \Delta \vdash} \parr\right)^-_g =$$

$$\frac{\dfrac{\Gamma^+, g^- : X^-, x^+ : A^+ \vdash \mathrm{a} \qquad \Delta^+, y^- : B^- \vdash \mathrm{b}}{\Gamma^+, g^- : X^-, \Delta^+, x : A^+ \parr B^- \vdash} \parr}{\Gamma^+, g^- : X^-, z^+ : (A^- \parr B^+) \,\&\, (A^+ \parr B^-), \Delta^+ \vdash} \&_2$$

When the translated type *forces* a choice, all the alternatives introduce at most one negative name. For example,

$$\left(\frac{\Gamma, x : A, y : B \vdash \mathrm{a}}{\Gamma, z : A \otimes B \vdash} \otimes\right)^-_z =$$

$$\frac{\dfrac{\Gamma^+, x^+ : A^+, y^- : B^- \vdash \mathrm{a}}{\Gamma^+, z^- : A^+ \otimes B^- \vdash} \otimes \qquad \dfrac{\Gamma^+, x^- : A^-, y^+ : B^+ \vdash \mathrm{a}}{\Gamma^+, z^- : A^- \otimes B^+ \vdash} \otimes}{\Gamma^+, z^- : A^+ \otimes B^- \oplus A^- \otimes B^+ \vdash} \oplus$$

In particular, when a TRAVERSE rule is applied, there will only be at most one negative name in the context, and therefore, at most one negative sequence.

□

Theorem 3. *Cut elimination commutes with the polarizing translation.*

Proof. Assume that an instance of fuse in derivation D can be eliminated in n steps. By case analysis, for each single rewriting step, the polarizing translation commutes. By induction, the polarizing translation commutes with the application of all steps, and, in turn, with the elimination of any number of fuse instances. □

3.3 From CLL^n_p proofs to λ-terms

While our double-negation embedding follows the standard pattern, the addition of new constructions, such as n-ary occurrences of types in a context or memory primitives, requires some care.

We chose to represent A^n as two values $\mathbb{N}, \mathbb{N} \to A^\bullet$, where the first value is the index of the first element. Recall that a scalar type A in the context is in fact a shorthand for A^1, and represented as such. Thus when applying an scalar eliminator, the variable is indexed by applying the first component to the second.

$$\begin{array}{l}(\mathrm{let}\ x, y = z; \mathrm{a})^\bullet = \\ \mathrm{let}\ s,\ \mu = z\ \mathrm{in}\ \mu\ s\ (\lambda(x,y) \mapsto \mathrm{let}\ y_0 = (0,\ \lambda\_.\ y)\ \mathrm{in} \\ \mathrm{let}\ x_0 = (0,\ \lambda\_.\ x)\ \mathrm{in}\ a^\bullet\,[\Gamma, x_0, y_0])\end{array}$$

When such n-ary contexts are split (in coslice, traverse and cut), we re-index. For example:

$$\begin{array}{l}(\mathrm{coslice}\ z\{x \mapsto_n \mathrm{a}; y \mapsto_m \mathrm{b}\})^\bullet = \\ \mathrm{let}\ s,\ \mu = z\ \mathrm{in}\ \mu\ s\ (\lambda i. \\ \mathrm{if}\ 0 \le i \wedge i < n\ \mathrm{then}\ \lambda x. \\ \quad \mathrm{let}\ x_0 = (0,\ \lambda\_.\ \lambda\kappa.\ \kappa \propto x)\ \mathrm{in}\ \mathrm{let}\ s_0,\ \mu_0 = \Gamma\ \mathrm{in} \\ \quad (\lambda\gamma_0.\ a^\bullet\,[\gamma_0, x_0])\ (i + s_0,\ \mu_0) \\ \mathrm{else}\ \lambda y.\ \mathrm{let}\ y_0 = (0,\ \lambda\_.\ \lambda\kappa_0.\ \kappa_0 \propto y)\ \mathrm{in} \\ \quad \mathrm{let}\ s_1,\ \mu_1 = \Delta\ \mathrm{in}\ (\lambda\delta_0.\ b^\bullet\,[\delta_0, y_0])\ (-n + i + s_1,\ \mu_1))\end{array}$$

Second, the rules sync and LOOP require a notion of memory. Memory is modeled as a key-value store, where each key of type $\mathcal{K}_d$ corresponds to a value of type d. Memory accesses can be embeded in the effect type by assuming the following functions. We need to allocate : $(\mathcal{K}_d \to \perp) \to \perp$ new keys, write : $(\mathcal{K}_d \to d) \to \perp$ values to them, and read : $(\mathcal{K}_d \to d \to \perp) \to \perp$ them back. The above structure can be constructed on top of any monoid, following Thielemann [29]. Using this structure, sync can be translated as follows:

$$\begin{array}{l}(\mathrm{sync}\{x : \mathbb{A}^{\perp n} \mapsto \mathrm{a}; y : \mathbb{A}^n \mapsto \mathrm{b}\})^\bullet = \\ \mathrm{allocate}\ (\lambda d_0.\ \ldots\ (\mathrm{allocate}\ (\lambda d_{n-1}. \\ (\mathrm{let}\ x = (0,\ \lambda\_.\ (n,\ \lambda i.\ \mathrm{write}\ d_i))\ \mathrm{in}\ a^\bullet\,[\Gamma, x]) \gg \\ (\mathrm{let}\ y = (n,\ \lambda i_0.\ \mathrm{read}\ d_i)\ \mathrm{in}\ b^\bullet\,[\Delta, y])))))\end{array}$$

In sum, by composing sequence polarisation and the double-negation embedding, we obtain a translation from a CLL^n term $\Gamma \vdash t$ into a term $\Gamma^\bullet \vdash t^\bullet : \perp$ in the polymorphic lambda calculus. For this translation, cut elimination results in semantically equivalent terms (Thm. 4).

Theorem 4. *For any two programs* a *and* b *communicating via type* A*,* $\mathrm{fuse}\{x : A^n \mapsto \mathrm{a}; y : A^\perp \mapsto \mathrm{b}\}^\bullet \approx_{\beta\eta} \mathrm{cut}\{x : A^n \mapsto \mathrm{a}; y : A^\perp \mapsto \mathrm{b}\}^\bullet$

Proof. By case analysis on each cut-elimination rule.

For example, when eliminating a cut between the consumer of a $\parr$ with the consumer of a $\otimes$:

$$\boxed{\mathrm{n}\parr\otimes}$$

$$(\mathrm{fuse}\{z : \bigparr_{n+m} A^\perp \mapsto \mathrm{coslice}\ z\{x \mapsto_n \mathrm{a}; y \mapsto_m \mathrm{c}\}; \bar{z} : \bigotimes_{n+m} A \mapsto$$

$$\mathrm{let}\ \bar{x} = \mathrm{slice}\ \bar{z}; \mathrm{b}\})^\bullet \stackrel{def}{=} \mathrm{let}\ \bar{z} = (0,\ \lambda z.\, \mathrm{let}\ z_0 =$$

$$(0,\ \lambda\_.\ z)\,\mathrm{in}\,\mathrm{let}\ s_0,\ \mu_0 = z_0\,\mathrm{in}\,\mu_0\ s_0\ (\lambda i.\,\mathrm{if}\,0 \le i \wedge i <$$

$$n\,\mathrm{then}\,\lambda x.\,\mathrm{let}\ s_1,\ \mu_1 =$$

$$\Gamma\,\mathrm{in}\,(\lambda\gamma_0.\,a^\bullet\,[\gamma_0, x])\,(i + s_1,\ \mu_1)\,\mathrm{else}\,\lambda y.\,\mathrm{let}\ s_2,\ \mu_2 =$$

$$\Delta\,\mathrm{in}\,(\lambda\delta_0.\,c^\bullet\,[\delta_0, y])\,(-n + i + s_2,\ \mu_2)))\,\mathrm{in}\,\mathrm{let}\ s,\ \mu =$$

$$\bar{z}\,\mathrm{in}\,\mu\ s\,(\lambda\bar{x}.\,\mathrm{let}\ \bar{x_0} = (n + m,\ \bar{x})\,\mathrm{in}\,b^\bullet\,[\xi, \bar{x_0}]) \approx \mathrm{let}\ x =$$

$$(0,\ \lambda i_0.\,\lambda x_0.\,\mathrm{let}\ s_0,\ \mu_0 = \Gamma\,\mathrm{in}\,(\lambda\gamma_0.\,a^\bullet\,[\gamma_0, x_0])\,(i_0 +$$

$$s_0,\ \mu_0))\,\mathrm{in}\,\mathrm{let}\ y = (0,\ \lambda i.\,\lambda y_0.\,\mathrm{let}\ s,\ \mu =$$

$$\Delta\,\mathrm{in}\,(\lambda\delta_0.\,c^\bullet\,[\delta_0, y_0])\,(i + s,\ \mu))\,\mathrm{in}\,\mathrm{let}\ \bar{x} =$$

$$(\mathrm{merge}\,m\,y\,x)\,\mathrm{in}\,b^\bullet\,[\xi, \bar{x}] \stackrel{def}{=} (\mathrm{fuse}\{x : A^\perp \mapsto \mathrm{a}; x : A^n \mapsto$$

$$\mathrm{fuse}\{y : A^\perp \mapsto \mathrm{c}; y : A^m \mapsto \mathrm{let}\ \bar{x} = \mathrm{merge}\ x, y; \mathrm{b}\}\})^\bullet$$

We define $\mathrm{merge}\, m\, x\, y$ as concatenating the two arrays in this manner: $\mathrm{merge}\, n\, (s,\ f)\, (t,\ g) \equiv (0,\ \lambda i.\, \mathrm{if}\, i \le n\, \mathrm{then}\, f\, (s+i)\, \mathrm{else}\, g\, (t+i-n))$ □

4. Cost-free abstractions

When writting complex programs, programmers want to define building blocks in isolation, 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.

4.1 Guaranteed fusion

Our cut-elimination algorithm is based on structural cut-elimination in CLL [23], with a one-sided sequent presentation [32].

Theorem 5. *Every 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 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:

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

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.

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

We proceed by mutual induction on the programs being fused and the intermediate type. When both programs start by eliminating the type being fused 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 back together. The binary case looks like this:

$$\frac{\dfrac{\Gamma, A^\perp \vdash \mathrm{a} \qquad \Delta, A^\perp \vdash \mathrm{c}}{\Gamma^n, \Delta^m, \parr_{n+m} A^\perp \vdash}\ \parr \qquad \dfrac{\Xi, A^{n+m} \vdash \mathrm{b}}{\Xi, \bigotimes_{n+m} A \vdash}\ \otimes}{\Gamma^n, \Delta^m, \Xi \vdash}\ \textsc{Fuse} \implies$$

$$\frac{\dfrac{\Gamma, A^\perp \vdash \mathrm{a} \qquad \dfrac{\Delta, A^\perp \vdash \mathrm{c} \qquad \dfrac{\Xi, A^{n+m} \vdash \mathrm{b}}{\Xi, A^n, A^m \vdash}\ \textsc{Merge}_n}{\Delta^m, \Xi, A^n \vdash}\ \textsc{Fuse}_m}{\Gamma^n, \Delta^m, \Xi \vdash}\ \textsc{Fuse}_n$$

  The merge construction is 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. In turn, coslice and traverse use the merged array in exactly one of the branches, which 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.

$$\frac{\dfrac{\dfrac{\Gamma, A^\perp \vdash \mathrm{a}}{\Gamma^{n+m}, \S_{n+m} A^\perp \vdash}\ \S \qquad \dfrac{\dfrac{\Delta, A \vdash \mathrm{b} \qquad \Xi, A \vdash \mathrm{c}}{\Delta^n, \Xi^m, \S_{n+m} A \vdash}\ \textsc{Fuse}}{\Delta^n, \Xi^m, \S_{n+m} A \vdash}\ \S}{\Gamma^{n+m}, \Delta^n, \Xi^m \vdash}\ \textsc{Fuse} \implies$$

$$\frac{\dfrac{\Gamma, A^\perp \vdash \mathrm{a} \qquad \Delta, A \vdash \mathrm{b}}{\Delta, \Gamma \vdash}\ \textsc{Fuse} \qquad \dfrac{\Gamma, A^\perp \vdash \mathrm{a} \qquad \Xi, A \vdash \mathrm{c}}{\Xi, \Gamma \vdash}\ \textsc{Fuse}}{\dfrac{\Delta^n, \Xi^m, \Gamma^n, \Gamma^m \vdash}{\Gamma^{n+m}, \Delta^n, \Xi^m \vdash}\ \textsc{Split}_n}\ \S$$

Cut elimination (and, in particular, the merge rule) requires deciding inequalities between sizes. If the relationship between the sizes cannot be inferred statically, then a single dynamic test is introduced.

This additional dynamic check does not foil our goal of guaranteed improvement through fusion. First, cut-elimination avoids the branching performed by the coslice or traverse for every array element. Because arrays can have arbitrary size, this tests would have costed more in general than any fixed number of dynamic tests introduced by cut. Second, the introduction of dynamic checks can be avoided altogether by introducing a suitable precondition (see Sec. 2.3).

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

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

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 every fuse eventually becomes ready.

The n-ary case $\textsc{Fuse}_n$ can be reduced to the unary one by commuting the size introducing A^n. This side will always be non-ready as long as $n \neq 1$, because no rule can eliminate a general n-ary variable.

Non-eliminating rules such as a cut, sync or loop commute with fuse. This means that allocation in a subprogram does not compromise fusion at an outer level.

Finally, our language can be extended at will with primitive atomic types and operations over them, as is done in our prototype compiler. In such an extension, cuts on primitive types will not always be eliminated, but left in a ready state. Because these values are immediately used, our goal of eliminating intermediate storage of result values is still fulfilled. Furthermore, in a computational interpretation of the calculus, the operands of the primitive operations are small and fit easily in a fast-access processor register.

4.2 Guaranteed improvement

We can now show that the fusion process described in the previous section does not increase the running time of the program.

Cost measure We measure the cost of a program as the number of β-reductions required to reduce its semantics to a normal form. Because of linearity, whether a call-by-name, call-by-value or call-by-need strategy is used is of limited relevance.

The n-ary types in contexts introduce sizes and indices, which are non-linear. Indexing into a variable is considered a β-reduction (cost 1), the same for multiplication. The marginal cost of additions on index variables is assumed negligible. As for pattern matching, we assume a cost of 1 for each application of if or case. Note that, because of linearity, we can amortize the cost of β-reductions inside λ-abstractions when computing the cost measure. On the other hand, let statements are used for clarity and should be read as syntactic substitutions with no run-time cost.

Branching Additionally, to be able to precisely count the number of reductions we need to know which side of each branch will be taken at run time. Therefore the measure depends on an environment giving this value. We additionally assert that the environment is consistent with whatever choices are made by the &-eliminators. (In effect this environment is an oracle.) In sum, if $\Gamma \vdash$ a and γ is a valid branch-predicting environment for Γ, then $|\mathrm{a}|[\gamma]$ is a natural number.

Theorem 6. *For any two programs a and b communicating via type A:*

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

Proof. By case analysis on each cut-elimination rule. % All the cases are found in the appendix.

For example, going back to the rule for $\bigotimes_n A$:

$$\boxed{\mathrm{n}\parr\otimes}$$

$$|\mathrm{fuse}\{z : \parr_{n+m} A^\perp \mapsto \mathrm{coslice}\ z\{x \mapsto_n \mathrm{a}; y \mapsto_m \mathrm{c}\}; \bar{z} : \bigotimes_{n+m} A \mapsto$$

$$\mathrm{let}\ \bar{x} = \mathrm{slice}\ \bar{z}; \mathrm{b}\}|[\Gamma, \Delta, \xi] =$$

$$2 + \sum\nolimits_n (1 + 3 \cdot \#\mathrm{v}(\Gamma) + \chi_+(A^\perp) + |\mathrm{a}|[\Gamma, x]) + \sum\nolimits_m (1 + 3 \cdot$$

$$\#\mathrm{v}(\Delta) + \chi_+(A^\perp) + |\mathrm{c}|[\Delta, y]) + 2 + |\mathrm{b}|[\xi, \bar{x}] \geq \sum\nolimits_n (\chi_+(A^\perp) +$$

$$3 \cdot \#\mathrm{v}(\Gamma) + |\mathrm{a}|[\Gamma, x]) + \sum\nolimits_m (\chi_+(A^\perp) + 3 \cdot \#\mathrm{v}(\Delta) +$$

$$|\mathrm{c}|[\Delta, y]) + n + m + |\mathrm{b}|[\xi, \bar{x}] = |\mathrm{fuse}\{x : A^\perp \mapsto \mathrm{a}; x : A^n \mapsto$$

$$\mathrm{fuse}\{y : A^\perp \mapsto \mathrm{c}; y : A^m \mapsto \mathrm{let}\ \bar{x} = \mathrm{merge}\ x, y; \mathrm{b}\}\}|[\Gamma, \Delta, \xi]$$

Here, $\#v(\Gamma)$ counts the number of variables in a context Γ. If A is a positive type, then $\chi_+(A) = 1$, otherwise $\chi_+(A) = 0$. □

5. Examples and benchmarks

To see how CLL^n scales to more complex examples, 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.

5.1 Methodology

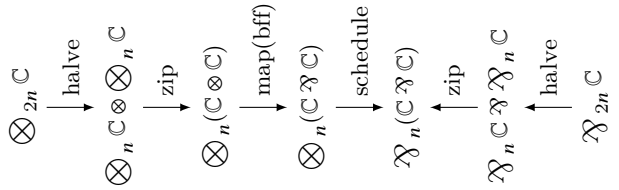
The generated code is compiled using GCC 6.2.1. The programs are run and timed on a Fedora 24, 2×8GB 1333MHz RAM, Intel i7 2630QM machine. Optimization level `-O3` is used to control for domain-agnostic transformations that state-of-the-art compilers can already perform, so that the unfused code is not unfairly disadvantaged. The running time (user space) is averaged over between 10 and 100 repetitions; the value is chosen for each benchmark to minimize variance. For each of the examples, two versions of the C code are generated by alternatively enabling and disabling cut elimination in the prototype compiler. 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 Cooley-Tukey radix-2 in-time decimation. At the core of this algorithm is the size-two discrete Fourier transform:

$$\mathrm{bff} \equiv (\mathrm{x}_0, \mathrm{x}_1) : \mathbb{C} \otimes \mathbb{C},\ (\mathrm{y}_0, \mathrm{y}_1) : \mathbb{C}^\perp \otimes \mathbb{C}^\perp \vdash$$
$$y_0 = x_0 + x_1 w_n^k$$
$$y_1 = x_0 - x_1 w_n^k$$

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 $\log n$ 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.



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 program in our framework. After fusion we obtain an efficient implementation which does not compute nor allocate any intermediate arrays. The generated code consists on one single loop which 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 time step t the displacement of the medium at each point can be computed from the displacements of that point and its neighbors at times $t-1$ and $t-2$.

If the stencil computation is simple enough to be performed with a small number of processor registers, 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. As in Sec. 2.9, we use a sequence type to implement this stencil operation. Because each step depends on the two previous ones, the input (and, therefore, the output), must have type $\S_n(\mathbb{R} \otimes \mathbb{R})$; and the full computation type $\S_n(\mathbb{R} \otimes \mathbb{R}) \multimap \S_n(\mathbb{R} \otimes \mathbb{R})$. The full stencil computation is realized as a *stream transformer* arrow in the spirit of Liu et al. [20]. Even if this involves higher-order combinators (e.g. sequences of functions), all these intermediate abstractions are fused away, leaving behind only elementary array operations.

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 repetitions, 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. [19], the heart of this algorithm is a `filterMax`-like operation, which from a single array of points produces both i) an array of points above a given line segment, and, among these points, ii) the one furthest away from it. For best performance, this operation should be made in with only one pass over the array. At the same time, we would like to write both operations i) and ii) independently.

For this to happen, we need to take an original array of points (e.g. $\S_n(\mathbb{R} \otimes \mathbb{R})$), and obtain from it a pair of sequences (e.g. $\S_n(\mathbb{R} \otimes \mathbb{R}) \parr \S_n(\mathbb{R} \otimes \mathbb{R})$).

$$
\begin{array}{l}
\mathrm{filterMaxLike} \equiv \Gamma, \Delta, a : \S_n(\mathbb{R} \otimes \mathbb{R}) \vdash \\
\mathrm{cut}\{b : \S_n(\mathbb{R}^\perp \parr \mathbb{R}^\perp) \otimes \S_n(\mathbb{R}^\perp \parr \mathbb{R}^\perp) \mapsto \\
\quad \text{let } c, d = b; \\
\quad \mathrm{traverse}\{a \text{ as } a_0, c \text{ as } c_1, d \text{ as } d_1 \mapsto_n \\
\qquad \mathrm{sync}\{x : \mathbb{R}^\perp \parr \mathbb{R}^\perp \mapsto a_0 \leftrightarrow x \\
\qquad\quad y : \mathbb{R} \otimes \mathbb{R}^2 \mapsto \text{let } c_0, d_0 = \mathrm{split}_1\ y; \\
\qquad\qquad\qquad \mathrm{mix}\{c_0 \leftrightarrow c_1;\ d_0 \leftrightarrow d_1\}\}\} \\
\quad b_0 : \S_n(\mathbb{R} \otimes \mathbb{R}) \parr \S_n(\mathbb{R} \otimes \mathbb{R}) \mapsto \\
\qquad \mathrm{connect}\ b_0\ \mathrm{to}\{c_2 \mapsto \mathrm{filter}[\Gamma, c_2];\ d_2 \mapsto \max[\Delta, d_2]\}\}
\end{array}
$$

The programs filter and $\max$ will each perform one traversal on the array. However, once all cuts are eliminated, only one single TRAVERSE over the input sequence will remain. Only a small tuple $\mathbb{R} \otimes \mathbb{R}$ needs to be allocated at each step with SYNC^2, instead of a full array.

5.5 Summary of results

The running times for the examples are summarized in Fig. 7. By applying cut elimination and a straightforward compilation scheme, it is possible to run programs rich in abstraction in times and resembling those of hand-optimized C code. These results show how the guaranteed fusion enables compositional programming in practical settings.

There is a discrepancy between Plain C and Fused for QuickHull. The increase in cost in the CLL^n version is due to the use of intermediate values of type $A \oplus 1$ when filtering and selecting points. This modeling choice has no overhead inside the kernels themselves, because fusion avoids the potential branching. But it still increases the cost where the program uses intermediate storage; namely when sending results between the different filterMax-like kernels that make the full QuickHull algorithm, or when introducing an accumulator with a LOOP construct.

In the case of wave propagation, the CLL^n version of the code performs better because it accesses each element of the input array exactly once, while the Plain C version reads them once for each application of the stencil in which they intervene.

Algorithm	Unfused	Fused	Plain C	Library
FFT kernel	127 ns	92 ns	92 ns	2.5 ns
Wave propagation	223 ns	2.5 ns	2.7 ns	1.8 ns
QuickHull	612 ns	58 ns	35 ns	210 ns

Figure 7: Benchmark summary. Code generated before and after applying fusion is compared to our own hand-optimized C program, and a state-of-the-art implementation (FFTW, QHull, Physis). Time is average CPU time divided by the number of input elements (QuickHull), or computed elements (wave propagation, FFT).

Finally, note that the state-of-the-art implementations may be heavily optimized down to the assembly instruction levels (e.g. FFTW) or more general in purpose (QHull), so comparisons must be drawn with care. The code measured is available at `https://lopezjuan.com/limestone`, revision v0.0.1-2.

6. Discussion

Size arithmetic Restricting size expressions to linear integer arithmetic might prove restrictive if one wants to iterate over an array in two dimensions, or use a fully general SYNC^k_n rule.

In the compiler implementation, we extend the grammar for size expressions as follows:

$$n, m ::= \cdots \quad | \quad n \cdot m$$

The product of two size expressions is always normalized by using the ring axioms (distributivity etc.). This yields a linear combination of monomials, each of which is treated as an independent variable. Furthermore, when applying a rule that changes the size of the context, such as $\parr$, the resulting sizes for each variable can be inferred by multivariate polynomial division (the remainder must be 0). This approach suffices for most practical examples involving unbounded array dimensions. If desired, congruence can be recovered by adding a primitive rule inhabiting the type $\forall \alpha : \mathbb{N}. \forall \beta : \mathbb{N}. \forall \gamma : \mathbb{N}. \forall p : \alpha \le \beta. \exists q : \alpha\gamma \le \beta\gamma.1$.

Other applications of LIA solvers to functional programming, such as Rondon et al. [27], treat multiplication as an uninterpreted function, preserving congruence at the expense of distributivity.

Code reuse and polymorphism Code reuse is achieved by giving names to derivations, which are substituted at the points where the their names are invoked. Cut elimination is then applied to merge the caller and callee derivations together.

For conciseness, this paper presents a version of LL without second order quantification. By contrast, the prototype does support a form of static polymorphism by generalizing the $\forall$ and $\exists$ connectives to range over type variables. Any cut introducing requiring that any cut introducing them is eliminated before type-checking and code-generation. This way we generalize many of the patterns in the examples can be generalized into reusable combinators.

Exponentials and sync 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. As we saw in Sec. 2.6, some common applications of exponentials can be realized in CLL^n by combining existing constructs, albeit in a sometimes verbose way.

For our calculus, adding exponentials would mean foregoing either guaranteed fusion (Thm. 5) or guaranteed improvement (Thm. 6). 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; may be used many times by means of storing the value.

Front-end The examples programs are written by combining several basic functions (map, dot,filter,...) which are in turn generated as internal representations of CLL^n trees by a set of combinators.

There are several ways in which CLL^n can be turned into a fully-fledged platform. One possibility is to write a stand-alone language on top. On the other hand, given the precise control of evaluation and ordering, CLL^n is well positioned to become an additional layer of syntax on top of an imperative language, such as C or Fortran. In this setting, CLL^n would provide a more principled, safer way of describing and fusing array operations, while the fine-grained computation is implemented in the imperative language.

6.1 Related Work

Fusion as Cut-Elimination Framing fusion as cut-elimination in a sequent calculus was first proposed by Marlow [21]. 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 [31] did in his seminal paper. By basing our work on a linear sequent calculus, we get additional benefits, such as the improvement guarantee.

Non-commutative linear logic and state The notion of a non-commutative multiplicative binary linear operator ($\triangleright$) was explored in depth by Retoré [26], and Reddy [25] uses it together with a dagger modality ($\dagger$) to model state.

Our system uses an n-ary version § of these non-commutative connectives. By making non-commutativity a property only of the connective, and not also of the context, we ensure that non-commutativity does not interfere with the fusion properties of the calculus, and, in particular, does not require the use of concurrent processes for its implementation.

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 [16] 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].

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." In effect, we need to provide suitable array primitives (Sec. 2.3), so that we can recover some programming power and write useful programs.

Another difference with the field of implicit complexity is in the goals we pursue. 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 run time. 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 improve performance.

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. First, pull arrays, also known as delayed arrays [18], 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]. 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. 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` represents programs which can write to memory, loop, and spawn parallel computations. The value of a push array can be recovered 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 to 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 [15], 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 . Push arrays suffer from a similar problem: unless a pull array calls the program it receives exactly once for each index, some elements will be undefined or, in a parallel setting, subject to race conditions.

Both problems stem from 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. A consequence of this choice is that operations such as pull array concatenation, while inefficient in the functional representation, behave well in our framework (Sec. 2.9). We believe that the choice of a classical, linear framework reveals the essential duality of push and pull arrays.

Repa The library Repa [18] 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 [17]. 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 requires many basic combinators to be defined as primitive notions by the library. Finally, the lack of linearity may still cause the duplication of computation.

Data-flow Fusion Lippmeier et al. [19] 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.

Ling Pouillard [24] is developing Ling, a programming language based on linear logic sporting a rich user-facing syntax. It implements many the constructs of CLL^n. It has a distinction between

data types and session types, which incorporates and expands on the distinction between size variables and linear variables.

Ling also sports a form of fusion. The transformation does not operate on the program terms themselves, but instead, the program is instantiated with concrete, constant sizes before being converted to a sequence of statements and loops. Then, fusion of these statements can be attempted. Cost and fusion guarantees, as well as the interaction with dynamic sizes, are still unexplored in Ling.

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 [30], deforestation [31] and fixed point promotion [22]. 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 will fail. Furthermore, they typically rely on rewriting whole functions, whereas shortcut fusion and its descendants relies on local rewrite rules which can be easily incorporated into an optimizing compiler.

Session types Wadler [32] shows that CLL corresponds to a process calculus free from deadlock. We propose a different correspondence from CLL derivations into programs. Freedom from deadlock in the process calculus corresponds to termination in the λ-calculus.

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 appears 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, linearity appears daunting, severely constraining the kind of programs one can write; but in time, one enjoys the cost-free abstractions granted by guaranteed fusion.

References

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

[2] E. Axelsson et al. Feldspar: A domain specific language for digital signal processing algorithms. In *MEMOCODE*, 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] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi, T. King, A. Reynolds, and C. Tinelli. Cvc4. volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer-Verlag, 2011.

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

[16] 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.

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

[18] B. Lippmeier, 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.

[19] 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.

[20] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows and their optimization. In *ACM Sigplan Notices*, volume 44, pages 35–46. ACM, 2009.

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

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

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

[24] N. Pouillard. The ling language - modular and precise resource management, 2015. URL `https://nicolaspouillard.fr/talks/ling-32c3/`.

[25] U. S. Reddy. A linear logic model of state. 1993.

[26] C. Retoré. *Ordered sequents and proof nets*. PhD thesis.

[27] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.

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

[29] H. Thielemann. monoid-transformer: Monoid counterparts to some ubiquitous monad transformers.

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

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

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