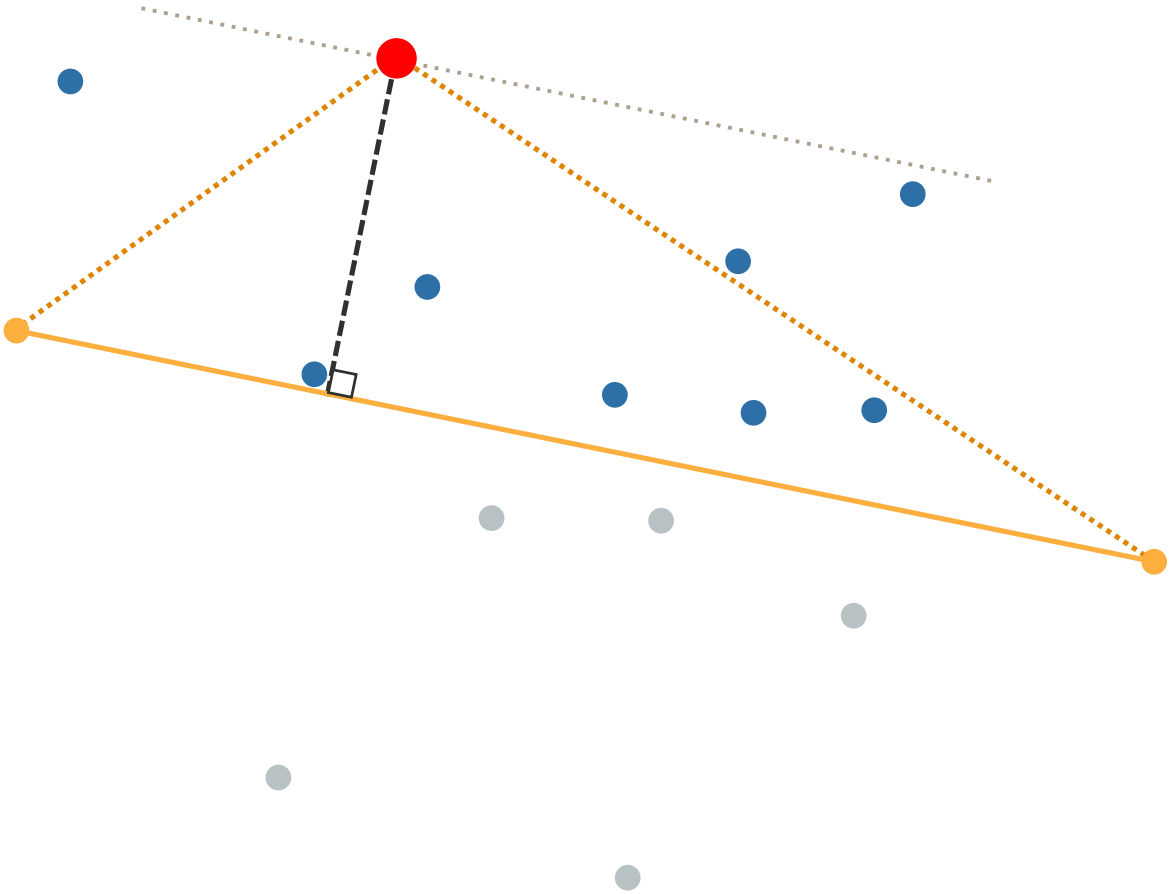


Efficient Functional Programming using Linear Types: The Array Fragment

Master's thesis in Computer Science

VÍCTOR LÓPEZ JUAN

Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2015

MASTER'S THESIS IN COMPUTER SCIENCE

Efficient Functional Programming using Linear Types: The Array Fragment

VÍCTOR LÓPEZ JUAN

Computer Science and Engineering
Computer Science Division
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2015

Efficient Functional Programming using Linear Types: The Array Fragment
VÍCTOR LÓPEZ JUAN

© VÍCTOR LÓPEZ JUAN, 2015

Master's thesis 2015:01
ISSN 1652-8557
Computer Science and Engineering
Computer Science Division
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Göteborg, Sweden 2015

Efficient Functional Programming using Linear Types: The Array Fragment
Master's thesis in Computer Science
VÍCTOR LÓPEZ JUAN
Computer Science and Engineering
Computer Science Division
Chalmers University of Technology

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.

Linear types serve as a tool to specify the desired behaviour (sharing or duplication). This means that the programmer is able to predict the performance of composition solely from the types of the composed functions. Girard's linear logic (LL) can be extended with vector types and a synchronization primitive, making a linear language with array manipulation capabilities.

In this master thesis, we improve on the n-ary extension of LL by introducing a self-dual, sequential array operator with an aggregation function. We provide a functional interpretation of the resulting calculus, which forms the basis for a low-level code generator.

We then illustrate the effectiveness of the extended 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 (1-dimensional stencil, QuickHull).

Keywords: Array-Programming, Classical Linear Logic

Acknowledgements

I thank my supervisor, Jean-Philippe Bernardy, for the thesis idea, his critical feedback throughout, and introducing me to the intricacies of academic research.

I thank Josef Sveningsson and Dan Rosén, who have followed up on our progress, and on whose work this thesis builds.

I am grateful to each member of the ProgLog group and the CSE department, who have made me feel at home from the first day. Our conversations have taught me that the beauty of logical systems goes beyond pragmatism.

I must show my appreciation to my examiner, Bengt Nordström, and to Patrik Jansson, for their support and their invaluable help navigating all related logistical matters.

I cannot thank Irene Lobo Valbuena and Fabian Ruch enough for their earnest opposition, their meticulous analysis of the final text, and their friendship.

And I will not forget my parents' continued encouragement despite the distance.

This work was made possible by a 2013 "la Caixa" scholarship.

Contents

1 Motivation

1.1 High-performance computing

A key application of computers is the modelling of real-world phenomena, such as the weather, aerodynamics, molecules, language or vision; by means of numerical computation. High-performance computing (HPC) aims to maximize the throughput of numerical computer programs by scheduling work efficiently and with minimal overhead. Increasing the throughput permits finer models for increased accuracy, or shorter computation times.

An important metric when comparing HPC platforms is the theoretical maximum number of floating-point operations per second (FLOPS) they can perform. In most architectures, the circuitry for performing floating-point operations is specialized, and cannot be used for any other purpose. Thus, an upper bound on the computational capacity can be obtained by looking at the throughput of these components.

How close the running system will get to the maximum throughput is highly dependent on the application. For a particular computer program, some proportion of its instructions will be floating-point operations, whereas the rest will be dedicated to logical decisions (branching), and, especially if written in higher-level languages, book-keeping and scheduling. Because these non-functional aspects of the program share the instruction pipeline with the arithmetic operations, they may cause floating-point throughput to suffer.

Traditionally, programmers use low-level languages such as Fortran or C for high-performance computing. Both of them closely reflect how the machine works, and afford very fine control over CPU and memory usage. This way, programmers can ensure that the program will perform as expected, with as little overhead as necessary.

1.2 Finite differences

To show how the trade-offs in high-performance computing pan out in practice, we will introduce finite differences as a driving example that will be used through the report. The `diff` function computes differences between each pair of consecutive elements of a finite vector. It is used in algorithms that approximate derivatives or interpolate polynomials. At the same time, it is a particular case of the more general class of stencil algorithms.

$$\{\mathrm{diff}_i(x)\}_{i=1}^{n} \equiv x_{i+1} - x_i$$

Note that, in order for the definition to make sense, x_i has to be defined from $i = 1$ to $i = n + 1$.

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, or as part of non-linear interpolation), one applies `diff` twice:

$$\mathrm{diff}^2(x) \equiv \mathrm{diff}(\mathrm{diff}(x))$$

This definition can be implemented as the following C function:

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

Observe that communicating the two applications of `diff` requires an intermediate array 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 shown below:

```
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];
}
```

1.3 Functional programming

The function `diff2_fused` has the same meaning and better performance characteristics than `diff2`. However, it is not clear that the two functions are equivalent; the compositionality of the definition of `diff2` is lost with the manual optimization.

Software engineering practice encourage programmers to structure their programs in modules that can be composed and reused. This way, programmers can write less code, saving time and reducing bugs. Each programming paradigm gives a different definition of what the components are, and how they communicate (Fig. 1.1).

Paradigm	Modules	Communication	Glue
Object-oriented	Objects	Messages	Inheritance, aggregation
Logic	Predicates	Rules	Entailment
Imperative	Procedures	Shared state	Procedure calls
Dataflow	Processes	Data input and output	Pipes
(Lazy) functional	Functions	Application (and evaluation)	Function composition, higher-order functions

Figure 1.1: *Comparison of programming paradigms. Some popular programming paradigms, each encompassing a large number of programming languages, are compared. Paradigms are characterized by the building blocks or modules that the programmer can define, and the way that these blocks can be glued together and communicate with each other.*

In his famous position paper, Hughes [1989] argues that a key advantage of lazy functional programming is that programs can be written by composition of simple building blocks, while retaining good runtime behaviour. This is realized by providing two new methods of program composition:

Higher-order functions Functions can be parametrized by arbitrary fragments of code. Without this feature, different versions of the program would need to be created for each different parameter that we would want to splice in.

Lazy evaluation The result of a function can be computed when needed, or not at all, and doesn't need to be held in memory all at once. In other languages, this would require specialized functions for each usage pattern of the result.

In the spirit of functional programming, `diff2` would be implemented 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 piece-wise, but one still pays the overhead of allocating each individual thunk. This overhead is acceptable when implementing IO-bound applications such as web services; but is excessive in a high-performance setting.

1.4 A case for linearity

Our goal is too keep the advantages granted by higher-order functions and lazy evaluation; while minimizing the overhead enough that the resulting language is suitable for HPC.

An intuitive approach to this goal, used in several functional DSLs [Keller et al., 2010, Axelsson et al., 2010, Svensson, 2011, Chakravarty et al., 2011, Elliott et al., 2003], is to represent an immutable array of type A by a function $Int \to A$; and deferring the reification of such functions into in-memory arrays to a later phase. Using such a representation, we can define diff2 as follows:

```
diff , diff2 :: Num a ⇒(Int →a) → (Int → a)
diff  f = \i → f (i + 1) − f i
diff2 f = diff (diff f)
```

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 sufficiently 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 optimizations.* We instead aim to give the programmer strong guarantees on the behaviour of the generated code.

Type systems are used by programming languages to give guarantees at compile time about the behaviour of the program at runtime. The earliest use case of types involves making judgements on the possible values that a variable may take, and how they will be represented. For example, it can guarantee that a floating point number will not be interpreted as an integer, or a reference to a string as a reference to a function. More advanced type systems can also express effects that a function may perform, such as input-output, mutation, or exceptions, and guarantee that the effects of a program at runtime will be limited to those appearing in its type.

In our approach, we enrich types with information about the usage and the scheduling of values. If a program is well-typed, we can guarantee that intermediate results can be fused away, without repeating computation.

2 Linear logic

We build our language on Linear Logic, presented as a sequent calculus by Girard [1987]. In its smallest version, it contains binary products and sums.

When presenting linear logic, we will make frequent references to the intuitive idea of processes communicating through channels. However, modelling concurrency is not the aim of our semantics. As we will show, it is possible to implement all of the connectives without the need for multiple execution contexts.

Types, terms and contexts A typed calculus such as linear logic is given by a set of *types*, a set of *terms*, and *typing rules* that relate pairs of a term and its type. Typically, terms are programs, and the type of a term indicates what sort of inputs the program expects, and the outputs that the program will produce as a result of these inputs. A well-known example of typed calculus is the simply-typed lambda calculus. If we interpret types as propositions, and terms as proofs (known as the Curry-Howard isomorphism), then the simply-typed lambda calculus corresponds to intuitionistic logic.

The typing relation between terms and programs can depend on a *context*. Each statement of the form "the term t has type T in context Γ" (written $\Gamma \vdash t : T$) is called a judgement. The context is a collection of names, or *bindings*, each of which has a type. They correspond to variables in scope when the program is interpreted.

Bindings are of the form $x : A$, where x is a name, and A is a type. Capital Greek letters (e.g. Γ) stand for a comma-separated list of zero or more bindings. We follow the convention that when contexts are mentioned on either side of a comma, they must be disjoint. Bindings may also be referred to as variables. The order of variables in a context is not relevant.

In our system we will have a single type for terms, the type of terminating programs. Alternatively, one may think of the context in which a term is well-typed as the dual type of the term.

Typing derivations A typed calculus defines whether a term and a type are related. This relationship is defined through a set of *typing rules*, from which new judgements can be derived from existing ones. A term has a type in a given context if it is possible to derive this judgement from the set of rules, without any initial judgements. The application of the typing rules defines a tree structure, which is called a *derivation tree*, or *derivation*. Each rule application is read as "if ⟨derivations above the bar are sound⟩ then ⟨judgement below the bar holds⟩".

Weakening and contraction In many type systems, there exist two rules, which may be given either explicitly or implicitly, called *weakening* and *contraction*. These rules are classified as *structural rules*, because they operate only on contexts.

A weakening rule states that we do not need to use all of the bindings in the context, we can ignore some of them (Fig. 2.1a). A logic with weakening is called monotonic, because the set of valid programs in a given context does not decrease when new elements are added to the context.

A contraction rule states that we may use some of the bindings in the context more than once (Fig. 2.1b). A logic with contraction is said to have idempotence of entailment, because adding a new bindings to the left of the entails symbol ($\vdash$) does not change the provability of the judgement.

$$\frac{\overline{\Gamma \vdash}}{\Gamma, x : A \vdash}$$

(a) *Weakening*

$$\frac{\overline{\Gamma, x : A, y : A \vdash}}{\Gamma, x : A \vdash}$$

(b) *Contraction*

Figure 2.1: *Linear logic* lacks *weakening and contraction rules.*

A logic is linear when it lacks the rules of weakening and contraction. In contrast, an affine logic has weakening, but lacks contraction. Both linear and affine logics are called *substructural*, because they are obtained by removing one or more structural rules from a more general calculus.

2.1 Girard's involutive new logic

Girard [1987] introduced linear logic, as a way of giving a more granular description of the constructive properties of intuitionistic logic.

Calculi based on propositional logic have a notion of *conjunction* (P and Q), *disjunction* (P or Q). Linear logic introduces one orthogonal notion; namely, whether a connective is additive or multiplicative. Additive binary connectives allow all elements of the context to be used when each of the operands is used; whereas multiplicative connectives require an implicit or explicit division of the context between the operands.

Together, we obtain four operators: multiplicative conjunction (A $\otimes$ B), multiplicative disjunction (A $\parr$ B), additive conjunction (A & B) and additive disjunction (A $\oplus$ B).

This proliferation of connectives is tamed by the introduction of an involutive negation $(.)^\perp$, where $A^{\perp\perp} \cong A$. This negation exhibits De Morgan duality both for the additive and the multiplicative fragments:

$$(A \otimes B)^\perp \cong A^\perp \parr B^\perp$$

$$(A \oplus B)^\perp \cong A^\perp \& B^\perp$$

Observe that the distinction between multiplicative and additive would be moot in the presence of weakening and contraction, because we can duplicate and remove parts of the context at will before and after the division.

We will denote Girard's linear logic as Classical Linear Logic (CLL), emphasizing the difference with intuitionistic linear logic, which lacks the involutive negation operator. Note that the use of the term "classical" is controversial, because the resulting system, unlike most classical logics, remains constructive at its core.

Duality Every type in CLL has a dual, as can be seen in Fig. 2.2, whose dual is the original type itself. In propositional calculi, a self-involutive negation, or excluded middle, would make the logic non-constructive: we cannot obtain an object just from the negation of its non-existence.

In a linear logic, a value of type $A^\perp$ is an obligation to provide a value of type A, which, because of linearity, must be fulfilled. A value of type $A^{\perp\perp}$ implies an obligation to provide an obligation to provide A, which one may see as equivalent an obligation to consume A; that is, a value of type A itself.

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

Figure 2.2: *List of type connectives in [Girard, 1987]. The types in the left column are dual to the types in the right column, and* vice versa*.*

In particular, this involution property allows us to remove all distinctions between inputs and results: a program producing an A is equivalent to a program consuming $A^\perp$.

A calculus for computation We are not only interested in the logical properties of CLL, but also in the computational interpretation. As per the Curry-Howard isomorphism, derivations in CLL will correspond to computer programs. 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.

Examples can also help obtain an intuition of the meaning of a formal system. For some rules, we will provide an informal interpretation in terms of processes that communicate through variables, in the spirit of Wadler [2012].

The full set of typing rules for the system are listed in Fig. 3.1. We will now introduce the rules in detail, each of them together with their dual. If the derivation is well-typed, the behaviour of the program will be "correct". In the π-calculus interpretation, this translates into freedom from deadlocks.

We will later provide a more formal interpretation of the terms as an embedding into the lambda calculus. As we will see, no concurrent composition of programs is required.

2.2 The AX rule

The axiom rule is a polymorphic elimination rule. It eliminates a value of type A and a value of type $A^\perp$ by "connecting" them ($\leftrightarrow$), and terminates the computation.

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

Under the linear types as session types interpretation, the AX rule connects two endpoints with a channel.

In imperative programming, AX expresses the idea of *assignment to a variable* or *returning* from a function. In contrast to these concepts, the AX rule is completely symmetrical: the roles of assigned and assignee can be reversed thanks to duality.

From a logical point of view, the AX rule concludes a proof from the conjunction of A and $\neg A$, which embodies the principle of non-contradiction.

2.3 The CUT rule

While AX is a polymorphic eliminator, cut is a polymorphic introduction rule. Together they are the only structural rules in CLL.

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

Each of the two sides of the cut is parametrized over a new name ($\mapsto$); the types of these names are duals of each other. The context is statically split between the sides when the cut is performed[1].

From a logical point of view, cut expresses that, at any point, either A or not A, that is, the principle of *excluded middle*.

In terms of processes, the cut rule intuitively creates a "communication channel" through which the two sides "communicate". Because the contexts are disjoint, and the communication channel is unique, there communication graph will be a tree. Thus, cyclic dependencies which could lead to deadlock are avoided.

Notice how, thanks to duality, cut becomes a polymorphic introduction rule; it will suffice to provide eliminators for each of the connectives (for example, see Fig. 2.4).

Cut elimination In mathematics and computer science, lemmas (respectively, constant bindings) are mere abstraction tools: they make the presentation of proofs and programs clearer, but do not essentially change the propositions that we can proof, or the programs that we may write.

The notion of cut is an artifact of the sequent calculus presentation, which logicians seek to eliminate. Calculi where every proof can be transformed into a cut-free proof are said to fulfil a *cut-elimination theorem*.

In CLL, all instances of cut can be eliminated. Furthermore, for a given proof, it is possible to specify which cuts should be kept, and which should be eliminated.

To make the notion of cut elimination more precise, we introduce an additional admissible rule, fuse. It is semantically equivalent to cut, but, instead of being part of the calculus, it will be defined inductively on the type A.

[1] Which binding is used by which side may be inferred by a smart compiler, but the decision still needs to be static.

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

Our cut-elimination algorithm is based on structural cut-elimination in CLL [Pfenning, 1995], with a one-sided sequent presentation [Wadler, 2012].

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

Proof. The admissible rule fuse is defined inductively on the type A, without introducing any use of cut. Thus, replacing each instance of cut with fuse yields a derivation without cut. □

For the proof of Thm. 1 to be complete, we need to define the fuse rule. We will do so as we explain each of the connectives in the system.

The definition will be given as a rewrite system. An instance of the fuse rule can be rewritten in terms of "*smaller*" instances of fuse.

For each branch of fuse, consider the largest distance to a rule where the introduced variable is eliminated. That number is the depth of the branch.

Ready state Each variable is immediately eliminated by a rule (the depth of both branches is 0). The two elimination rules "cancel out", and fuse now works over a smaller type.

Non-ready state One of the branches has depth larger than 0. Then, the lowest rule in that branch does not work on the introduced variable; so it is possible to apply it before the cut is introduced. In other words, we *commute* the first rule in the branch with the cut. The depth of the branch is now 1 unit smaller.

Theorem 2. *The definition of* fuse *is well-founded.*

Proof. Each rewrite step reduces either the size of the type (ready state), or the size of (one of) the branches (non-ready state). Thus, the rewrite always terminates. □

We will start by defining fuse for the cases where the rules closest to the branch are AX and CUT. For each rule we introduce, we will use both the sequent notation, and a program notation.

The AX rule can only be in ready state, because it eliminates all the variables in the context. When one of the sides of the cut is an axiom rule, cut elimination corresponds to *variable renaming.*

$$\boxed{\text{Ax}}$$

$$\frac{\Gamma, A^\perp \vdash a[x] \qquad \dfrac{}{A^\perp, A \vdash}\ \textsc{Ax}}{\Gamma, A^\perp \vdash}\ \textsc{Fuse} \Longrightarrow\ \Gamma, A^\perp \vdash a[w]$$

$$\mathsf{fuse}\{x : A^\perp \mapsto a[x];\ y : A \mapsto y \leftrightarrow w\} \Longrightarrow a[w]$$

The situation for cut is the opposite to that for AX: it does not eliminate any variables, so it will always be in non-ready state. The only possible rewrite is to commute the cut with the fuse.

$$\boxed{\kappa\text{cut}}$$

$$\frac{\Gamma, A^\perp \vdash a \qquad \dfrac{\Delta, B^\perp \vdash b \qquad \Xi, A, B \vdash c}{\Delta, \Xi, A \vdash}\ \textsc{Cut}}{\Gamma, \Delta, \Xi \vdash}\ \textsc{Fuse} \Longrightarrow \frac{\Delta, B^\perp \vdash b \qquad \dfrac{\Xi, B, A \vdash c \qquad \Gamma, A^\perp \vdash a}{\Gamma, \Xi, B \vdash}\ \textsc{Fuse}}{\Gamma, \Delta, \Xi \vdash}\ \textsc{Cut}$$

$$\mathsf{fuse}\{x : A^\perp \mapsto a \\ \quad y : A \mapsto \mathsf{cut}\{v : B^\perp \mapsto b;\ w : B \mapsto c\}\} \Longrightarrow$$

$$\mathsf{cut}\{v : B^\perp \mapsto b \\ \quad w : B \mapsto \mathsf{fuse}\{y : A \mapsto c;\ x : A^\perp \mapsto a\}\}$$

Because cut can commute with fuse, the programmer can choose which cuts to eliminate, and which to preserve.

2.4 Atomic types ($\mathbb{Z}$, $\mathbb{R}$, ...)

The syntax for linear logic supports atomic types. In computing, these include primitive machine values, such as integers ($\mathbb{Z}$) or double-precision floating-point ($\mathbb{R}$).

The only polymorphic way of eliminating an atom is by means of the axiom rule:

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

In effect, because each logical connective has its own elimination rule, it is possible to restrict the axiom rule to atomic types without losing expressive power. When applied to non-atomic types, one may consider the axiom rule as an admissible rule defined inductively on the types of the variables.

However, real-world applications require programs to manipulate primitive values in non-trivial ways. This includes introducing literals into the context, and reducing them using operations specific to their type. These are *primitive operations*, which can be embedded in the system as shown in Fig. 2.3. We give only a few examples; the concrete set of primitive operations is orthogonal to the rest of the calculus, and highly dependent on the targeted platform.

$$z : \mathbb{Z}^\perp \vdash z \leftrightarrow 0 \qquad x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z}^\perp \vdash z \leftrightarrow +[x, y] \qquad x : \mathbb{Z}, y : \mathbb{Z}, z : (1 \oplus 1)^\perp \vdash z \leftrightarrow \leq[x, y]$$

(a) *Numeric literal* (b) *Addition operation* (c) *Comparison operation*

Figure 2.3: *Examples of primitive operations in* CLL^n. *The type* $1 \oplus 1$ *is the type of booleans; it is considered primitive for the purposes of cut elimination.*

Cut elimination When one of the variables of the fuse operation is used by a primitive operation, the fuse must be converted back into a cut. If we allow for primitive operations, then the cut elimination theorem must be nuanced:

Theorem 3. *For every derivation in CLL there exists a derivation where all instances of* cut *are in* ready *state, and one of their sides is a primitive operation.*

Proof. Because primitive operations eliminate the whole context, commute rules for them are not required.

- If either side is not ready, it is not primitive. Apply the corresponding commute rule.
- If any of the sides is a primitive operation, replace by a cut; which will fulfil the premises of the theorem.

□

The operands of the primitive operations are small and fit easily in a fast-access location (such as a register). Furthermore, because the rules will be in ready state, these values will be immediately used. Therefore, our goal of eliminating the need for intermediate storage of result values is still fulfilled.

2.5 The multiplicative fragment ($\parr$,$\otimes$)

Most type systems allow for building new types by *aggregating* several smaller types into a larger one. In functional programming one can form the product type $A \times B$, whose values are pairs $(a, b) : A \times B$, $a : A$ and $b : B$.

For each pair of types A and B CLL has two product types: the tensor product ($A \otimes B$), and the par product, ($A \parr B$).

The tensor product ($\otimes$)

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

In the interpretation as processes, a name $x : A \otimes B$ can produce, when read, two names $x : A$ and $y : B$. These new names may be used at any point without the risk of deadlock.

The par product ($\parr$) In the interpretation as processes, a name $z : A \parr B$ also produces, when read, two names $x : A$ and $y : B$.

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

As opposed to the tensor ($\otimes$) case, x and y cannot be used immediately in any order we wish. For example, in the case $z : \mathbb{Z}^\perp \parr \mathbb{Z}$, producing names $x : \mathbb{Z}^\perp$ and $y : \mathbb{Z}$, y might not produce any data until x is fully consumed, or vice versa.

To guarantee freedom from deadlock, the names should be consumed in independent processes; each operating on a separate part of the context.

Introduction rules Introduction of every type can be expressed with cut, and the eliminator for its dual. Fig. 2.4 shows how to construct a value of type $A \otimes B$ from each of its components (one value of type A, and one value of type B).

$$\dfrac{\dfrac{\dfrac{}{a : A, y_1 : A^\perp \vdash a \leftrightarrow y_1}\ \textsc{Ax} \qquad \dfrac{}{b : B, y_2 : B^\perp \vdash b \leftrightarrow y_2}\ \textsc{Ax}}{a : A, b : B, y : A^\perp \parr B^\perp \vdash \mathsf{connect}\ y\ \mathsf{to}\{y_1 \mapsto a \leftrightarrow y_1; y_2 \mapsto b \leftrightarrow y_2\}}\ \parr \qquad \Gamma, x : A \otimes B \vdash a}{\Gamma, a : A, b : B \vdash \mathsf{cut}\{y : A^\perp \parr B^\perp \mapsto \mathsf{connect}\ y\ \mathsf{to}\{y_1 \mapsto a \leftrightarrow y_1; y_2 \mapsto b \leftrightarrow y_2\}; x : A \otimes B \mapsto a\}}\ \textsc{Cut}$$

$$\begin{array}{l}\Gamma, a : A, b : B \vdash \\ \mathsf{cut}\{y : A^\perp \parr B^\perp \mapsto \\ \qquad \mathsf{connect}\ y\ \mathsf{to}\{y_1 \mapsto a \leftrightarrow y_1;\ y_2 \mapsto b \leftrightarrow y_2\} \\ \quad x : A \otimes B \mapsto a\}\end{array}$$

Figure 2.4: *Introducing a type by eliminating its dual*

The pattern is analogous for all other connectives in the language.

Linear implication ($\multimap$) Intuitionistic presentations of linear logic make use of the $\multimap$ operator to model functions. The type $A \multimap B$ is similar to the functional arrow type $A \to B$, with the difference that the "function" will use its input exactly once, and the output must also be used exactly once.

- $A \multimap B \stackrel{def}{=} A^\perp \parr B$
- $(A \multimap B)^\perp \cong A \otimes B^\perp$

2.5.1 Cut elimination rules

We aim to prove that all cuts can be eliminated. We will now cover the case when the rule in either side of a fuse rule eliminates a multiplicative connective.

Ready state If both sides of fuse are in ready state, then we know that both are, respectively, the elimination rules for $\otimes$ and $\parr$.

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

$$\dfrac{\dfrac{\Gamma, A^\perp \vdash a \quad \Delta, B^\perp \vdash b}{\Gamma, \Delta, A^\perp \parr B^\perp \vdash}\,\parr \quad \dfrac{\Xi, A, B \vdash c}{\Xi, A \otimes B \vdash}\,\otimes}{\Gamma, \Delta, \Xi \vdash}\,\textsc{Fuse} \Longrightarrow \dfrac{\Gamma, A^\perp \vdash a \quad \dfrac{\Delta, B^\perp \vdash b \quad \Xi, A, B \vdash c}{\Delta, \Xi, A \vdash}\,\textsc{Fuse}}{\Gamma, \Delta, \Xi \vdash}\,\textsc{Fuse}$$

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

$$\begin{array}{l}\mathsf{fuse}\{\bar{x} : A^\perp \mapsto a \\ \qquad x : A \mapsto \mathsf{fuse}\{\bar{y} : B^\perp \mapsto b;\ y : B \mapsto c\}\}\end{array}$$

Commuting of tensor ($\otimes$) If the side of fuse containing an eliminator for tensor is not ready, then we need to commute the fuse past the rule.

$$\boxed{\kappa\otimes}$$

$$\dfrac{\Gamma, A^\perp \vdash a \quad \dfrac{\Delta, A, B, C \vdash b}{B \otimes C, \Delta, A \vdash}\,\otimes}{\Gamma, B \otimes C, \Delta \vdash}\,\textsc{Fuse} \Longrightarrow \dfrac{\dfrac{\Delta, B, C, A \vdash b \quad \Gamma, A^\perp \vdash a}{\Gamma, \Delta, B, C \vdash}\,\textsc{Fuse}}{\Gamma, B \otimes C, \Delta \vdash}\,\otimes$$

$$\mathsf{fuse}\{x : A^\perp \mapsto a;\ y : A \mapsto \mathsf{let}\ u, v = w;\ b\} \Longrightarrow$$

$$\mathsf{let}\ u, v = w;\ \mathsf{fuse}\{y : A \mapsto b;\ x : A^\perp \mapsto a\}$$

Commuting of par ($\parr$) If the side of fuse containing an eliminator for par is not ready, then we need to commute the fuse past the rule.

$$\boxed{\kappa\parr}$$

$$\dfrac{\Xi, C^\perp \vdash a \quad \dfrac{\Gamma, A \vdash b \quad \Delta, C, B \vdash c}{\Gamma, A \parr B, \Delta, C \vdash}\,\parr}{\Gamma, A \parr B, \Delta, \Xi \vdash}\,\textsc{Fuse} \Longrightarrow \dfrac{\Gamma, A \vdash b \quad \dfrac{\Delta, B, C \vdash c \quad \Xi, C^\perp \vdash a}{\Delta, \Xi, B \vdash}\,\textsc{Fuse}}{\Gamma, A \parr B, \Delta, \Xi \vdash}\,\parr$$

$$\begin{array}{l}\mathsf{fuse}\{x : C^\perp \mapsto a \\ \qquad y : C \mapsto \mathsf{connect}\ w\ \mathsf{to}\{u \mapsto b;\ v \mapsto c\}\} \Longrightarrow\end{array}$$

$$\begin{array}{l}\mathsf{connect}\ w\ \mathsf{to} \\ \quad \{u \mapsto b;\ v \mapsto \mathsf{fuse}\{y : C \mapsto c;\ x : C^\perp \mapsto a\}\}\end{array}$$

2.5.2 The multiplicative units (1,⊥)

Functional programming languages have a unit type, which corresponds to programs which run successfully, but do not return any output. These are executed for their side effects. Because values of unit type contain no information, they are units for the product type. For example, there is a one-to-one correspondence between values $x : \mathbb{Z}$, and values $z : \mathbb{Z} \times Unit$, where $z = (x, ())$.

CLL has two unit types, one (1) and bottom (⊥). They are respectively units for the tensor ($\otimes$) and par ($\parr$) products.

In functional programming languages, unit values can be discarded, unused, without affecting correctness.

In linear logic, this would violate the exactly once requirement.

Values of type 1 can be freely ignored (i.e. having them in the context does not affect provability). However, we must do so explicitly, because the order in which they are used may be relevant when linearity is used to handle scheduling or side effects.

$$\frac{\Gamma \vdash a}{\Gamma, x : 1 \vdash \mathsf{let}\ \diamond = x; a}\ 1$$

Values of type $\bot$ correspond to non-terminating programs to which we may yield control. As such, they can only be used if there are no other pending values in the context.

$$\frac{}{x : \bot \vdash \mathsf{yield\ to}\ x}\ \bot$$

- $1 \otimes A \cong A$
- $\bot \parr A \cong A$

2.5.3 Example

For example, in order to compute $tan^2 x = \frac{sin^2 x}{cos^2 x}$, we may write the following Haskell program.

```
type float2 = (float , float )

-- Primitives
(+), (−), (*), (/) :: float → float → float
sin2 , cos2 :: float → float

sin2_cos2 :: float → float2
sin2_cos2 a = (sin2 a, cos2 a)

tan2 :: float → float
tan2 a = let (x,y) = sin2_cos2 a in x / y
```

When compiling `tan2`, the compiler may or may not allocate an intermediate tuple for the result of `sin2_cos2 a`, depending on its internal heuristics. In CLL^n, we can make this behaviour explicit [2]:

```
sin2_cos2 : ℝ ⊸ ℝ ⊗ ℝ
sin2_cos2 ≡ { a : ℝ, result : (ℝ ⊗ ℝ)⊥ ↦
              sync { a' ↦ a ↔ a' }
                   { a1 a2 ↦
                     connect result to
                       { r1 ↦ r1 ↔ sin2 1a}
                       { r2 ↦ r2 ↔ cos2 2a} } }

tan2 : ℝ ⊸ ℝ
tan2 ≡ { a : ℝ, result : ℝ⊥ ↦
         cut { xy' ↦ sin2_cos2 a xy' }
             { xy ↦ let x, y = xy result ↔ x / y } } }
```

Because the two values of the tuple are produced independently, cut elimination can completely remove the intermediate tuple (Fig. 2.5)

[2]The sync rule is used for duplicating a value. It provides a way out of linearity for certain types. Its meaning and interaction with cut are explained in detail in Sec. 3.3.

```
tan2 ≡ { a, result ↦
    cut { xy' ↦ sync { a' ↦ a ↔ a' }
              { a1 a2 ↦
              connect xy' to
                  { r1 ↦ r1 ↔ sin2 a1}
                  { r2 ↦ r2 ↔ cos2 a2} } } }
        { xy ↦  let x, y = xy;
                result ↔ x / y } } }
```

(a) *Inlined*

```
tan2 ≡ { a, result ↦
    sync { a' ↦ a ↔ a' } { a1 a2 ↦
        cut { xy' ↦ connect xy' to
                  { r1 ↦ r1 ↔ sin2 a1}
                  { r2 ↦ r2 ↔ cos2 a2} } } }
            { xy ↦  let x, y = xy;
                    result ↔ x / y } } }
```

(b) *Step 1*

```
tan2 ≡ { a, result ↦
    sync { a' ↦ a ↔ a' } { a1 a2 ↦
        cut { r1 ↦ r1 ↔ sin2 a1 }
            { x ↦  cut { r2 ↦ r2 ↔ cos2 a2}
                       { y ↦  result ↔ x / y }
            } } }
```

(c) *Step 2*

```
tan2 ≡ { a, result ↦
    sync { a' ↦ a ↔ a' } { a1 a2 ↦
        result ↔ sin2 a1 / cos2 a2 } }
```

(d) *Cut-free derivation*

Figure 2.5: *Computing squared tangent in* CLL^n *with independent calls to* `sin` *and* `cos`*.*

2.6 The additive fragment ($\oplus$,&)

If the multiplicative fragment is concerned with *scheduling*, the additive fragment of linear logic models *alternatives*.

There are two additive connectives, plus ($\oplus$) and with (&).

Plus connective ($\oplus$) A name has type $(A \oplus B)$ if it will behave either as A or B when consumed. The consumer has no control over whether the behaviour will be A or B, so they must provide derivations for both cases. In other words, the consumer performs case analysis on the possible alternatives.

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

With connective (&) A name has type $(A \& B)$ if it will behave either as A or B when consumed. As opposed to the plus ($\oplus$) case, it is the consumer who chooses whether to pick the left or right alternatives; only the one that is requested will be produced.

$$\frac{\Gamma, x : A \vdash a}{\Gamma, z : A \& B \vdash \mathsf{let}\ \mathsf{inl}\ x = z; a}\ \&_1$$

$$\frac{\Gamma, x : B \vdash a}{\Gamma, z : A \& B \vdash \mathsf{let}\ \mathsf{inr}\ x = z; a}\ \&_2$$

2.6.1 Cut elimination

Ready state In the ready state, cut elimination corresponds to dead-code elimination. If whether $A \oplus B$ will behave as A or B is known statically, then the unused branch in the $\oplus$ rule can be eliminated.

$$\mathsf{fuse}\{z : A^\perp \,\&\, B^\perp \mapsto \mathsf{let\ inl}\ x = z;\ a \\ \bar{z} : A \oplus B \mapsto \mathsf{case}\ \bar{z}\ \mathsf{of}\{\mathsf{inl}\ \bar{x} \mapsto b;\ \mathsf{inr}\ \bar{y} \mapsto c\}\} \Longrightarrow$$

$$\mathsf{fuse}\{x : A^\perp \mapsto a;\ \bar{x} : A \mapsto b\}$$

$$\mathsf{fuse}\{z : A^\perp \,\&\, B^\perp \mapsto \mathsf{let\ inr}\ x = z;\ a \\ \bar{z} : A \oplus B \mapsto \mathsf{case}\ \bar{z}\ \mathsf{of}\{\mathsf{inl}\ \bar{x} \mapsto b;\ \mathsf{inr}\ \bar{y} \mapsto c\}\} \Longrightarrow$$

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

Non-ready state If the fuse is in the non-ready state, the fuse is commuted into each of the alternatives for the additive rule. Unlike the situation with multiplicative connectives, commuting non-ready fuse over the additive fragment may result in an increase in the size of the derivation tree.

$$\mathsf{fuse}\{x : A^\perp \mapsto a;\ y : A \mapsto \mathsf{let\ inl}\ u = w;\ b\} \Longrightarrow$$

$$\mathsf{let\ inl}\ u = w;\ \mathsf{fuse}\{y : A \mapsto b;\ x : A^\perp \mapsto a\}$$

$$\mathsf{fuse}\{x : A^\perp \mapsto a;\ y : A \mapsto \mathsf{let\ inr}\ u = w;\ b\} \Longrightarrow$$

$$\mathsf{let\ inr}\ u = w;\ \mathsf{fuse}\{y : A \mapsto b;\ x : A^\perp \mapsto a\}$$

$$\mathsf{fuse}\{x : A^\perp \mapsto a \\ y : A \mapsto \mathsf{case}\ w\ \mathsf{of}\{\mathsf{inl}\ u \mapsto b;\ \mathsf{inr}\ v \mapsto c\}\} \Longrightarrow$$

$$\mathsf{case}\ w\ \mathsf{of}\{\mathsf{inl}\ u \mapsto \mathsf{fuse}\{y : A \mapsto b;\ x : A^\perp \mapsto a\} \\ \mathsf{inr}\ v \mapsto \mathsf{fuse}\{y : A \mapsto c;\ x : A^\perp \mapsto a\}\}$$

In computational terms, the decision information that was once encoded as an intermediate value is now determined by the position in the program.

2.6.2 The additive units (0, $\top$)

Whereas multiplicative units (Sec. 2.5.2) represent types whose values contain no information (that is, they have only one distinct value); additive units are empty types, with no possible values.

In functional programming languages, the existence of a value of an empty type implies that the current code path is unreachable. There is a one-to-one correspondence between values of type $x : T$ and values $z : 0 + T$, where $z = \mathsf{inr}\ x$, because there are no values for the left alternative.

CLL has an empty type 0, and its dual, $\top$.

The existence of a value of type 0 in the context implies that the current code path will never be executed. Thus, we can immediately halt execution without affecting the correctness of the program.

$$\frac{}{\Gamma, x : 0 \vdash \mathsf{dump}\ \Gamma\ \mathsf{in}\ x}\ 0$$

On the other hand, there is no rule for eliminating values of type $\top$. The existence of such a rule would render every context Γ trivially provable.

$$\frac{\top \vdash \top \qquad \dfrac{}{\Gamma, 0 \vdash}\ 0}{\Gamma \vdash}\ \textsc{Cut}$$

2.6.3 Examples

One straightforward usage of additive connectives is for handling exceptional behaviour. A program can acknowledge the possibility of error by consuming $\mathsf{error}^\perp \,\&\, \mathbb{Z}^\perp$, instead of a plain $\mathbb{Z}^\perp$. The caller of that program must handle both possibilities (`inr` when successful, and `inr` in case of failure), by pattern-matching on $\mathsf{error} \oplus \mathbb{Z}$.

Another more involved scenario arises in languages with lazy or call by name semantics, such as Haskell.

```
sqrt :: Int → Int

ifPositive :: Integer →a → a → a
ifPositive a x _ | a > 0 = x
ifPositive _ _ y         = y

safeSqrt :: Integer →Integer
safeSqrt a = ifPositive a (sqrt a) (sqrt (− a))
```

The partial function `sqrt` is only defined on the positive integers. Thanks to laziness, the erroneous computation (either `sqrt a` or `sqrt (-a)`), is implicitly discarded; but this choice is not made in the same place as the computation is created.

This behaviour can be expressed within CLL. Observe that the lazy thunks `sqrt_pos` and `sqrt_neg` will explicitly discard (`dump`) the values they closed over as soon as they are deemed unnecessary.

```
ifPositive : ℤ ⊸ A&1 ⊸ A&1 ⊸ A
ifPositive a x y result =
  case ( a > 0 : 1 ⊕ 1 ) of
    { inl → let inr = x in let inl x₀ = x in result ↔ x₀ }
    { inr → let inr = x in let inl y₀ = y in result ↔ y₀ }}

safeSqrt : ℤ ⊸ ℤ
safeSqrt a result =
  sync { a } { a₁ a₂ a₃ ↦
      cut { sqrt_pos' ↦ case sqrt_pos'
              { inl result ↦ result ↔ sqrt a₁ }
              { inr bottom ↦dump a₁ ; yield to bottom }
          { sqrt_pos ↦
            cut { case { inl result ↦ result ↔ sqrt (− a₂) }
                       { inr bottom ↦dump a₂ ; yield to bottom } }
                { sqrt_neg ↦ ifPositive a₁ sqrt_pos sqrt_neg result }
          }
      }
```

If compiled directly, the above would involve the creation of closures or thunks for the variables `pos` and `neg`, so that the execution of `sqrt` can be deferred. A good compiler would avoid them, but this is not self-evident to the programmer.

CLL^n makes the lazy semantics explicit, and guarantees that any overhead arising from them can be eliminated. After inlining and applying the cut-elimination rules, we obtain the following:

```
safeSqrt : ℤ ⊸ ℤ
safeSqrt a result =
  sync { a } { a₀ a₁ a₂ ↦
    { case (a₀ > 0)
      { inl → dump a₁; result ↔ sqrt a₁ }
      { inr → dump a₁; result ↔ sqrt (− a₂) } } }
```

2.7 Mix and halt rules

When programming, there are scenarios where there is an excess of freedom. For example, when copying values between disjoint memory locations, the programmer can do so in any order while still avoiding deadlocks.

The programming model described in Sec. 2.5 and Sec. 2.5.2 has perfect duality, which implies that the programmer cannot make any scheduling decisions himself; either the producer or the consumer has to specify an order. The MIX rule allows him to reduce an excess of freedom by forcing a specific split of the context.

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

Another conundrum occurs when a program has an empty context, meaning that it does not need to do any more work. Then it should be able to simply halt without yielding to another program.

$$\frac{}{\vdash \mathsf{halt}}\ \textsc{Halt}$$

Halt corresponds to a process that performs no action. Mixing two derivations corresponds to scheduling the two processes in some arbitrary order.

We additionally require mix to be associative operation on derivations in any model of the calculus, and HALT to be its unit. Under these conditions, MIX and HALT induce a monoid structure on derivations.

Cut elimination Because the MIX rule does not eliminate any names, if this rule appears immediately after an instance fuse, the instance will be in *non-ready state*.

$$\boxed{\kappa\text{mix}}$$

$$\frac{\dfrac{\Gamma, A^\perp \vdash a \qquad \Delta \vdash b}{\Gamma, \Delta, A^\perp \vdash}\ \textsc{Mix} \qquad \Xi, A \vdash c}{\Gamma, \Delta, \Xi \vdash}\ \textsc{Fuse} \implies \frac{\Delta \vdash b \qquad \dfrac{\Gamma, A^\perp \vdash a \qquad \Xi, A \vdash c}{\Gamma, \Xi \vdash}\ \textsc{Fuse}}{\Gamma, \Delta, \Xi \vdash}\ \textsc{Mix}$$

$$\mathsf{fuse}\{x : A^\perp \mapsto \mathsf{mix}\{a;\, b\};\, y : A \mapsto c\} \Longrightarrow$$

$$\mathsf{mix}\{b;\, \mathsf{fuse}\{x : A^\perp \mapsto a;\, y : A \mapsto c\}\}$$

On the other hand, HALT can only be used on an empty context. Because each branch of fuse adds one new variable to the context, HALT cannot appear immediately after fuse in a well-typed derivation, so it can be disregarded when defining the latter.

2.8 Parametric polymorphism (∃,∀)

When manipulating containers (for example, pairs), there are operations which can be written *generically* without depending on the concrete type contained in the structure. Swapping the components of a pair is one such operation.

Even if the generic presentation of a term is (by definition) independent of the values of the type variables, the semantics may differ depending on the type with which the variable is instantiated (for example, swapping a tuple of double-precision values may require bigger machine registers than swapping integers).

The type information required to make these distinctions is conveyed as a *code*. This can range from a full representation of the type to a bare number indicating its size in memory.

Polymorphism is expressed using the universal (∀) and existential (∃) quantifiers, such that $(\exists\alpha.A[\alpha])^\perp \cong \forall\alpha.A[\alpha^\perp]^\perp$.

Universal quantification ($\forall$) Eliminating a value of a universally-quantified type requires providing a concrete type for the type variable.

$$\frac{\Gamma, x : A[B] \vdash a}{\Gamma, z : \forall\alpha.A[\alpha] \vdash \mathsf{let}\ x = z@B; a}\ \forall$$

A name $\forall\alpha.A[\alpha]$ expects a *code* for the type α, and produces a value $A[\alpha]$.

Existential quantification ($\exists$) Eliminating a value of an existentially-quantified type yields a fresh type variable of kind $*$, which is added to an implicit context.

$$\frac{\Gamma, x : A[\beta] \vdash a}{\Gamma, z : \exists\alpha.A[\alpha] \vdash \mathsf{let}\ x\langle\beta\rangle = z; a}\ \exists$$

$\exists\alpha.A[\alpha]$ corresponds to a name yielding a *code* for the type α, and then producing a value of type $A[\alpha]$.

2.8.1 Cut elimination

Non-ready state Commuting with fuse is straightforward, because no context demotions are involved.

$\boxed{\kappa\forall}$

$$\frac{\Gamma, A^\perp \vdash a \qquad \dfrac{\Delta, A, B[B] \vdash b}{\forall\alpha.B[\alpha], \Delta, A \vdash}\ \forall}{\Gamma, \forall\alpha.B[\alpha], \Delta \vdash}\ \textsc{Fuse} \Longrightarrow \frac{\dfrac{\Delta, B[B], A \vdash b \qquad \Gamma, A^\perp \vdash a}{\Gamma, \Delta, B[B] \vdash}\ \textsc{Fuse}}{\Gamma, \forall\alpha.B[\alpha], \Delta \vdash}\ \forall$$

$$\mathsf{fuse}\{x : A^\perp \mapsto a;\ y : A \mapsto \mathsf{let}\ u = w@B;\ b\} \Longrightarrow$$

$$\mathsf{let}\ u = w@B;\ \mathsf{fuse}\{y : A \mapsto b;\ x : A^\perp \mapsto a\}$$

$\boxed{\kappa\exists}$

$$\frac{\Gamma, A^\perp \vdash a \qquad \dfrac{\Delta, A, B[\beta] \vdash b}{\exists\alpha.B[\alpha], \Delta, A \vdash}\ \exists}{\Gamma, \exists\alpha.B[\alpha], \Delta \vdash}\ \textsc{Fuse} \Longrightarrow \frac{\dfrac{\Delta, B[\beta], A \vdash b \qquad \Gamma, A^\perp \vdash a}{\Gamma, \Delta, B[\beta] \vdash}\ \textsc{Fuse}}{\Gamma, \exists\alpha.B[\alpha], \Delta \vdash}\ \exists$$

$$\mathsf{fuse}\{x : A^\perp \mapsto a;\ y : A \mapsto \mathsf{let}\ u\langle\beta\rangle = w;\ b\} \Longrightarrow$$

$$\mathsf{let}\ u\langle\beta\rangle = w;\ \mathsf{fuse}\{y : A \mapsto b;\ x : A^\perp \mapsto a\}$$

Ready state Cut elimination corresponds to specialization of the generic code; or, in other words, variable substitution.

$\boxed{\forall\exists}$

$$\frac{\dfrac{\Gamma, A[B^\perp]^\perp \vdash a}{\Gamma, \forall\alpha : k.A[\alpha^\perp]^\perp \vdash}\ \forall \qquad \dfrac{\Xi, A[\beta] \vdash b}{\Xi, \exists\alpha : k.A[\alpha] \vdash}\ \exists}{\Gamma, \Xi \vdash}\ \textsc{Fuse} \Longrightarrow \frac{\Gamma, A[B^\perp]^\perp \vdash a \qquad \Xi, A[B] \vdash b}{\Gamma, \Xi \vdash}\ \textsc{Fuse}$$

$$\begin{array}{r}\mathsf{fuse}\{z : \forall\alpha : k.A[\alpha^\perp]^\perp \mapsto \mathsf{let}\ x = z@B;\ a \\ \bar{z} : \exists\alpha : k.A[\alpha] \mapsto \mathsf{let}\ \bar{x}\langle\beta\rangle = \bar{z};\ b\} \quad \Longrightarrow\end{array}$$

$$\mathsf{fuse}\{x : A[B^\perp]^\perp \mapsto a;\ \bar{x} : A[B] \mapsto b\}$$

3 An n-ary extension to CLL

Girard [1987] intended for his logic to subsume (non-linear) intuitionistic logic. With the connectives we have introduced so far, we cannot write programs much more interesting than "swapping the components of a pair".

To recover the expressive power lost through linearity, Girard [1987] introduces exponentials. For each type A, we have $!A$, and $?A$, with $(!A)^\perp \cong ?A^\perp$. The key distinction between $!A/?A$ and a plain A is that the former provide a general escape hatch from linearity: values of the form $!A$ admit weakening, and values of the form $?A$ can be contracted (Sec. 2).

Finally, with exponentials, the intuitionistic implication $A \to B$ can be embedded as $!A \multimap ?B$. Thus, the simply typed lambda calculus can be embedded in LL.

Of course, this flexibility does not come for free: it is much harder to express cost guarantees if exponentials are allowed. As we see in this chapter, an n-ary extension is a more controlled way of achieving similar expressiveness.

As an alternative, Bernardy and Svenningsson exclude exponentials, instead extending the system with n-ary array types. In this section, we will explain how this extension works, and demonstrate that the resulting calculus is enough to describe useful programs.

Contexts We first extend the syntax of contexts with special support for n-ary variables. ($\Gamma ::= - \mid \Gamma,\ x : A^n$).

While A^n may be intuitively understood as n values of type A, the binding $x : A^n$ introduces a single variable x.

As a rule, the eliminators for all connectives operate on names of arity 1. Accessing individual values is done using special-purpose rules, which we discuss below.

We omit the superscript when it is equal to 1. If the arity of a name is 0, it is equivalent to that name not actually being in the context; in particular, if all names in a context have arity 0, the HALT rule can be applied.

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.

Sizes are polynomials over size variables, which reside in an implicit, non-linear context. For this explanation, we assume that we have an oracle that can compare any two polynomials, and assess divisibility. In Sec. 6 we address how this oracle can be realized in a practical implementation.

N-ary cuts The CLL^n calculus introduces a managed way of iterating and duplicating computation: the n-ary cut, which generalizes the unary cut from Chapter 2:

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

as well as the corresponding fuse, whose definition we will explain in this chapter:

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

In both rules, one of the introduced variables has arity n (left side), and the other always has arity 1 (right side). The left-side program will run the program on the right side each time that the name of arity n (x) is used, which, by linearity, will be exactly n times.

Notice how the arity of the names in Δ is multiplied by n below at the root of the rule, compared to the context above the bar. This is an example of context *demotion*, and is the main mechanism for indexing into an n-ary variable and retrieving the individual values. Each time it is called, the right side will use the corresponding part of Δ for producing the result (y).

Implications of n-ary fuse on cut elimination The introduction of n-ary fuse does not affect the case when both branches are ready, because eliminators only act on variables of size 1.

A key observation is that, whenever an instance of fuse of arity n is not ready, *the branch of arity n can always be commuted*. Thus, when handling the non-ready case in fuse, we can restrict ourselves to this branch.

Because the rules in Chapter 2 do not change the arity of bindings, their associated commuting rules can be applied directly to the n-ary case.

Split and merge Variables of arity n can be split into two new variables of arities a and b, as long as $n = a + b$.

$$\frac{\Gamma, x : A^n, y : A^m \vdash a}{\Gamma, z : A^{n+m} \vdash \mathsf{let}\ x, y = \mathsf{split}_n\ z; a}\ \textsc{Split}_n$$

Ax

$$\frac{\dfrac{\Gamma, A^m, A^n \vdash a}{\Gamma, A^{n+m} \vdash}\ \textsc{Split}_m \qquad \Delta, A^\perp \vdash b}{\Gamma, \Delta^{n+m} \vdash}\ \textsc{Fuse}_{n+m} \implies \frac{\dfrac{\Delta, A^\perp \vdash b \qquad \Gamma, A^m, A^n \vdash a}{\Gamma, \Delta^n, A^m \vdash}\ \textsc{Fuse}_n \qquad \Delta, A^\perp \vdash b}{\dfrac{\Gamma, \Delta^m, \Delta^n \vdash}{\Gamma, \Delta^{n+m} \vdash}\ \textsc{Split}_m}\ \textsc{Fuse}_m$$

$$\mathsf{fuse}\ \{z : A^{n+m} \mapsto \mathsf{let}\ x, y = \mathsf{split}_m\ z;\ a;\ \bar{z} : A^\perp \mapsto b\} \implies$$

$$\begin{array}{l}\mathsf{let}\ \Delta, \Delta = \mathsf{split}_m\ \Delta; \\ \mathsf{fuse}\{\bar{z} : A^\perp \mapsto b \\ \quad x : A^m \mapsto \mathsf{fuse}\{\bar{z} : A^\perp \mapsto b;\ y : A^n \mapsto a\}\}\end{array}$$

It is also possible to define a merge rule, to join together two variables of arities n and m into a new variable of arity $n + m$.

$$\frac{\Gamma, x : A^{n+m} \vdash a}{\Gamma, z : A^m, y : A^n \vdash \mathsf{let}\ x = \mathsf{merge}\ z, y; a}\ \textsc{Merge}_m$$

Similarly to fuse, this rule is admissible; that is, we can transform any program making use of this rule into one which does not.

To achieve this, we need to handle three cases, the first two of which are straightforward:

Ready case The variable introduced by merge is immediately eliminated, which implies that its arity is 1. Because all arities are ≥ 0, one of the names being merged must have arity zero, and can be silently ignored.

Non-ready case, non-demoted If the arity of the name is not changed by the rule, then merge commutes with it without changes. This is the case for all the rules described in Chapter 2, except for SPLIT.

In the case of SPLIT, a test is performed to check how the split point relates to the sizes of names being merged; only then can they commute.

Non-ready case, demoted This is the non-trivial case. When the arity of a variable changes, the behaviour of merge depends on the particular rule.

merge-n⅋

$$\frac{\dfrac{\Delta, A, B \vdash a}{\parr_{m+n} B, \Delta^{m+n}, A^{m+n} \vdash}\ \parr}{\parr_{m+n} B, A^m, A^n, \Delta^{m+n} \vdash}\ \textsc{Merge}_m \implies \frac{\dfrac{A, \Delta, B \vdash a \qquad A, \Delta, B \vdash a}{\parr_{m+n} B, A^m, A^n, \Delta^m, \Delta^n \vdash}\ \parr}{\parr_{m+n} B, A^m, A^n, \Delta^{m+n} \vdash}\ \textsc{Split}_m$$

3.1 N-ary products or arrays ($\bigotimes_n A$, $\parr_n A$)

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.

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

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

⅋-Arrays The dual of the tensor array is the ⅋-array, written ($\parr_n A$). By duality, the consumer of a ⅋-array must be able to consume its elements in any given order. Hence, the eliminator of $\parr_n A$ (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 one of the n parts the context was divided in.

At runtime, a will be executed n times.

$$\frac{\Gamma, x : A \vdash a}{\Gamma^n, z : \parr_n A \vdash \mathsf{coslice}\ z\{x \mapsto_n a\}}\ \parr$$

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.

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

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

Cut elimination rules

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

$$\frac{\dfrac{\Gamma, A^\bot \vdash a \qquad \Delta, A^\bot \vdash c}{\Gamma^n, \Delta^m, \parr_{n+m} A^\bot \vdash}\ \parr \qquad \dfrac{\Xi, A^{n+m} \vdash b}{\Xi, \bigotimes_{n+m} A \vdash}\ \otimes}{\Gamma^n, \Delta^m, \Xi \vdash}\ \textsc{Fuse} \quad\Longrightarrow\quad \frac{\Gamma, A^\bot \vdash a \qquad \dfrac{\dfrac{\Delta, A^\bot \vdash c \qquad \dfrac{\Xi, A^{n+m} \vdash b}{\Xi, A^n, A^m \vdash}\ \textsc{Merge}_n}{\Delta^m, \Xi, A^n \vdash}\ \textsc{Fuse}_m}{\Gamma^n, \Delta^m, \Xi \vdash}\ \textsc{Fuse}_n$$

$$\begin{array}{l}\mathsf{fuse}\{z : \parr_{n+m} A^\bot \mapsto \mathsf{coslice}\ z\{x \mapsto_n a;\ y \mapsto_m c\} \\ \quad\ \bar{z} : \bigotimes_{n+m} A \mapsto \mathsf{let}\ \bar{x} = \mathsf{slice}\ \bar{z};\ b\}\end{array} \quad\Longrightarrow$$

$$\begin{array}{l}\mathsf{fuse}\{x : A^\bot \mapsto a \\ \qquad x : A^n \mapsto \mathsf{fuse}\{y : A^\bot \mapsto c \\ \qquad\qquad\qquad y : A^m \mapsto \mathsf{let}\ \bar{x} = \mathsf{merge}\ x, y;\ b\}\}\end{array}$$

Commuting of n-ary tensor (⊗) If the side of fuse containing an eliminator for tensor is not ready, then we need to commute the fuse past the rule.

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

$$\frac{\Gamma, A^\bot \vdash a \qquad \dfrac{\Delta, A^n, B^n \vdash b}{\bigotimes_n B, \Delta, A^n \vdash}\ \otimes}{\Gamma^n, \bigotimes_n B, \Delta \vdash}\ \textsc{Fuse}_n \quad\Longrightarrow\quad \frac{\dfrac{\Delta, B^n, A^n \vdash b \qquad \Gamma, A^\bot \vdash a}{\Gamma^n, \Delta, B^n \vdash}\ \textsc{Fuse}_n}{\Gamma^n, \bigotimes_n B, \Delta \vdash}\ \otimes$$

$$\mathsf{fuse}\{x : A^{\perp} \mapsto a;\ y : A^n \mapsto \mathsf{let}\ z = \mathsf{slice}\ w;\ b\} \Longrightarrow$$

$$\mathsf{let}\ z = \mathsf{slice}\ w;\ \mathsf{fuse}\{y : A^n \mapsto b;\ x : A^{\perp} \mapsto a\}$$

Commuting of n-ary par ($\parr$) If the side of fuse containing an eliminator for par is not ready, then we need to commute the fuse past the rule.

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

$$\frac{\Gamma, A^{\perp} \vdash a \qquad \dfrac{\Delta, A, B \vdash b \qquad \Xi, B \vdash c}{\Xi^m, \parr_{n+m} B, \Delta^n, A^n \vdash}\ \parr}{\Gamma^n, \Xi^m, \parr_{n+m} B, \Delta^n \vdash}\ \textsc{Fuse}_n \Longrightarrow \frac{\dfrac{\Delta, B, A \vdash b \qquad \Gamma, A^{\perp} \vdash a}{\Gamma, \Delta, B \vdash}\ \textsc{Fuse} \qquad \Xi, B \vdash c}{\Gamma^n, \Xi^m, \parr_{n+m} B, \Delta^n \vdash}\ \parr$$

$$\mathsf{fuse}\{x : A^{\perp} \mapsto a \quad y : A^n \mapsto \mathsf{coslice}\ w\{u \mapsto_n b;\ v \mapsto_m c\}\} \Longrightarrow$$

$$\mathsf{coslice}\ w\ \{u \mapsto_n \mathsf{fuse}\{y : A \mapsto b;\ x : A^{\perp} \mapsto a\};\ v \mapsto_m c\}$$

3.2 Size polymorphism

In a CLL^n derivation, size variables are symbolic and receive concrete values at runtime. Size variables used in the types of a context are bound in an implicit context and have kind $\mathbb{N}$. They share the context with regular type variables, which have kind $*$.

Functions that operate on values of different sizes generically are typed analogously to parametrically polymorphic functions, by means of the $\forall$ and $\exists$ connectives defined in Sec. 2.8.

Finally, type checking derivations using symbolic sizes may require the programmer to specify bounds on sizes. This is achieved by introducing a kind $n \leq m$ for each pair of sizes n and m. This kind is inhabited by type-level witnesses of the given inequality. Witnesses carry no information other than their mere existence.

3.3 Data types

Linearity arguably prevents the general use of weakening and contraction. This means that we cannot discard any general value. If this were the case, we could introduce values of any type just by performing a cut and discarding the left branch, rendering the system trivial (and useless).

However, this does not mean that no values support weakening and contraction. For example, we can introduce and remove any amount of values of type 1:

$$\frac{\dfrac{}{\perp \vdash}\ \perp \qquad \Gamma, 1, 1 \vdash a}{\Gamma, 1 \vdash}\ \textsc{Cut}$$

$$\frac{\Gamma \vdash a}{\Gamma, 1 \vdash}\ 1$$

Information in the form of bits can be duplicated and discarded in a similar way:

$$\frac{\dfrac{\dfrac{}{\perp \vdash}\ \perp}{\perp \& \perp \vdash}\ \&_1 \qquad \Gamma, (1 \oplus 1)^2 \vdash a}{\dfrac{\Gamma \vdash}{\Gamma, 1 \vdash}\ 1 \qquad \dfrac{\Gamma \vdash}{\Gamma, 1 \vdash}\ 1}\ \textsc{Cut}_2 \qquad \frac{\dfrac{\dfrac{}{\perp \vdash}\ \perp}{\perp \& \perp \vdash}\ \&_2 \qquad \Gamma, (1 \oplus 1)^2 \vdash a}{\Gamma, 1 \oplus 1 \vdash}\ \oplus\ \textsc{Cut}_2$$

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

$$\frac{}{x : \bot \vdash \mathsf{yield\ to}\ x}\ \bot \qquad \frac{\Gamma \vdash a}{\Gamma, x : 1 \vdash \mathsf{let}\ \diamond = x; a}\ 1 \qquad \frac{}{\vdash \mathsf{halt}}\ \textsc{Halt} \qquad \frac{}{\Gamma, x : 0 \vdash \mathsf{dump}\ \Gamma\ \mathsf{in}\ x}\ 0$$

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

$$\frac{\Gamma, x : A \vdash a \qquad \Gamma, y : B \vdash b}{\Gamma, z : A \oplus B \vdash \mathsf{case}\ z\ \mathsf{of}\{\mathsf{inl}\ x \mapsto a; \mathsf{inr}\ y \mapsto b\}}\ \oplus \qquad \frac{\Gamma, x : A \vdash a}{\Gamma, z : A \& B \vdash \mathsf{let\ inl}\ x = z; a}\ \&_1$$

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

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

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

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

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

Figure 3.1: *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 Chapter 2 and Chapter 3.*

$$\frac{\dfrac{\Gamma \vdash a}{\Gamma, 1 \vdash}\ 1 \qquad \dfrac{\Gamma \vdash a}{\Gamma, 1 \vdash}\ 1}{\Gamma, 1 \oplus 1 \vdash}\ \oplus$$

In general, linearity constraints can be overcome for every term that can be represented as a collection of bits. We define data types as the smallest set of types $\mathcal{D}$, containing:

- The types 1, $\bot$, 0.
- All primitive types which can be fully represented as bits, such as integers ($\mathbb{Z}$) and double-precision floating-point ($\mathbb{R}$).
- For each $A, B \in \mathcal{D}$, $A \otimes B$, $A \oplus B$, $A \parr B$.
- For each size n and type $A \in \mathcal{D}$, $\bigotimes_n A$, $\parr_n A$.
- For each type $A[n] \in \mathcal{D}$, $\exists n : \mathbb{N}.A[n]$.

Note that weakening and contraction for types in $\mathcal{D}$ that depend on a dynamic size n is not witnessed by the calculus.

The sync rule The sync rule makes the application of weakening and contraction for these data types explicit, in a way that can be compiled efficiently later on (for example, by allocating some form of intermediate memory). It is a generalization of cut, but applicable only to data types.

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

Cut elimination

$$\boxed{\kappa\text{sync}}$$

$$\frac{\Gamma, A^{\perp} \vdash a \qquad \dfrac{\dfrac{\Delta, B^{\perp\, m} \vdash b \qquad \Xi, A^{n}, B^{mk} \vdash c}{\Delta, \Xi, A^{n} \vdash}\ \textsc{sync}^{k}_{m}}{\Gamma^{n}, \Delta, \Xi \vdash}\ \textsc{Fuse}_{n}}{} \Longrightarrow \frac{\Delta, B^{\perp\, m} \vdash b \qquad \dfrac{\dfrac{\Xi, B^{mk}, A^{n} \vdash c \qquad \Gamma, A^{\perp} \vdash a}{\Gamma^{n}, \Xi, B^{mk} \vdash}\ \textsc{Fuse}_{n}}{\Gamma^{n}, \Delta, \Xi \vdash}\ \textsc{sync}^{k}_{m}}{}$$

$$\begin{aligned}&\mathsf{fuse}\{x : A^{\perp} \mapsto a\\ &\quad y : A^{n} \mapsto \mathsf{sync}\{v : B^{\perp\, m} \mapsto b;\ w : B^{mk} \mapsto c\}\} \Longrightarrow\end{aligned}$$

$$\begin{aligned}&\mathsf{sync}\{v : B^{\perp\, m} \mapsto b\\ &\quad w : B^{mk} \mapsto \mathsf{fuse}\{y : A^{n} \mapsto c;\ x : A^{\perp} \mapsto a\}\}\end{aligned}$$

Parallel communication (BICUT) Another notion that is not possible in general in the calculus introduced so far is a cut over more than one variable. The reason to forbid cuts over two or more variables (BICUT) is that it can be used to introduced cyclic dependencies between values, leading to deadlock.

$$\frac{\Gamma, x : A^{\perp}, y : B^{\perp} \vdash a \qquad \Delta, u : A, v : B \vdash b}{\Gamma, \Delta \vdash \mathsf{bicut}\{x : A^{\perp}, y : B^{\perp} \mapsto a; u : A, v : B \mapsto b\}}\ \textsc{Bicut}$$

If BICUT is allowed, we could apply weakening not only to data values, but also to their duals:

$$\dfrac{\dfrac{}{A^{\perp}, A \vdash}\ \textsc{Ax} \qquad \dfrac{\dfrac{\dfrac{}{A^{\perp}, A \vdash}\ \textsc{Ax} \qquad \dfrac{}{\Gamma \vdash a}\ \textsc{Mix}}{\Gamma, A^{\perp}, A \vdash}\ \textsc{Mix} \qquad \dfrac{\dfrac{}{A, A^{\perp} \vdash}\ \textsc{Ax} \qquad \dfrac{\Gamma, A^{\perp}, A^{\perp}, A, A \vdash}{\Gamma, A^{\perp}, A^{\perp}, A^{2} \vdash}\ \textsc{Split}}{\Gamma, A^{\perp}, A, A^{\perp} \vdash}\ \textsc{sync}^{2}}{\Gamma, A^{\perp} \vdash}\ \textsc{Bicut}}{}$$

However, this problem cannot arise if the information moving between the two sides of the BICUT travels in a single direction; that is, if all the types of the variables in one side are data.

Extended sync rule The sync rule can also accommodate BICUT over homogeneous, n-ary products of data.

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

3.4 Example: Finite Differences

The difference operators described in the introduction are instances of 1-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 a wrap-around strategy: the template for our stencil computation is

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

where x and y are the two copies of the input array and *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 *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):

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

After fusion, the implementation of *diff* looks like this:

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

One issue with coslice is that handling each element requires branching on its index to choose 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. 4.3. After fusion, the coslice on the intermediate $\parr$-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.

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

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 $\bigotimes_2 \bigotimes_n A \multimap \bigotimes_n A$).

- 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 $\bigotimes_4 \bigotimes_n A \multimap \parr_n A$.

- 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 that *diff* is made to either access or produce elements in arbitrary orders, which hinders composition. In Chapter 4 we recover compositionality by coordinating the order of production between the producer and the consumer through a new array type.

4 Sequence arrays ($\S_n\, A$)

We have seen that the consumer of a tensor array masters the processing order, while the consumer of a $⅋$-array is a slave which must respond to any requested order. A third way exists: when the order of processing is agreed in advance by the producer and the consumer. We call such an array a sequence, and write it $\S_n\, A$. Sequences are defined to be self-dual:

$(\S_n\, A)^\perp = \S_n\, A^\perp$.

As in $⅋$-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:

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

In this incarnation of the logic we will only consider the case where the preagreed order is sequential, starting at index 0 and finishing at $n-1$. However, other schedules could be possibly supported by the language: for example, a permutation of the sequential order, a completely parallel schedule on both sides, or one enforcing locality restrictions in an otherwise global schedule.

In the process interpretation, a name behaves as $\S_n\, A$ if it can be used as a A n times. Each time it is used, the next value in the sequence will be read. This corresponds to the notion of iterators in imperative programming languages.

4.1 Cut elimination

The definition of fuse for sequences is analogous to that for $⅋$-arrays in the more simple cases.

Ready case If the type of fuse is a sequence, and both sides are instances of the traverse rule, then they can be reduced to a single instance of traverse.

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

$$\boxed{\S n}$$

$$\begin{array}{l}\mathsf{fuse}\{x : \S_{n+m}\, A^\perp \mapsto \mathsf{traverse}\{x\ \mathsf{as}\ x_1 \mapsto_{n+m} a\} \\ \quad y : \S_{n+m}\, A \mapsto \\ \quad\quad \mathsf{traverse}\{y\ \mathsf{as}\ y_1 \mapsto_n b;\ y\ \mathsf{as}\ y_2 \mapsto_m c\}\} \end{array} \quad \Longrightarrow$$

$$\begin{array}{l}\mathsf{let}\ \Gamma, \Gamma = \mathsf{split}_n\, \Gamma; \\ \mathsf{traverse}\{\mapsto_n \mathsf{fuse}\{x_1 : A^\perp \mapsto a;\ y_1 : A \mapsto b\} \\ \qquad\qquad \mapsto_m \mathsf{fuse}\{x_1 : A^\perp \mapsto a;\ y_2 : A \mapsto c\}\}\end{array}$$

Because the traverse rule implements both splitting and zipping, the transformations involved in the rewrite rule are more complex than for slice or coslice. We will present them as an inductive procedure.

Input For both the left and right traverse rules, a queue is created containing the all branches in that rule; we will call them left and right queues. We will say that a queue is ready if its first branch traverses part of the cut sequence, thus introducing a *cut sequence element*.

Output The result of cut elimination will be an instance of traverse. The branches are determined by iteratively applying the following algorithm:

- If either queue is not ready, pop the first branch from the queue and add it to the result queue.
- If both queues are ready, pop their first branches, and compare their lengths (e.g. m and n).
  - If $m = n$, introduce a cut of type A with variable names matching those of the cut sequence elements.
  - If $m < n$, split all variables (of arity n) used by the right branch into two variables of arities m and $n - m$. Create two copies of the branch, with the latter lengths, each using the corresponding variables resulting from the split. The copy with length $n - m$ is put back in the right queue, while the one with length m is appended to the result queue.
  - If $m > n$, the procedure is as in the previous step, but with the roles of the left and right queues reversed.
  - If sizes cannot be compared statically, a runtime check would have to be introduced, and two versions of the code are generated. Instead, the programmer is notified of this increase in program size, so that he can assert the relationship between the sizes explicitly.

Each step reduces the total length of the queues by at least one; the procedure terminates when both queues are empty.

Commuting of traverse If the side of fuse containing an eliminator for tensor is not ready, then we need to commute the fuse past the rule. As in the coslice rule, the fuse commutes into the branch where the fuse variable is used.

$$\boxed{\kappa \mathrm{n}\S}$$

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

$$\begin{aligned}&\mathsf{fuse}\{x : A^\perp \mapsto a\\ &\quad y : A^n \mapsto \mathsf{traverse}\{w \text{ as } u \mapsto_n b;\ w \text{ as } v \mapsto_m c\}\} \Longrightarrow\end{aligned}$$

$$\begin{aligned}&\mathsf{traverse}\{w \text{ as } u \mapsto_n \mathsf{fuse}\{y : A \mapsto b;\ x : A^\perp \mapsto a\}\\ &\qquad\qquad w \text{ as } v \mapsto_m c\}\end{aligned}$$

4.2 Loops

As explained in Chapter 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; which enables operations where the computation on each element of the array depends on the values of some or all of the previous elements. In functional programming, this corresponds to a left fold.

This is embodied in the LOOP rule, which propagates a single data item of type D across a fixed number of iterations, possibly modifying it at each step:

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

When applying the rule, the programmer fixes 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 choose 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 $zip(*)$ with a LOOP computing the sum of the result.

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

After cut-elimination, a single loop remains.

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

The rule can also be implemented with steps of type $A \multimap A$ instead of $A \multimap A \oplus 1$. For all applications discussed in the paper, the LOOP type is small, so the performance profile of both alternatives is the same.

Cut elimination No variables are eliminated; the only rules required to preserve the cut elimination property are commuting conversions.

$$\boxed{\kappa\text{loop}}$$

$$
\dfrac{\Gamma, A^\perp \vdash a \quad \dfrac{\Delta, A^\perp \vdash b \qquad \Xi, A^n, \S_m(A \otimes (A^\perp \,\&\, 1)) \vdash c}{\Delta, \Xi, A^n \vdash}\ \textsc{loop}}{\Gamma^n, \Delta, \Xi \vdash}\ \textsc{Fuse}_n \Longrightarrow
$$

$$
\dfrac{\Delta, A^\perp \vdash b \quad \dfrac{\Xi, \S_m(A \otimes (A^\perp \,\&\, 1)), A^n \vdash c \qquad \Gamma, A^\perp \vdash a}{\Gamma^n, \Xi, \S_m(A \otimes (A^\perp \,\&\, 1)) \vdash}\ \textsc{Fuse}_n}{\Gamma^n, \Delta, \Xi \vdash}\ \textsc{loop}
$$

$$
\begin{array}{l}
\mathsf{fuse}\{x : A^\perp \mapsto a \\
\quad y : A^n \mapsto \\
\quad\quad \mathsf{loop}\{w : A^\perp \mapsto b;\ u : \S_m(A \otimes (A^\perp \,\&\, 1)) \mapsto c\}\} \Longrightarrow \\
\\
\quad\quad\quad\quad \mathsf{loop}\{w : A^\perp \mapsto b \\
\quad\quad\quad\quad\quad u : \S_m(A \otimes (A^\perp \,\&\, 1)) \mapsto \\
\quad\quad\quad\quad\quad\quad \mathsf{fuse}\{y : A^n \mapsto c;\ x : A^\perp \mapsto a\}\}
\end{array}
$$

4.3 Conversions between sequences and other 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 traverse:

tensorToSequence $\equiv$a : $\bigotimes_n A$, b : $\S_n A^\perp \vdash$
slice a { a : $A^n \mapsto$ **traverse** { b **as** b' $\mapsto$ a $\leftrightarrow$ b' } }

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:

$$\frac{\Gamma, x : {D^\perp}^n \vdash a \qquad \Delta, y : D^n \vdash b}{\Gamma, \Delta \vdash \mathsf{sync}\{x : {D^\perp}^n \mapsto a; y : D^n \mapsto b\}} \text{SYNC}_n$$

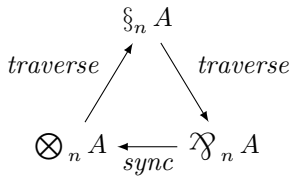
The conversion is then implemented as

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

The sync construction is in general unsound. Indeed, one program may start by demanding data from the 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 conversions from $\bigotimes_n A$ to $\S_n A$ and $\S_n A$ to $\parr_n A$ correspond to traversals, where the only cost is updating an indexing counter. On the other hand, the conversion from $\parr_n A$ to $\bigotimes_n A$ corresponds to memory allocation (explicit sharing 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.

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

Mix and halt Both MIX and HALT can be implemented in terms of traverse applied to 0 sequences.

4.4 Finite differences 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. 4.1). For this example, we use zero-padding to adjust the size of the input array. Other alternatives such a smaller sequence or wrapping around are also possible in the current language.

$$
\begin{array}{l}
\mathrm{diff} \;\equiv\; \mathrm{a} : \S_n\,\mathbb{R},\ \mathrm{b} : \S_n\,\mathbb{R}^{\perp} \vdash \\
\quad \textbf{loop}\ \{\ \mathrm{x'} \mapsto \mathrm{x'} \leftrightarrow 0\ \}\ \ \mathrm{x} : \S_n(\mathbb{R} \otimes (\mathbb{R}^{\perp} \,\&\, \bot)) \mapsto \\
\quad \textbf{traverse}\ \{\ \mathrm{a}\ \textbf{as}\ \mathrm{a}_0,\ \mathrm{b}\ \textbf{as}\ \mathrm{b}_0,\ \mathrm{x}\ \textbf{as}\ \mathrm{x}_0 \mapsto \\
\qquad \mathrm{let}\ \ \mathrm{x}_0,\ \mathrm{x}_0\text{'} = \mathrm{x}_0 \\
\qquad \textbf{sync}\ \{\ \mathrm{a'} \mapsto \mathrm{a'} \leftrightarrow \mathrm{a}_0\ \}\ \mathrm{a}_1\ \ \mathrm{a}_2 \mapsto \\
\qquad \textbf{mix}\ \{\ \mathrm{let}\ \ \mathrm{inl}\ \ \mathrm{a}_2 = \mathrm{a}_1;\ \mathrm{a}_2 \leftrightarrow \mathrm{x}_0\text{'} \\
\qquad\quad ;\ \mathrm{b}_0 \leftrightarrow \mathrm{a}_2\ -\ \mathrm{x}_0\ \}\ \}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{diff2} \;\equiv\; \mathrm{a} : \S_n\,\mathbb{R},\ \mathrm{b} : \S_n\,\mathbb{R}^{\perp} \vdash \\
\quad \textbf{cut}\ \{\ \mathrm{x'} \mapsto\ \mathrm{diff}\ \ \mathrm{a}\ \ \mathrm{x'}\ \} \\
\qquad\ \{\ \mathrm{x} \mapsto\ \ \ \mathrm{diff}\ \ \mathrm{x}\ \ \mathrm{b}\ \ \}
\end{array}
$$

Figure 4.1: *First and second order differences in* CLL^n.

After fusing, we obtain a single traversal of the input array, with constant allocation of memory (Fig. 4.2). Furthermore, source and destination arrays are guaranteed to be accessed in a cache-friendly way (Fig. 4.3).

$$
\begin{array}{l}
\mathrm{diff2} \;\equiv\; \mathrm{a} : \S_n\,\mathbb{R},\ \mathrm{c} : \S_n\,\mathbb{R}^{\perp} \vdash \\
\quad \textbf{loop}\ \{\ \mathrm{x'} \mapsto \mathrm{x'} \leftrightarrow 0\ \}\ \mathrm{x} : \S_n(\mathbb{R} \otimes (\mathbb{R}^{\perp} \,\&\, \bot)) \mapsto \\
\quad \textbf{loop}\ \{\ \mathrm{y'} \mapsto \mathrm{y'} \leftrightarrow 0\ \}\ \mathrm{y} : \S_n(\mathbb{R} \otimes (\mathbb{R}^{\perp} \,\&\, \bot)) \mapsto \\
\quad \textbf{traverse}\ \{\ \mathrm{a}\ \textbf{as}\ \mathrm{a}_0,\ \mathrm{c}\ \textbf{as}\ \mathrm{c}_0,\ \mathrm{x}\ \textbf{as}\ \mathrm{x}_0,\ \mathrm{y}\ \textbf{as}\ \mathrm{y}_0 \mapsto \\
\qquad \mathrm{let}\ \ \mathrm{x}_1,\ \mathrm{x}_1\text{'} = \mathrm{x}_0 \\
\qquad \mathrm{let}\ \ \mathrm{y}_1,\ \mathrm{y}_1\text{'} = \mathrm{y}_0 \\
\qquad \textbf{sync}\ \{\ \mathrm{a'} \mapsto \mathrm{a'} \leftrightarrow \mathrm{a}_0\ \}\qquad\quad \mathrm{a}_1\ \ \mathrm{a}_2 \mapsto \\
\qquad \textbf{sync}\ \{\ \mathrm{b'} \mapsto \mathrm{b'} \leftrightarrow \mathrm{a}_1\ -\ \mathrm{x}_1\ \}\ \mathrm{b}_1\ \ \mathrm{b}_2 \mapsto \\
\qquad \textbf{mix}\ \{\ \mathrm{let}\ \ \mathrm{inl}\ \ \mathrm{x}_2\text{'} = \mathrm{x}_1\text{'}\ ;\ \mathrm{x}_2\text{'} \leftrightarrow \mathrm{x}_0\text{'} \\
\qquad\quad ;\ \mathrm{let}\ \ \mathrm{inl}\ \ \mathrm{y}_2\text{'} = \mathrm{y}_1\text{'}\ ;\ \mathrm{y}_2\text{'} \leftrightarrow \mathrm{b}_2 \\
\qquad\quad ;\ \mathrm{c} \leftrightarrow \mathrm{b}_1\ -\ \mathrm{y}_1\ \}\ \}
\end{array}
$$

Figure 4.2: *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 4.3: *Compiled code for fused, second-order differences. Redundant, single static assignments of primitive values to local variables have been removed for readability; they are guaranteed to be eliminated by the C compiler as part of the register allocation phase.*

5 A computational interpretation

The usual challenge when giving a semantics to linear logic is modelling its apparently classical behaviour, which resists a constructive treatment. However, we can recover an intuitionistic view by applying double negation to all types. Any further levels of negation are not an issue, because, both in classical and intuitionistic logics, it holds that $\neg A$ is isomorphic to $\neg\neg\neg A$.

Hence, we can give a model for the logic by embedding it into the lambda calculus. To make the syntax more understandable, we will use a variant of System F which includes algebraic data types (products and sums) as the type system. This has the added advantage of bringing the description of the cost model closer to a real target language.

A well-typed derivation in CLL^n will translate to a well-typed term in System F. Note that the embedding is lossy; in particular, linearity constraints will translate into meta-theoretical conditions.

Scope Thanks to the cut elimination property (Sec. 2.8), it is possible to eliminate all uses of *parametric polymorphism* ($\forall\alpha.A[\alpha]$ and $\exists\alpha.A[\alpha]$) before interpreting a derivation. Therefore, we will limit ourselves to giving a meaning to quantification over sizes ($\forall n : \mathbb{N}.A[n]$ and $\exists n : \mathbb{N}.A[n]$).

5.1 Interpretation of types

As a first step, we will define an interpretation of the Chapter 2 and Chapter 3. This includes the whole language, except for sequential arrays. Afterwards, we will be able to extend this interpretation to handle sequences by a polarization step, in some cases. Together, this will define a *restricted interpretation* that will be used in the compilation of the examples in Chapter 7.

We will assume the existence of a type of effects $\perp\!\!\!\perp$. The interpretation of a term in the language will always have type $\perp\!\!\!\perp$, representing the effects or results of the computation.

- $\perp\!\!\!\perp$ is a monoid with unit nop and binary operation $\gg$. The structure is required to interpret mix and halt. This monoid structure also induces a function $concat : (\mathbb{N}d \to \perp\!\!\!\perp) \to \perp\!\!\!\perp$, which is used to implement $\mathsf{traverse}$.
- Assume d is the interpretation of a type which is a data type. Then, there exists:
  - $\mathcal{K}_d$, an infinite type of keys.
  - $allocate : (\mathcal{K}_d \to \perp\!\!\!\perp) \to \perp\!\!\!\perp$, which represents the allocation of (uninitialized) memory to store a value of type d.
  - $write : (\mathcal{K}_d \to d) \to \perp\!\!\!\perp$, which represents writing a value of type d to a previously allocated memory location.
  - $read : (\mathcal{K}_d \to d \to \perp\!\!\!\perp) \to \perp\!\!\!\perp$, which represents reading a value of type d to a previously written memory location.

  Note that it is always possible to extend any existing monoid into an effect type with the above interface by using a state monoid transformer as provided by Thielemann [2009]. The state variable will be given type $\mathbb{N} \times (\mathbb{N} \to \bigcup \mathcal{D})$, where each of the keys $\mathcal{K}_d$ are distinct natural numbers, and $\bigcup \mathcal{D}$ is a supertype of all data types.

  Initially, the first component is 0, and the second component has some arbitrary value for all inputs; then:

  - *allocate* returns the first component and then increments it in place.
  - *read* indexes the second component with the key.
  - *write* redefines the value of the second component at the key to the given value.

The first challenge that arises when translating CLL is the involutive negation operator, which lacks a straightforward translation into an intuitionistic calculus such as System F.

We will employ a double negation translation for our embedding.

Definition 1. *We define the translation of a type $A^\bullet$ in the commutative fragment of* CLL^n *as follows:*

$$\begin{aligned}
\mathbb{A}^\bullet &= (\mathbb{A} \to \bot) \to \bot & \mathbb{A}^{\bot\bullet} &= \mathbb{A} \to \bot \\
0^\bullet &= (\varnothing \to \bot) \to \bot & \top^\bullet &= \varnothing \to \bot \\
1^\bullet &= \bot \to \bot & \bot\!\!\!\bot^\bullet &= \bot \\
(A \oplus B)^\bullet &= (A^\bullet + B^\bullet \to \bot) \to \bot & (A \,\&\, B)^\bullet &= A^{\bot\bullet} + B^{\bot\bullet} \to \bot \\
(A \otimes B)^\bullet &= (A^\bullet \times B^\bullet \to \bot) \to \bot & (A \parr B)^\bullet &= A^{\bot\bullet} \times B^{\bot\bullet} \to \bot
\end{aligned}$$

$$\begin{aligned}
(\forall n : \mathbb{N}.A[n])^\bullet &= \mathbb{N} \times A[n]^\bullet \to \bot & (\exists n : \mathbb{N}.A[n])^\bullet &= (\mathbb{N} \times A[n]^\bullet \to \bot) \to \bot \\
(\bigotimes_n A)^\bullet &= (\mathbb{N} \times (\mathbb{N} \to A^\bullet) \to \bot) \to \bot & (\parr_n A)^\bullet &= \mathbb{N} \times (\mathbb{N} \to A^{\bot\bullet}) \to \bot
\end{aligned}$$

5.2 Heuristic interpretation of sequence types

The *sequence* operator is self-dual in the logic. Semantically, this means that neither the producer nor the consumer of a sequence is in control of the program flow. When compiling the program, both processes will need to be run concurrently, either by means of multiple threads, or coroutines. The efficiency of both approaches would be unsatisfactory in a high-performance setting.

Sequences are touted as a type of symmetric array, where none of the sides is in control of the order of computation. Full adherence to this premise would require that the producer and consumer of a sequence are run in concurrent processes. However, our goal is to produce efficient, sequential code. This is achieved by having one single thread of control; or, in other words, by every value having a well-determined producer and consumer.

A first approach, which requires minimal changes to the language, is to statically assign a polarity to each of the sequences in the program, so that each $\S_n A$ becomes either $\S^+{}_n A$ or $\S^-{}_n A$. In both cases, elements are produced sequentially. The difference is that, in positive sequences, the consumer controls the point in the execution of the program at which the elements are produced; whereas for negative sequences, the consumer must consume the elements at any instant that they are produced (although the order is still predetermined).

We can understand non-polarised sequences as introducing a form of polarity polymorphism, realized by agreeing on the order of evaluation. The agreed-upon order is strictly sequential, which gives a sensible meaning to the LOOP rule. Polarization resolves the polymorphism by choosing one specific assignment of polarities.

Definition 2. *Positive sequences have the same computational interpretation as tensor arrays and negative sequences the same as $\parr$-arrays:*

$$(\S^+_n A)^\bullet = ((\mathbb{N} \to A^\bullet) \to \bot) \to \bot \qquad (\S^-_n A)^\bullet = (\mathbb{N} \to A^{\bot\bullet}) \to \bot$$

Definition 3. *The* sequence arity of a type *is defined inductively as follows:*

$$\begin{aligned}
|A \oplus B| &= max(|A|, |B|) & |A \,\&\, B| &= max(|A|, |B|) \\
|0| &= |0| & |\top| &= |\top| \\
|A \otimes B| &= |A| + |B| & |A \parr B| &= max(|A|, |B|) \\
|1| &= 0 & |\bot\!\!\!\bot| &= 0 \\
|\bigotimes_n A| &= \infty \cdot |A| & |\parr_n A| &= |A| \\
|\S^+_n A| &= |A| & |\S^-_n A| &= max(1, |A|) \\
|\mathbb{A}| &= 0 & |\mathbb{A}^\bot| &= 0
\end{aligned}$$

Definition 4. *The* sequence arity of a context *is defined from the sequence arity of a type, by considering* $A^n \cong \bigotimes_n A$*, and* $A, B \cong A \otimes B$*.*

When a traverse rule is applied, if the context contains at most one negative sequence (that is, only one sequence needs to have control), then the computation can occur in a single thread, and only one loop counter is used by the producer and the consumer. If this condition holds for every instance of the traverse rule, then the program is maximally efficient. We will call such an assignment of polarities a *single-threaded* assignment.

Definition 5. *A polarization is single-threaded if:*

- *Whenever one or more sequences are eliminated (either with the* AX *or* traverse*), there is* at most one *negative sequence in the context (that is, the context has sequence arity 1).*
- *Sequences introduced with* cut *have opposite signs at each end.*
- *Sequences introduced by* LOOP *and* SYNC *have positive signs.*

The last condition is trivial because polarization can be changed to fulfil it without rendering the previous two false.

For many practical programs, it is possible to give such a polarization statically *without altering the derivation* in any other way:

Theorem 4. *If no* $\oplus$ *rule (or any other form of branching rule) is used in between the introduction of a sequence with* cut*, and the elimination of all the enclosing connectives, then there exists a single-threaded polarization.*

Proof. Assign positive polarities to all the sequences in the root of the derivation. Thus, the sequence arity of the context at the root is 0.

We will prove that, if the sequence arity of the context at the root is ≤ 1. there exists a polarization in which the sequence arities of all contexts are ≤ 1; by proceeding inductively we may proceed inductively on the structure of the derivation.

The only place where new sequences are introduced is cut. In this case, we call the side that receives the negative sequence as the negative side, and the other as the positive side.

Note that, if the arity of the cut is more than n, the negative sequence in the context must necessarily be used in the non-demoted side, because sequences have arity one as per the induction hypothesis.

Because of the hypotheses, there exists in both sides a simple path of elimination rules deconstructing the newly-introduced variable. Polarities of the sequences in the new variables are assigned so that, on both sides, there will be no more than one negative sequence in the context at any point. □

We can derive a simple corollary, that can be checked quickly by looking at the root, and each of the cuts:

Corollary 1. *Assume a derivation in which no sequences introduced by* cut *are nested inside other connectives.*

For every assignment of polarities to the sequences in the root of the derivation such that there is only one negative sequence, there exists a single-threaded polarization.

All of the conditions in Thm. 4 are trivial if sequences are not nested, and have arity one.

Corollary 2. *Assume a derivation in there are no* cut*s involving sequences.*

For every assignment of polarities to the sequences in the root of the derivation such that there is only one negative sequence, there exists a single-threaded polarization. The condition in Cor. 1 is trivial if no cut *involves a sequence.*

The hypotheses in Thm. 4 are sufficient, but not necessary. Giving a more precise characterization is difficult; we present instead a general polarization mechanism that applies to any derivation.

5.3 Generalized interpretation of sequence types

We first remark that finding a single-threaded polarization (Def. 5) for the general case is not possible. A counter-example follows (Fig. 5.1):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1, A^\perp, B^\perp \vdash a}{{\Gamma_1}^n, \S_n A^\perp, \S_n B^\perp \vdash}\,\S \quad \dfrac{\Delta_1, C^\perp \vdash b}{{\Delta_1}^n, \S_n C^\perp \vdash}\,\S}{\dfrac{{\Gamma_1}^n, {\Delta_1}^n, \S_n A^\perp, \S_n B^\perp \parr \S_n C^\perp \vdash}{{\Gamma_1}^n, {\Delta_1}^n, \S_n A^\perp, \S_n B^\perp \parr \S_n C^\perp, 1 \vdash}\,1}\,\parr \quad \dfrac{\dfrac{\Gamma_1, B^\perp \vdash c}{{\Gamma_1}^n, \S_n B^\perp \vdash}\,\S \quad \dfrac{\Delta_1, A^\perp, C^\perp \vdash d}{{\Delta_1}^n, \S_n A^\perp, \S_n C^\perp \vdash}\,\S}{{\Gamma_1}^n, {\Delta_1}^n, \S_n A^\perp, \S_n B^\perp \parr \S_n C^\perp \vdash}\,\parr}{{\Gamma_1}^n, {\Delta_1}^n, \S_n A^\perp, \S_n B^\perp \parr \S_n C^\perp, 1 \vdash}\,1}\,\oplus \quad \dfrac{\dfrac{\Xi_1, B, C \vdash e}{{\Xi_1}^n, \S_n B, \S_n C \vdash}\,\S}{{\Xi_1}^n, \S_n B \otimes \S_n C \vdash}\,\otimes}{1 \oplus 1, {\Gamma_1}^n, {\Delta_1}^n, \S_n A^\perp, \S_n B^\perp \parr \S_n C^\perp \vdash}\,\textsc{Cut}$$
$$1 \oplus 1, {\Gamma_1}^n, {\Delta_1}^n, {\Xi_1}^n, \S_n A^\perp \vdash$$

(a) *Left side*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_2, A, B^\perp \vdash f}{{\Gamma_2}^n, \S_n A, \S_n B^\perp \vdash}\,\S \quad \dfrac{\Delta_2, C^\perp \vdash g}{{\Delta_2}^n, \S_n C^\perp \vdash}\,\S}{\dfrac{{\Gamma_2}^n, {\Delta_2}^n, \S_n A, \S_n B^\perp \parr \S_n C^\perp \vdash}{{\Gamma_2}^n, {\Delta_2}^n, \S_n A, \S_n B^\perp \parr \S_n C^\perp, 1 \vdash}\,1}\,\parr \quad \dfrac{\dfrac{\Gamma_2, B^\perp \vdash h}{{\Gamma_2}^n, \S_n B^\perp \vdash}\,\S \quad \dfrac{\Delta_2, A, C^\perp \vdash i}{{\Delta_2}^n, \S_n A, \S_n C^\perp \vdash}\,\S}{{\Gamma_2}^n, {\Delta_2}^n, \S_n A, \S_n B^\perp \parr \S_n C^\perp \vdash}\,\parr}{{\Gamma_2}^n, {\Delta_2}^n, \S_n A, \S_n B^\perp \parr \S_n C^\perp, 1 \vdash}\,1}\,\oplus \quad \dfrac{\dfrac{\Xi_2, B, C \vdash j}{{\Xi_2}^n, \S_n B, \S_n C \vdash}\,\S}{{\Xi_2}^n, \S_n B \otimes \S_n C \vdash}\,\otimes}{1 \oplus 1, {\Gamma_2}^n, {\Delta_2}^n, \S_n A, \S_n B^\perp \parr \S_n C^\perp \vdash}\,\textsc{Cut}$$
$$1 \oplus 1, {\Gamma_2}^n, {\Delta_2}^n, {\Xi_2}^n, \S_n A \vdash$$

(b) *Right side*

$$\dfrac{1 \oplus 1, {\Gamma_1}^n, {\Delta_1}^n, {\Xi_1}^n, \S_n A^\perp \vdash a \qquad 1 \oplus 1, {\Gamma_2}^n, {\Delta_2}^n, {\Xi_2}^n, \S_n A \vdash b}{1 \oplus 1, 1 \oplus 1, {\Gamma_1}^n, {\Gamma_2}^n, {\Delta_1}^n, {\Delta_2}^n, {\Xi_1}^n, {\Xi_2}^n \vdash}\,\textsc{Cut}$$

(c) *Full program*

Figure 5.1: *Non-polarizable derivation*

The impossibility can be easily verified by enumerating all possible polarizations. In this example, the core problem is that we do not know in advance the result of the branch on $1 \oplus 1$. In general, in the presence of branching, even if there exist polarizations for each of the alternatives, they may each place non-local requirements that cannot be reconciled.

Solving this problem requires a way of communicating which of the branches received a negative sequence on the other side of cut. This communication takes place each time that a negative multiplicative connective is eliminated, that is, a par product $A \parr B$, a $\parr$-array $\bigparr_n A$, or a negative sequence ${\S^-}_n A$.

Definition 6. *We define the positive A^+ and negative A^- translations as follows. The translation operators commute with all the other connectives: for example, $(A \otimes B)^+ = A^+ \otimes B^+$.*

$$(A \otimes B)^+ = A^+ \otimes B^+ \qquad (A \parr B)^- = A^- \parr B^-$$
$$(A \parr B)^+ = (A^- \parr B^+) \& (A^+ \parr B^-) \qquad (A \otimes B)^- = A^+ \otimes B^- \oplus A^- \otimes B^+$$
$$(\bigotimes_n A)^+ = \bigotimes_n A^+ \qquad (\bigparr_n A)^- = \bigparr_n A^-$$
$$(\bigparr_n A)^+ = \forall i : \mathbb{N}. \forall p : i < n. \bigparr_i A^- \parr A^+ \parr \bigparr_{n-i-1} A^- \qquad (\bigotimes_n A)^- = \exists i : \mathbb{N}. \exists p : i < n. \bigotimes_i A^+ \otimes A^- \otimes \bigotimes_{n-i-1} A^+$$
$$(\S_n A)^+ = \bigotimes_n A^+ \qquad (\S_n A)^- = \bigparr_n A^-$$

After the transformation, no sequences remain in the calculus; they are translated either as positive tensor arrays, or negative $\parr$-arrays. This results in a slightly different set of rules: in particular, the LOOP rule now introduces $\bigotimes_m (D \otimes (D^\perp \& 1))$, instead of ${\S^+}_m (D \otimes (D^\perp \& 1))$.

Theorem 5. *For every* CLL^n *derivation, there exists an equivalent one which does not contain sequences.*

Proof. To construct the equivalent derivation, it is necessary to keep track of which names where translated positively, and which negatively.

Observe that:

- When a negatively-translated name is eliminated, only one negatively translated name is introduced into each new branch; even if the translation introduces a sum type (either $\exists$ or $\oplus$).
- When a positively-translated name is eliminated, and the translation introduces a choice type (either $\forall$ or $\&$), it is possible to perform the choice in such way that only one negatively translated name is introduced into each new branch. If there is no choice, all the introduced types will be the result of a positive translation.

This means that, if the context in the root contains only one negatively translated name Because negative names introduce at most one negative name in each branch, and positive names leave at least one branch without a negative binding, it is possible to preserve the invariant of having at most one negative name in the context at all times.

In particular, because there will be at most one $\parr$-array in the context when applying traverse, the traverse may always be rewritten in the form of coslice. □

5.4 Interpreting derivations into the Lambda Calculus

Once we have a well-typed, single-threaded polarization, we can interpret CLL^n terms $\gamma : \Gamma \vdash t$ into terms in System F with free variables γ of type $\Gamma\bullet$.

For readability, we leave some details in the code below implicit:

- Shifting control, that is, natural conversions from values of type α to values of type $(\alpha \to \bot\!\!\!\bot) \to \bot\!\!\!\bot$.
- Indexing and generalising into n-ary variables, that is, converting between $\mathbb{N} \times (\mathbb{N} \to \alpha)$ and α using the index variables available in the context. This includes demoting contexts.

Translation of sequences The TRAVERSE rule performs both slicing, splitting and scheduling; this makes its implementation less straightforward than the other rules.

First, apply the semantics of slice to each positive sequence, giving each of them control. We obtain bindings of the form $\mathbb{N} \times (\mathbb{N} \to A^\bullet)$ for each sequence ${\S^+}_n\, A$.

Then, the translation of the traverse rule depends on the polarities of the sequences in the context:

- All sequences are positive (first case): Consider the combined size m of all branches, and define a function that, for each index from 0 to $m-1$, performs the corresponding step.

  Sequence all the steps together with concat.

- One negative sequence (second case): The negative sequence translates identically to a $\parr$-array. Apply the semantics of coslice to it, interpreting the corresponding subterm at each step.

  If a branch of the traverse rule does not make use of the negative sequence, this branch is scheduled as in the previous case (zero negative sequences) either before or after the last element of the branch to the right, or before the first element of the branch to the left.

$$
\begin{aligned}
x \leftrightarrow y^\bullet &= x\,y \\
\mathsf{cut}\{x : A^n \mapsto a; y : A^\perp \mapsto b\}^\bullet &= (\lambda x.a^\bullet)\,(n, \lambda i.\lambda y.b^\bullet) \\
\mathsf{mix}\{a; b\}^\bullet &= a^\bullet \gg b^\bullet \\
\mathsf{yield\ to}\ x^\bullet &= x \\
\mathsf{let}\ \diamond = x; a^\bullet &= x\,a^\bullet \\
\mathsf{halt}^\bullet &= \mathsf{nop} \\
\mathsf{dump}\ \Gamma\ \mathsf{in}\ x^\bullet &= x\,(\lambda x.x) \\
\mathsf{let}\ x, y = z; a^\bullet &= z\,(\lambda(x, y).a^\bullet) \\
\mathsf{connect}\ z\ \mathsf{to}\{x \mapsto a; y \mapsto b\}^\bullet &= z\,(\lambda x.a^\bullet, \lambda y.b^\bullet) \\
\mathsf{case}\ z\ \mathsf{of}\{\mathsf{inl}\ x \mapsto a; \mathsf{inr}\ y \mapsto b\}^\bullet &= z\,(\lambda w.\mathsf{case}\,w\,\mathsf{of}\,\mathsf{inl}\,x \to\ a^\bullet \mid \mathsf{inr}\,y \to\ b^\bullet) \\
\mathsf{let}\ \mathsf{inl}\ x = z; a^\bullet &= z\,\mathsf{inl}\,(\lambda x.a^\bullet) \\
\mathsf{let}\ \mathsf{inr}\ x = z; a^\bullet &= z\,\mathsf{inr}\,(\lambda x.a^\bullet) \\
\mathsf{let}\ x, y = \mathsf{split}_n\ z; a^\bullet &= (\lambda(n, z).(\lambda x.(\lambda y.a^\bullet)\,(m, z))\,(n, z))\,z \\
\mathsf{let}\ x = \mathsf{slice}\ z; a^\bullet &= z\,(\lambda x.a^\bullet) \\
\mathsf{coslice}\ z\{x \mapsto_n a; y \mapsto_m b\}^\bullet &= z\,(n + m, \lambda i.\mathsf{if}\,0 \le i \wedge i < n\,\mathsf{then}\,\lambda x.a^\bullet\ \mathsf{else}\,\lambda y.b^\bullet\ \mathsf{end}) \\
\mathsf{traverse}\{x : \underset{n}{\S^+} A\ \mathsf{as}\ x', y : \underset{n}{\S^+} B\ \mathsf{as}\ y' \mapsto_n a\}^\bullet &= \mathit{schedule}\,(\lambda i.(\lambda x'.(\lambda y'.a^\bullet)\,y\,i)\,x\,i) \\
\mathsf{traverse}\{x : \underset{n}{\S^-} A\ \mathsf{as}\ x', y : \underset{n}{\S^+} B\ \mathsf{as}\ y' \mapsto_n a\}^\bullet &= x\,(\lambda i.\lambda x'.y\,(\lambda y'.a^\bullet)\,y\,i) \\
\mathsf{sync}\{x : D^{\perp n} \mapsto a; y : D^n \mapsto b\}^\bullet &= \mathit{allocate}\,(\lambda d_0.\ldots\ \mathit{allocate}\,(\lambda d_{n-1}.(\lambda x.a^\bullet)\,(n, \lambda i.\mathit{write}\,d_i) \gg \\
&\quad (\lambda y.b^\bullet)\,(n, \lambda i_0.\mathit{read}\,d_i))) \\
\mathsf{loop}\{x : D^\perp \mapsto a; y : \underset{m}{\S}\,(D \otimes (D^\perp \,\&\, 1)) \mapsto b\}^\bullet &= \mathit{allocate}\,(\lambda d.(\lambda x.a^\bullet)\,\mathit{write}\,d \gg \\
&\quad (\lambda y.b^\bullet)\,(m, \lambda i.\lambda u.\mathit{read}\,d\,(\lambda r.u\,(r, \\
&\quad \lambda u_0.\mathsf{case}\,u_0\,\mathsf{of}\,\mathsf{inl}\,w \to\ \mathit{write}\,d \mid \mathsf{inr}\,o \to\ o))))) \\
\mathsf{let}\ x = z@B; a^\bullet &= \iota\,z\,(B, (\lambda x.a^\bullet) \circ \mathsf{shift}) \\
\mathsf{let}\ x\langle\beta\rangle = z; a^\bullet &= \iota\,z\,(\lambda(\beta, x).a^\bullet)
\end{aligned}
$$

Figure 5.2: *Embedding of* CLL^n *into the lambda calculus.*

5.5 Guaranteed improvement

We can measure the cost in terms of the number of reductions for each rule.

This is made easier by the fact that:

- Each variable is used exactly once.
- Each variable will be fully applied.

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. First, we define a measure on types:

Definition 7. *We define the size* $|A|$ *of a type* A *as follows, where* $|A| = |A^\perp|$*:*

$$
\begin{aligned}
|A \oplus B| &= 1 + max(|A|, |B|) & |0| &= 0 \\
|A \otimes B| &= |A| + |B| & |1| &= 0 \\
|\bigotimes_n A| &= n \cdot |A| & |\underset{n}{\S} A| &= n \cdot |A|
\end{aligned}
$$

Definition 8. *The cost $|a|$ of a program a is defined in Fig. 5.3. It corresponds closely to the number of β-reductions required to evaluate each term in the System F embedding (Fig. 5.2).*

$$
\begin{aligned}
|x \leftrightarrow y| &= |A| \\
|\mathsf{cut}\{x : A^n \mapsto a; y : A^\perp \mapsto b\}| &= 1 + |a| + n \cdot |b| \\
|\mathsf{mix}\{a; b\}| &= 1 + |a| + |b| \\
|\mathsf{yield\ to}\ x| &= x \\
|\mathsf{let}\ \diamond = x; a| &= 1 + |a| \\
|\mathsf{halt}| &= 1 \\
|\mathsf{dump}\ \Gamma\ \mathsf{in}\ x| &= 1 \\
|\mathsf{let}\ x, y = z; a| &= 1 + |a| \\
|\mathsf{connect}\ z\ \mathsf{to}\{x \mapsto a; y \mapsto b\}| &= 1 + |a| + |b| \\
|\mathsf{case}\ z\ \mathsf{of}\{\mathsf{inl}\ x \mapsto a; \mathsf{inr}\ y \mapsto b\}| &= 1 + max(|a|, |b|) \\
|\mathsf{let\ inl}\ x = z; a| &= 1 + |a| \\
|\mathsf{let\ inr}\ x = z; a| &= 1 + |a| \\
|\mathsf{let}\ x, y = \mathsf{split}_n\ z; a| &= 1 + |a| \\
|\mathsf{let}\ x = \mathsf{slice}\ z; a| &= 1 + |a| \\
|\mathsf{coslice}\ z\{x \mapsto_n a; y \mapsto_m b\}| &= 1 + n \cdot |a| + m \cdot |b| \\
|\mathsf{traverse}\{x : \S^+_n A\ \mathsf{as}\ x', y : \S^+_n B\ \mathsf{as}\ y' \mapsto_n a\}| &= 1 + 2 + n \cdot |a| \\
|\mathsf{traverse}\{x : \S^-_n A\ \mathsf{as}\ x', y : \S^+_n B\ \mathsf{as}\ y' \mapsto_n a\}| &= 1 + 2 + n \cdot |a| \\
|\mathsf{sync}\{x : D^{\perp n} \mapsto a; y : D^n \mapsto b\}| &= 1 + |a| + |b| + n \cdot |D| \\
|\mathsf{loop}\{x : D^\perp \mapsto a; y : \S_m (D \otimes (D^\perp \,\&\, 1)) \mapsto b\}| &= 1 + |a| + |b| + |D| \\
|\mathsf{let}\ x = z@B; a| &= 1 + |a| \\
|\mathsf{let}\ x\langle\beta\rangle = z; a| &= 1 + |a|
\end{aligned}
$$

Figure 5.3: *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* AXIOM *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.*

One can then state and prove the property of interest:

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

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

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

In the case of sequences, each application of the traverse rule may introduce an new index variable if all the sequences in the context are positively polarized. A cut on a sequence type forces the polarities of each of the sequences introduced to be distinct. For example, positive on the left, and negative on the right.

After the cut elimination step, there will be a negative sequence in the context of the resulting subderivation iff there is a negative sequence in the side of the cut where a positive sequence was introduced. Because cut elimination does not introduce more traversals in which all sequences are positive, the overhead due to these indices is preserved or reduced by the cut.

6 Compilation

Programs for our language are terms in CLL^n. A term with a valid derivation in CLL^n can be translated into well-typed term in System F (see Chapter 5). By implementing this translation scheme in a functional language, we would obtain an interpreter for CLL^n.

However, if our goal is to use CLL^n in a high-performance setting, a direct implementation of a functional semantics is not practical. In general, functional languages require advanced compiler optimizations and heuristics to be performant. Given that our goal is to give the programmer control over the performance of the program, a naïve translation into a functional language would just bring us back to square one.

This is solved by targeting a low-level language instead, where we can control the costs of each operation that we implement. By taking advantage of the linearity of the code, we can allocate objects with less overhead than a regular functional runtime would have.

The overhead of calling a closure is significant, regardless of its presentation. Closures are generated each time that the flow of control switches from a consumer to a producer; for example, each time that a positive value $\mathbb{A}$ is obtained from a $\parr$-array $\bigparr_n \mathbb{A}$.

The key insight is that we can arrange the rule applications in such a way that the number of times that control needs to change is minimized. This process is called *focusing* [Miller and Saurin, 2007].

The input to the compiler is an AST (for example, generated by a Haskell DSL). The nodes of the AST are parameterized over two types, one for references (`r`) and one for names (`n`); the value of a reference must always match a name introduced by a parent node.

```
data LL r n where
  Ax       ::  r → r → LL r n
  Cut      ::  Type r n → n → LL r n → n → LL r n → LL r n
  Split    ::  r → Size r → n → Size r → n → LL r n → LL r n
  Slice    ::  r → n → LL r n → LL r n
  CoSlice  ::  r → [(Size r,n,LL r n)] → LL r n
  Tensor   ::  r → n → n → LL r n → LL r n
  Par      ::  r → n → LL r n → n → LL r n → LL r n
  Plus     ::  r → n → LL r n → n → LL r n → LL r n
  With     ::  Bool → r → n → LL r n → LL r n
  One      ::  r → LL r n → LL r n
  Zero     ::  r → LL r n
  Bot      ::  r → LL r n
  Exist    ::  r → String → n → LL r n → LL r n
  Forall   ::  r → Size r → n → LL r n → LL r n
  Trav     ::  [r] → [(Size r, ([Maybe n], LL r n))] → LL r n
  What     ::  String → [r] → LL r n
 deriving (Show)

pattern Halt     = Trav [ ] [ ]
pattern Mix a b = Trav [ ] [(ISz 1, ([ ], a)), (ISz 1, ([ ], b))]
```

Sizes are given as polynomials over the integers; indices and sizes share the same representation.

```
data Size r where
  VSz   ::  r → Size r
  ISz   ::  Integer → Size r
  (:+)  ::  Size r → Size r → Size r
  (:−)  ::  Size r → Size r → Size r
  (:×)  ::  Size r → Size r → Size r
```

```
type Index = Size

instance Num (Size r) where
  (+) = (:+)
  (−) = (:−)
  (*) = (:×)
  fromInteger = ISz
```

All types can be built from the positive connectives, and a negation operator:

```
data Type r n where
  (:⊥)          ::  Type r n → Type r n
  (:⊗), (:⊕)    ::  Type r n → Type r n → Type r n
  I, O          ::  Type r n
  Ex            ::  (Size r → Type r n) → Type r n
  Var           ::  r → Type r n
  Perp          ::  Type r n → Type r n
  BigTensor     ::  Size r → Type r n → Type r n
```

With this notation, duality of types is clearly involutive:

```
dual :: Type r n → Type r n
dual (Perp x) = x
dual x = Perp x
```

Negative connectives are defined in terms of positive connectives by means of pattern synonyms [Augustsson et al., 2011]:

```
pattern BigPar sz x ← Perp (BigTensor sz (dual → x))
pattern x :⅋ y ← Perp ((dual → x) :⊗ (dual → y))
pattern x :& y ← Perp ((dual → x) :⊕ (dual → y))
pattern T = Perp O
pattern B = Perp I
pattern All t ← Perp (Ex ((dual .) → t))
```

Phases The transformation process is divided into the phases in Fig. 6.1:

Phases 1. to 4. operate on a CLL^n derivation, optimizing it and enriching it with additional information. Phase 5. works with a generic effect type $\perp\!\!\!\perp$, which Phase 6 specializes to generate C code.

Linearity check The input to this phase is a general CLL^n term, together with a list of free names that the term is supposed to consume.

The first requirement for correctness in CLL^n is that values are used the right number of times, and in the right branch of the rules. The rules in CLL^n can be read in two ways: either defining the possible contexts in each of the branches depending on the context at the root (top down); or the context at the root depending on the contexts of the branches (bottom up).

In general, given a context at the root of a rule, the choice of how it should be split between the branches is non-local. Therefore, we choose a bottom up approach from this phase: we start at the leaves, which are rules that explicitly eliminate every variable in the context; and build up to the root of the derivation. Each rule in the derivation imposes constraints on the contexts of the branches: for example, any names introduced by the rule must appear exactly once on the context of the corresponding subtree.

Observe that, in CLL^n, linearity depends not only on the variables in a given context, but also on their arities. However, arities of variables are determined by the types; because of this, this aspect of linearity is deferred to the type-checking stage.

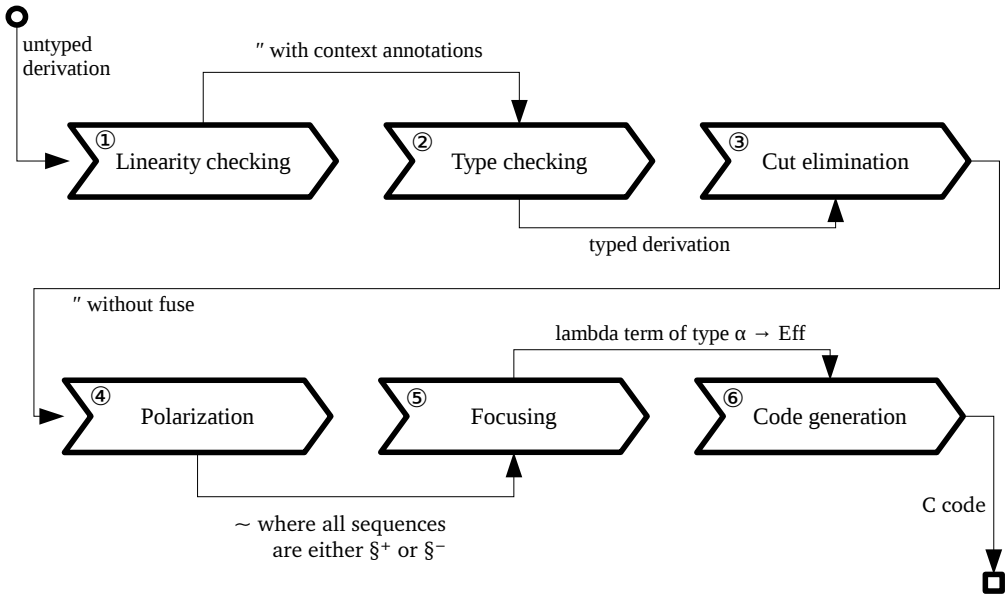


Figure 6.1: *Code generation pipeline*

We proceed to the next phase iff the list of names inferred for the whole derivation corresponds to the one given as input.

The result of this step is a derivation where each subtree is annotated with the names in the context at its root.

Type check The input to this phase is a derivation tree, together with the types of the variables consumed at the root of the node.

While linearity check is bottom-up, type checking is top-down: we start at the root of the derivation, and infer the types of the variables in the branches based on the types of the variables in the local context, and the information contained in the rule. This proceeds recursively.

The output of this phase is a derivation where each reference and each name have been annotated with a type.

Cut elimination The input to this phase is a general, well-typed derivation.

Each `cut` that the user has marked to eliminate is rewritten according to the rules in Chapter 2 and Chapter 3.

After cut elimination is performed, both the linearity check and the type check are performed again. This refreshes annotations required by later stages of the compilation.

At the end of this phase, a derivation with no instances of `fuse` is produced. More importantly, any `cut` involving polymorphic types will have been eliminated, thus removing that concern from later phases. In all other aspects, the output of this phase has the same presentation as the input: in particular, if no cuts are marked to eliminate, the program would be left unchanged.

The phase succeeds as long as all the inequalities between sizes that arise can be decided with the available information on the size variables. Otherwise, the user is warned, and required to provide this information (either by introducing a manual branch, or by adding the inequation to the context).

Dynamic size tests Type checking and cut elimination rules depend on the relative magnitude of sizes. To disambiguate them, we would need to introduce dynamic size tests.

For example, consider the following example. In the absence of any external information, two outcomes are possible:

$$\begin{aligned}
&\mathsf{fuse}\{x : \S_{n+m}\, A^{\perp} \mapsto \\
&\quad \mathsf{traverse}\{x \,\mathsf{as}\, x_1 \mapsto_m a;\ x \,\mathsf{as}\, x_2 \mapsto_n b\} \\
&\quad y : \S_{n+m}\, A \mapsto \\
&\quad \mathsf{traverse}\{y \,\mathsf{as}\, y_1 \mapsto_n c;\ y \,\mathsf{as}\, y_2 \mapsto_m d\}\} \Longrightarrow \\
&\qquad\qquad \mathsf{let}\ \gamma 1, \gamma 1 = \mathsf{split}_n\, \gamma 1;\ \mathsf{let}\ \delta 2, \delta 2 = \mathsf{split}_{-n+m}\, \delta 2; \\
&\qquad\qquad \mathsf{traverse}\{\mapsto_n \mathsf{fuse}\{x_1 : A^{\perp} \mapsto a;\ y_1 : A \mapsto c\} \\
&\qquad\qquad\qquad\qquad \mapsto_{-n+m} \mathsf{fuse}\{x_1 : A^{\perp} \mapsto a;\ y_2 : A \mapsto d\} \\
&\qquad\qquad\qquad\qquad \mapsto_n \mathsf{fuse}\{x_2 : A^{\perp} \mapsto b;\ y_2 : A \mapsto d\}\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{fuse}\{x : \S_{n+m}\, A^{\perp} \mapsto \\
&\quad \mathsf{traverse}\{x \,\mathsf{as}\, x_1 \mapsto_m a;\ x \,\mathsf{as}\, x_2 \mapsto_n b\} \\
&\quad y : \S_{n+m}\, A \mapsto \\
&\quad \mathsf{traverse}\{y \,\mathsf{as}\, y_1 \mapsto_n c;\ y \,\mathsf{as}\, y_2 \mapsto_m d\}\} \Longrightarrow \\
&\qquad\qquad \mathsf{let}\ \delta 1, \delta 1 = \mathsf{split}_m\, \delta 1;\ \mathsf{let}\ \gamma 2, \gamma 2 = \mathsf{split}_{n-m}\, \gamma 2; \\
&\qquad\qquad \mathsf{traverse}\{\mapsto_m \mathsf{fuse}\{x_1 : A^{\perp} \mapsto a;\ y_1 : A \mapsto c\} \\
&\qquad\qquad\qquad\qquad \mapsto_{n-m} \mathsf{fuse}\{x_2 : A^{\perp} \mapsto b;\ y_1 : A \mapsto c\} \\
&\qquad\qquad\qquad\qquad \mapsto_m \mathsf{fuse}\{x_2 : A^{\perp} \mapsto b;\ y_2 : A \mapsto d\}\}
\end{aligned}$$

Dynamic checks are expensive at runtime. To avoid them, we would require an oracle that can decide whether the kind corresponding to a proposition of the form $n \leq m$ is inhabited.

In the presence of multiplication, integer arithmetic is undecidable[1] in general, but, for many practical scenarios, it is possible to verify arithmetic predicates automatically.

In our implementation, this oracle takes the form of bounds provided by the user in the form of witnesses (Sec. 3.2), which are fed into a simple solver. The solver can:

- Compare polynomials coefficient-wise, by recognizing that an integer polynomial is ≥ 0 all natural numbers if all its coefficients are ≥ 0.
- In general, recognize that, if $p(x) \geq 0$ holds, then $p(x) + n \geq 0$ holds for every natural number n.

These simple rules are enough to compile all of the examples in Chapter 7 without manual intervention.

Sequence polarization The input to this phase is a typed derivation, where each reference and each name have been annotated with a type and an arity.

In the majority use cases for our language, cut elimination should remove most or all of the sequences introduced in the derivation.

Those that remain are polarized according to the simple polarization method (Sec. 5.1). Notice that the conditions for a single-threaded polarization can be reduced to a boolean satisfiability (SAT) problem, which is decidable. Thus, instead of checking the hypothesis of Thm. 4, and mirroring the induction proof, we can obtain the most general solution (if it exists) by solving the required constraints.

In the output of this phase is also a derivation annotated with types and arities, but all the sequence types have been replaced by either a positive or a negative sequence:

$\S_n\, A \Rightarrow \S^+{}_n\, A$ or $\S^-{}_n\, A$

[1] Non-linear integer arithmetic is powerful enough to express its own consistency; decidability would contradict Gödel's incompleteness theorem.

6.1 Functional translation

The input to this phase is a typed derivation, where all the self-dual connectives (that is, sequences) have been given a polarity, either positive or negative. Because of this, we can now talk exclusively about positive types ($A \otimes B$, $A \oplus B$, $\bigotimes_n A$, $\S^+{}_n A$, 1, 0 and $\mathbb{A}$); and negative types ($A \parr B$, $A \& B$, $\parr_n A$, $\S^-{}_n A$, $\bot$, $\top$ and $\mathbb{A}^\bot$), which are their respective duals.

The key idea behind the embedding of a "classical" logic into an intuitionistic one is the double negation translation, where all values are always closures of the form $\alpha \to \perp\!\!\!\perp$ for some α; $(\alpha \to \perp\!\!\!\perp) \to \perp\!\!\!\perp$ for a positive type.

```
data Pos e where

  VArr    :: (Index (Ref e) → Val e) → Pos e
  VExist  :: Sz e → Val e → Pos e
  VTensor :: Val e → Val e → Pos e
  VPlus   :: Bool → Val e → Pos e
  VOne    :: Pos e
  VAtom   :: String → Pos e


data Val r where
  Pos   :: Pos r → Val r
  Neg   :: (Pos r → r) → Val r
  Shift :: ((Pos r → r) → r) → Val r
```

The double-negation translation realizes the involutive property of linear negation, as an isomorphism between `(Val r → r) → r` and `Val r` :

```
shift  :: Type a b → ((Val r → r) → r) → Val r
shift  (Perp _) v = Neg $ \p → v $ \(Neg f) → f p
shift  _        v = Shift $ \g → v $ \case
                                       Pos p   → g p
                                       Shift k → k g

unshift :: Type a b → Val r → ((Val r → r) → r)
unshift (Perp _) v@(Neg _) k = k v
unshift _        v@(Pos _) k = k v
unshift _        v@(Shift a) k = a (k . Pos)
```

The context of a derivation translates into an environment of indexable names (`Offset`):

```
data Offset e = Offset { at0      :: Val e
                       , demoteAt :: Sz e → Ix e → Offset e
                       }


type Env e = [(Name e,(Offset e, Ty e))]
```

When compiling CLL^n, the translation can be interwined with normalization-by-evaluation, directly yielding a translation. This is implemented as `eval` and `coeval` functions.

```
eval  :: (Eff e, Eq (Name e)) ⇒ Name e → Ty e → LL' e → Env e → (Val e → e) → e
coeval :: (Eff e, Eq (Name e)) ⇒ LL' e → Env e → e
```

In the functional semantics (Chapter 5), there is a common pattern which involves obtaining a negated value from each of the branches of a rule. The `eval` function takes a derivation, singles out one of the free names, and produces a value. This value will have one less open variable than the original derivation.

```
eval name ty@(Perp _) t env n = coeval t ((name,(U (Neg (n . Pos)),ty)):env)
eval name ty t env k          = k $ Neg $ (\x → coeval t ((name,(U (Pos x),ty)):env))
```

`coeval` translates a term into the type of terms ($\perp\!\!\!\perp$); it implements the semantics given in Chapter 5.

```
coeval t env = case t of

  Cut ty x t' y u' → eval x (dual ty) t' env $ \a → coeval u' ((y,(U a,ty)):env)

  Ax x y → case (lookup x env, lookup y env) of
     (Just (N k,_), Just (P p,_)) → p k
     (Just (P p,_), Just (N k,_)) → p k

  Bot z → case lookup z env of Just (N k,I) → k VOne

  One z t' → case lookup z env of Just (P k,Perp I) → k $ \VOne → coeval t' env

  Split z s1 n1 s2 n2 t' → case lookup z env of
    Just (Offset _ f, t) →
      coeval t' ((n1, (vec (at0 . f (s1 + s2))          , t)):
                 (n2, (vec (at0 . f (s1 + s2) . (+ s1)), t)):env)

  Slice z x t → case lookup z env of
    Just (P k,BigTensor _ ta) → k $ \(VArr f) → coeval t ((x, (vec f, ta)):env)

  CoSlice z [(sz,x,t')] → case lookup z env of
    Just (N k,BigPar _ ta) → k $ VArr (\i → shift ta$ eval x ta t' (demoteEnv sz i env))

  Tensor z x y t' → case lookup z env of
    Just (P k,ta :⊗ tb) → k $ \(VTensor a b) → coeval t' ((x,(U a,ta)):(y,(U b,tb)):env)

  Par z x t' y u' →  case lookup z env of
    Just (N k,ta :⅋ tb) → eval x ta t' env $ \a ->
                          eval y tb u' env $ \b → k $ VTensor a b

  Plus z x t' y u' → case lookup z env of
    Just ((P k), ta :⊕ tb) → k $ \(VPlus choice a) → if choice
                                                     then coeval t' ((x,(U a,ta)):env)
                                                     else coeval u' ((y,(U a,tb)):env)

  With c z x t' → case lookup z env of
    Just (N k, ta :& tb) → eval x (if c then ta else tb) t' env $ \a → k $ VPlus c a

  Exist z _tvar x t' → case lookup z env of
    Just (P k, Ex tt) → k $ \(VExist sz a) → coeval t' ((x,(U a,tt sz)):env)

  Forall z sz x t' → case lookup z env of
    Just (N k, All tt) → eval x (tt sz) t' env $ \a → k $ VExist sz a
```

6.1.1 Normalization by evaluation (NbE)

The translation of a derivation will yield an open term of type $\perp\!\!\!\perp$. If we generated code from this term directly, we would incur a big overhead in function calls.

To mitigate this overhead, we make use of normalization-by-evaluation as introduced by Berger et al. [1998]. Most functional languages evaluate only up to WHNF; that is, no β-reductions (terms of the form $(\lambda y.y)\,t$) at the top level. Normalization by evaluation achieves long normal form; that is, no β-reductions at any level, even inside λ-abstractions. The algorithm does so without the need for rewrite rules; instead, a suitable model is chosen, and functions to bring terms into the model and back from it are provided.

More specifically, the normalizing translation can be specialized for a given effect type by providing implementations of $reify$ and $coreify$, which will respectively consume and produce a value.

```
class (Eff e, Ref e ~ Atom e) ⇒ Reifier e where
  type Atom e
  reify   :: Atom e → Ty e → (Val e → e) → e
  coreify :: Atom e → Ty e → Val e → e
```

The reifier allows us to embed values from the logic into the model. In the simpler case,

```
toVal :: Reifier e ⇒ Name e → Ty e → Val e
toVal x (Perp t) = Neg  $ \a → coreify x (Perp t) (Pos a)
toVal x t        = Shift $ \k → coreify x t       (Neg k)
```

Now, we can obtain a closed effect term from an open one, by providing values for each of the open variables.

```
normalize :: (Reifier e, Eq n, n ~ Name e) ⇒ [(Name e, Ty e)] → LL n n → e
normalize ctx t = coeval t [(n, (U$ toVal n t, t)) | (n,t) ← ctx]
```

The nature of the result is determined by the effect type; examples are C code Sec. 6.2, or a derivation (Sec. 6.1.2).

6.1.2 Focusing

Cut elimination is said to reduce *bureaucracy*, by normalizing a class of equivalent proofs into a smaller, normal form. Another way of reducing bureaucracy is by normalizing the order in which connectives are eliminated. Two proofs may remove connectives in a different order, and be otherwise equivalent (Fig. 6.2).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, A^\perp \vdash}\,\text{Ax}}{A^\perp \,\&\, A, A \vdash}\,\&_1 \quad \dfrac{\dfrac{}{A^\perp, A \vdash}\,\text{Ax}}{A^\perp \,\&\, A, A^\perp \vdash}\,\&_2}{A \oplus A^\perp, A^\perp \,\&\, A \vdash}\,\oplus \quad \dfrac{}{\bot \vdash}\,\bot}{A \oplus A^\perp, A^\perp \,\&\, A \parr \bot \vdash}\,\parr$$

(a) *Non-focalized*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, A^\perp \vdash}\,\text{Ax}}{A, A^\perp \,\&\, A \vdash}\,\&_1 \quad \dfrac{}{\bot \vdash}\,\bot}{A^\perp \,\&\, A \parr \bot, A \vdash}\,\parr \quad \dfrac{\dfrac{\dfrac{}{A^\perp, A \vdash}\,\text{Ax}}{A^\perp, A^\perp \,\&\, A \vdash}\,\&_2 \quad \dfrac{}{\bot \vdash}\,\bot}{A^\perp \,\&\, A \parr \bot, A^\perp \vdash}\,\parr}{A \oplus A^\perp, A^\perp \,\&\, A \parr \bot \vdash}\,\oplus$$

(b) *Strongly focalized*

Figure 6.2: *Two LL proofs equivalent with respect to focusing.*

As per Laurent [2004], a proof is strongly focalized when, every time a negative connective is eliminated, all negative connectives immediately nested in it are eliminated too, without any other rules in between. For example, we will say that Fig. 6.2b is and Fig. 6.2a is not.

The semantic framework that we propose also gives a mechanism to focus linear logic proofs. First, observe that derivations form a monoid under MIX, with unit HALT. If we restrict ourselves to Girard's LL, it is possible to define `reify` and `coreify` with derivations as the effect type.

```
instance Monoid (LL a a) where
  mempty = Halt
  mappend = Mix

instance Eff (LL String String) where
  type Ref (LL String String) = String
  type Name (LL String String) = String
  allocate = undefined

instance Reifier (LL String String) where
  type Atom (LL String String) = String

  reify :: (n ~ String, e ~ LL n n) ⇒ Atom e → Ty e → (Val e → e) → e
  reify x t0 k = case t0 of
    t :⊗ u → Tensor x nx ny $ reify nx t $ \a → reify ny u $ \b → kp (VTensor a b)
    t :⊕ u → Plus x nx (reify nx t (kp . VPlus True)) ny (reify ny u (kp . VPlus False))
    Ex t → Exist x nt nx $ reify nx (t $ VSz nt) (kp . VExist (VSz nt))
    I → One x $ kp VOne
    O → Zero x
    Var _ → kp (VAtom x)
    neg@(Perp _) → k $ Neg $ \v → coreify x neg (Pos v)
   where kp = k . Pos
         nx = fresh x "x"; ny = fresh x "y"; nt = fresh x "t"; ix = fresh x "i"


  coreify x typ@(Perp _) (asShift → k) = let \v → case (typ,v) of
      (t :⅋ u,VTensor a b) → Par x nx (coreify nx t a) ny (coreify ny u b)
      (t :& u,VPlus c a) → With c x nx (coreify nx (if c then t else u) a)
      (All t,VExist ty a) → Forall x ty nx (coreify nx (t ty) a)
      (B,VOne) → Bot x
      (Perp (Var _),VAtom y) → Ax x y
    where nx = fresh x "x"; ny = fresh x "y"
  coreify x typ (Neg k) = reify x typ $ \(Pos a) → k a -- Pattern ok because typ is positive
```

One can see that the reified value is focused, because:

1. As long as one finds negative types, one remains in `coreify`.
2. The only way non-negative rules can be introduced into this chain is by means of a `Shift`.
3. Coreify is never passed a `Shift` by `reify` or `coreify`. In fact, shifts are only created in `toVal`, and when interpreting the n-ary `coslice` rule.

Furthermore, the definitions of reify and coreify do not make use of the `cut` rule. This means that, when we restrict ourselves to Girard's LL, normalization corresponds to the combination of cut elimination and focusing.

```
focus :: (e ~ LL n n, n ~ String) ⇒ ([(n,Ty e)],LL n n) → LL n n
focus = uncurry normalize
```

6.2 Code generation

The core of the translation is a representation of linear types as C types, and a description of how to read and write values of a given type.

6.2.1 Memory representation

The function `ctype` translates a CLL^n type into its C representation, according to Fig. 6.3.

ctype :: Type **String String** → Atom C → C

CLL type	Representation
$\mathbb{Z}\bullet, \mathbb{R}\bullet, \ldots$	`int, double`
$A \otimes B\bullet$	`struct{A• pi1; B• pi2;}`
$1\bullet, 0$	`char[0]`
$A \oplus B\bullet$	`struct{bool tag; union{A• inl; B• inr;}}`
$\exists n : \mathbb{N}.A\bullet$	`struct{size_t n; A• value;}}`
$\bigotimes_n A^\perp\bullet$	`void f(size_t i, A•)`
$A^\perp\bullet$	`void f(A• arg1)`

Figure 6.3: *Representation of CLL types in C (unoptimized)*

Optimized representation A major bottleneck in modern architectures is memory access. If intermediate values are not eliminated, choosing an appropiate representation for them can have a strong impact in the performance of the program. This representation is also the interface through which the program will receive inputs and produce results; a level of programmer friendliness is required.

As an example of how function types are translated, consider greaterThan : $\mathbb{Z} \multimap \mathbb{Z} \multimap 1 \oplus 1$. With the standard translation, its type would translate to the baroque:

void greaterThan(**int** a, **void** (*g)(**void** (*h)(**int**,
void(*c)(**struct** { bool tag; **union** { **void** inl; **void** inr; } val; }))))

Instead, we add a number of rules to represent types in CLL^n in a way that is closer to standard practice in C programming.

Primitive booleans The type $1 \oplus 1$ can be rendered as a standalone `bool`. This has no effect on performance, but simplifies the FFI from the programmers perspective.

`void greaterThan(int a, void (*g)(void (*h)(int, void(*c)(bool))))`,

Uncurrying Partial application of a function may perform work with the first argument that should be done only once; regardless of how many times the result is used.

However, in CLL^n, values are used *exactly once*. As a consequence, partial application can always be implemented by keeping all arguments in scope, and calling the function only once.

`void greaterThan(int a, int b, void(*c)(bool))`

This optimization applies to the connectives $A \multimap B$, $\forall\alpha : \mathbb{N}.B$, and $A \& B$.

Output arguments If the dual of a type is plain data, it can be returned by using a pointer to a memory location, instead of an anonymous function.

In our example, the type can be reduced to:

`void greaterThan(int a, int b, bool* c)`

Memory allocation is handled according to the type.

Stack-based output If the function returns a single, primitive value, it can be returned using the stack, instead of using a pointer. Thus, we obtain a very similar type to what a developer may have written:

`bool greaterThan(int a, int b)`

Empty fields A field of type `void` is not valid C. All such values are removed from the final representation. Products, sums and vectors of empty types are also empty.

```
struct { bool tag; union { int inl; } }*
```

Sparse sequences If the sequence is sparse, the straightforward representation will have poor data locality.

By storing the `tag`s using run-length encoding, and putting the data of a single type on one block, we can greatly improve performance.

```
struct { size_t * skips; int* inl; }*
```

This representation generalizes to any sequence of type $\S_n(A \oplus B)$ or $\S_n(A \oplus B)^\perp$, where $A \oplus B$ is a data type.

The performance impact is most significant when the in-memory sizes of A and B are very different, because there will be less fragmentation in the array, and, as a consequence, less wasted space in cache.

Because these optimizations depend exclusively and predictably on the types involved, and not on the use that is made of the values, they are consistent with our goals of guaranteed efficient compilation.

6.2.2 Memory access

We can generate C code to manipulate CLL^n values by providing a `Reifier` instance.

First, imperative C code forms a monoid under sequential composition, where the identity is the empty instruction. In this definition, the ⊡ operator performs non-semantic concatenation of C code, treated as a string.

```
newtype C      = C        String deriving (IsString, Syntax)

instance Monoid C where
  mempty = ";"
  mappend a b = a ⊡ ";␣" ⊡ b
```

Note that, although the input language is functional, the language can still be compiled easily without closure objects or a garbage collected heap. This is because all memory allocation happens in a first-in, first-out basis, due to the tree structure of the derivations.

Because all the variables that a closure depends on stay on the stack during the lifetime of the closure, we can restrict ourselves to the lexical closures introduced by Breuel [1988]. For a given derivation, the depth of the stack at runtime is bounded by the depth of the derivation tree (assuming a normalized form where the axiom rule is only applied to primitive types).

The allocation function shown here is a greatly simplified version, appropiate only for primitive data types.

```
instance Eff C where
  type Ref  C = String
  type Name C = String
  allocate sz ty n k =
    ctype (ty) (var n ⊡ "[]") ⊡ ";" <>
    var n = "malloc(" ⊡ idx sz ⊡ "␣*␣sizeof(*" ⊡ var n ⊡ "))" <>
    k (\i → reify (var n ⊡ "[" ⊡ idx i ⊡ "];") ty)
      (\i → reify (var n ⊡ "[" ⊡ idx i ⊡ "];") (dual ty)) <>
    "free(" ⊡ var n ⊡ ")"
```

Finally, we implement the `reify` and `coreify` functions:

```
instance Reifier C where
  type Atom C = String
```

```
  reify :: (e ~ C) ⇒ Atom e → Ty e → (Val e → e) → e
  reify x t0 k = case t0 of
    t :⊗ u → ctype t nx ⊡ "=" ⊡ var x ⊡ ". left ;\n" ⊡
             ctype u ny ⊡ "=" ⊡ var x ⊡ ". right ;\n" ⊡
             (reify nx t $ \a → reify ny u $ \b → kp (VTensor a b))
    t :⊕ u → "if ("⊡var x⊡".tag){" ⊡ ctype t nx ⊡ "=" ⊡ var x ⊡ ". info . inl ;\n" ⊡
                                      reify nx t (kp . VPlus True) ⊡
                                "\n␣}else{␣\n" ⊡ ctype u ny⊡"␣="⊡var x⊡".info.inr␣" ⊡
                                      reify nx u (kp . VPlus False) ⊡ "\n};\n␣"
    I → kp VOne
    O → "abort();"
    Var _ → kp (VAtom x)
    BigTensor len t → kp (VArr (\i → unreify' nz t $ \x' → "x(" ⊡ idx i ⊡ ",␣" ⊡ var x' ⊡ ")"))
    neg@(Perp _) → k $ Neg $ \v → coreify x neg (Pos v)
   where kp = k . Pos
         nx = fresh x "x"; ny = fresh x "y"; nz = fresh x "z"


  coreify :: (e ~ C) ⇒ Atom e → Ty e → Val e → e
  coreify x ty v = unreify x ty v $ \x' → var x ⊡ parens (var x') ⊡ ";\n"

unreify' :: (n ~ String, eff ~ C) ⇒ Atom eff → Type n n → (Atom eff → eff) → Val eff
unreify' z typ κ = case typ of
  Perp t0 → Neg $ \p → unreify z typ (Pos p) κ
  _       → Shift $ (\μ → unreify z typ (Neg μ) κ)

unreify :: (n ~ String, eff ~ C) ⇒ Atom eff → Type n n → Val eff → (Atom eff → eff) → eff
unreify z typ b κ = (⊡ κ (co z)) $ case (typ,b) of
      (Perp t0, Pos v) → ctype t0 (co z) ⊡ ";\n" ⊡
        case (typ,v) of
          (t :⅋ u,VTensor a b) → unreify nx t a $ \nx' ->
                                 unreify ny u b $ \ny' ->
                                 var (co z) ⊡ ". left␣=␣" ⊡ var nx' ⊡ ";\n" ⊡
                                 var (co z) ⊡ ". right␣=␣" ⊡ var ny' ⊡ ";\n"
          (t :& u,VPlus c a) ->
              var (co z) ⊡ ".tag␣=␣" ⊡ (if c then "0" else "1") ⊡ ";\n" ⊡
              (unreify nx (if c then t else u) a $ \nx' ->
              var (co z) ⊡ ". info ." ⊡ (if c then "inl" else "inr") ⊡"␣=␣" ⊡ var nx' ⊡ ";\n")
          (B,VOne) → ";\n"
          (Perp (Var _),VAtom y) → var (co z) ⊡ "␣=␣" ⊡ var y ⊡ ";\n"
          (BigPar len t,VArr v) → "void␣" ⊡ var (co z) ⊡ "(int␣" ⊡ var ni ⊡ ",␣"⊡ ctype t z⊡")␣{␣"⊡
                                      coreify z typ (v (VSz ni)) ⊡
                                  "};\n"
        (_,   Neg k) → "void␣" ⊡ var (co z) ⊡ "("⊡ ctype typ z⊡")␣{␣"⊡ reify z typ (\(asShift → a) → a k) ⊡
       where nx = fresh z "x"; ny = fresh z "y"; ni = fresh z "i"
             co = ("CO_" ⊡)
```

If we apply `normalize`, as was done in Sec. 8.3, we obtain the compiler; focusing is intertwined with the code output.

```
compile :: (e ~ C, n ~ Name C, Eq n) ⇒ ([(Name e,Ty e)],LL n n) → e
compile = uncurry normalize
```

7 Examples and benchmarks

We use the sequences extension to implement and benchmark kernels for computing 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.

7.1 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.

7.2 Wave propagation stencil

A popular way of simulating physical systems is by dividing the space into a mesh, and time into discrete steps. Then, given physical parameters on each vertex on the mesh (e.g. energy, amplitude, temperature, speed, acceleration…) on a particular time-step, the values on the next time step for each point of the grid depend in a simple way on the values on the neighbours.

Motivation A numerical solution for wave propagation can be described as a stencil computation [Hoekstra et al., 2014]. 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 at times $t-1$ and $t-2$ (Fig. 7.2).

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, 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. 4.4, we use a sequence type to implement this stencil operation. Because there are two arrays involved, we use the type $\S_n(\mathbb{R}\otimes\mathbb{R}) \multimap \S_n(\mathbb{R}\otimes\mathbb{R})$.

CLL Implementation We decompose the algorithm using a linear version of standard arrow combinators [Hughes, 2000]. With them, streaming computations can be expressed as morphisms and products in a suitable category. The resulting data flow is shown in Fig. 7.3.

Measurement We compare the program against the following C code:

```
void wave0 (size_t n, double tau, double *a, double *b, double *d) {
    d[0] = waveL(tau, a[0], b[0], b[1]);
    for(int i = 1; i < n-1; i++) {
      d[i] = waveC(tau, b[i-1], b[i], b[i+1], a[i]);
    }
    d[n-1] = waveR(tau, a[0], b[0], b[1]);
}
```

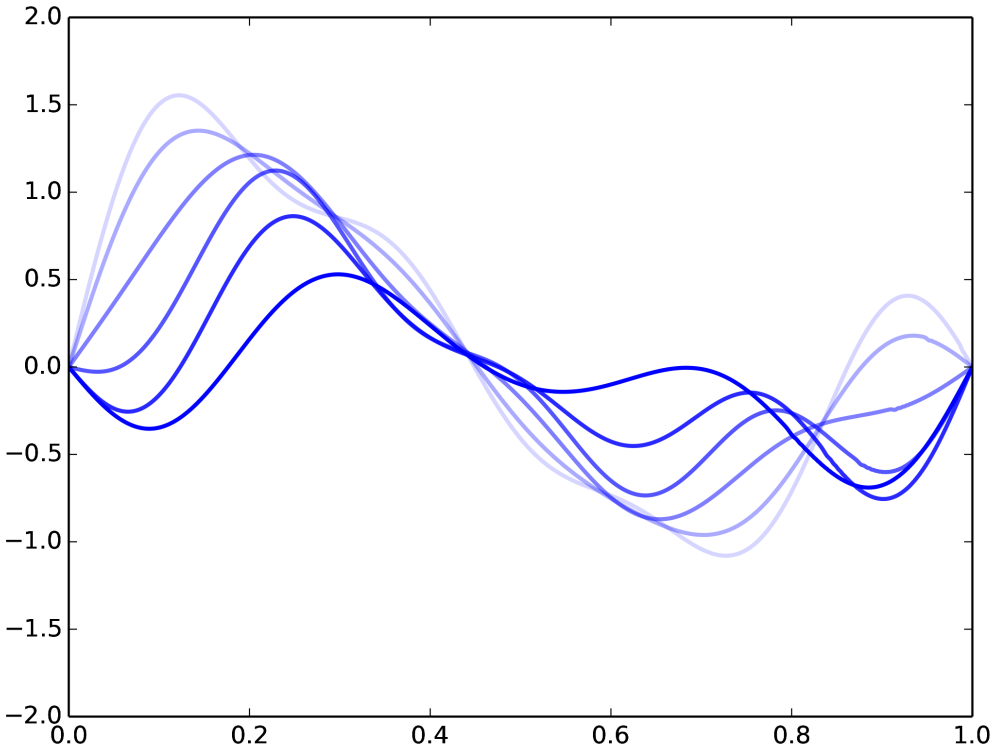


Figure 7.1: *Successive steps of wave stencil*

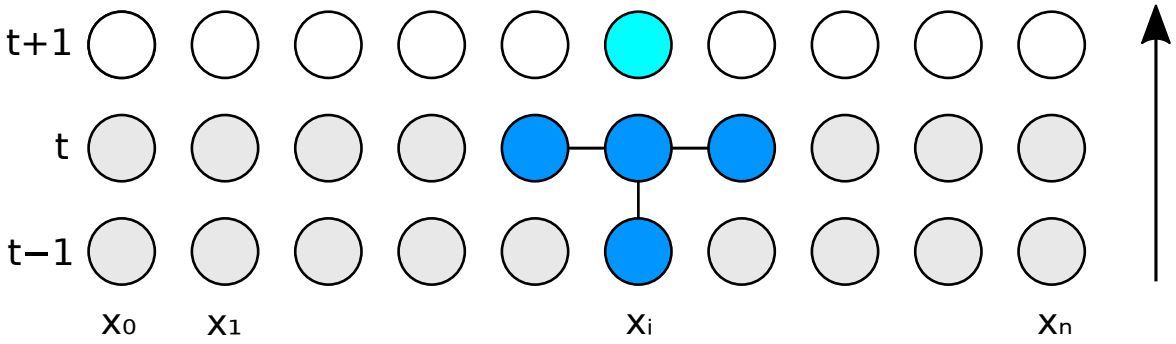


Figure 7.2: *Stencil for 1-dimensional wave propagation*

$(***) \;:\; \S_n(A \multimap B),\ \S_n(C \multimap D),\ (\S_n(A \otimes C) \multimap B \otimes D)^\perp \vdash$
$(\&\&\&) :\ \S_n(D \multimap B),\ \S_n(D \multimap C),\ (\S_n(D \multimap B \otimes C))^\perp \vdash$
$(>>>) :\ \S_n(A \multimap B),\ \S_n(B \multimap C),\ \S_n(A \multimap C) \vdash$
$(++) \;:\; \S_n A,\ \S_m A,\ \S_{m+n} A \vdash$
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 7.3: *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.*

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).

Analysis The performance difference is due to the fact that the code that duplicates work has a simpler structure; as such, GCC can optimize it into vectorized instructions. This optimization would not be applicable if the operation being performed on each cell was more complex than linear arithmetic.

7.3 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 [Barber et al., 1996] 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. [2013], 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 (Fig. 7.5). 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.

- Select leftmost and rightmost points in the array (1 pass).

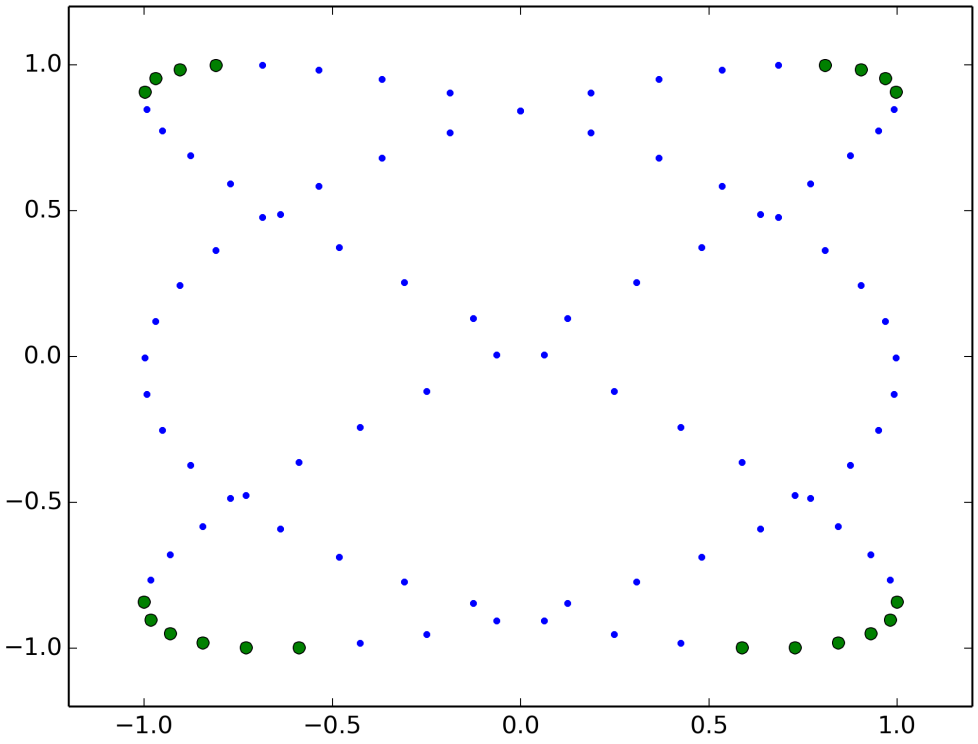


Figure 7.4: *Convex hull of Lissajous figure*

- 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 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.

Methodology We compare against the following C code:

#define X vec[0]
#define Y vec[1]

typedef struct { **double** vec[2]; } point;

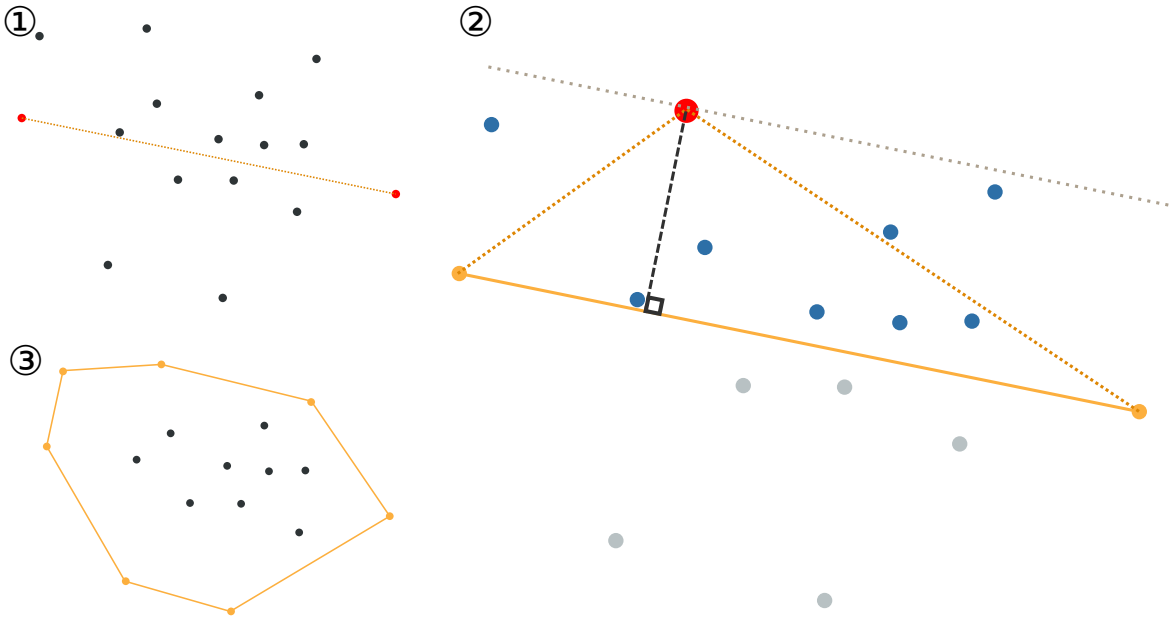


Figure 7.5: *QuickHull algorithm: (1) Selection of two seed points, (2) recursive step selecting the points above the line, and the one furthest away from it, yielding two more lines; and (3) the end result, where the vertices of the enclosing polygon form the convex hull. Diagram based on [Westhoff, 2009], CC-BY-SA 3.0.*

```
typedef point vector;

/* we don't divide by the norm of v, since we only care about > 0, and comparison */
/* Distance should be exactly zero if both vectors are equal, or one of them is zero */
static inline double distance(vector v, vector pv) {
    return pv.Y * v.X - v.Y * pv.X;
}

static inline vector diff(point b, point a) {
  vector result = { { b.X - a.X, b.Y - a.Y } };
  return result;
}

void quickhullStep1Manual (size_t n, const point * all, point* left_, point* right_) {
  point left, right;
  left = right = all[0];

  for(int i = 1; i < n; i++) {
    const point p = all[i];
    if (p.X < left.X) { left = p; }
    else if (p.X > right.X) { right = p; }
  }

  *right_ = right;
  *left_ = left;
}

void quickhullStep2Manual (size_t n, const point * all,
```

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

$\text{splitTopAbove} : \text{all} : \S_n(\mathbb{R} \otimes \mathbb{R} \oplus 1),$
$\quad \text{top'} : (\mathbb{R} \otimes \mathbb{R} \oplus 1)^\perp,\ \text{above'},\ \text{rest'} : (\S_n(\mathbb{R} \otimes \mathbb{R} \oplus 1))^\perp \vdash$
cut { comap1 fst rest' } rest_0' $\mapsto$
cut { comap1 fst above' } above_0' $\mapsto$
cut { withDistTo left right } withDistance_1 $\mapsto$
cut { map1 withDistance_1 all } all_0 $\mapsto$
cut { map1 withDupl all_0 } all_1 $\mapsto$
cut { awd' $\mapsto$ splitS (≥ 0) all_1 awd' rest' } awd $\mapsto$
cut { above_1' $\mapsto$ copy1 awd above_0' above_1' }
{ above_1 $\mapsto$ best1 $(\geq)$ above_2 top' }

Figure 7.6: *QuickHull kernel. The set of points above the line containing segment AB, and the one furthest away from this line are computed.*

```
                                point left, point right,
                                size_t* r_, point * above,
                                size_t* s_, point * below,
                                point* max_, point* min_) {
  point min; point max; size_t r = 0, s = 0;

  vector v = diff(right, left);

  double dmin = 0; double dmax = 0;

  for(int i = 0; i < n; i++) {
    const point p = all[i];
    point pv = diff (p, left);
    double d = distance(v, pv);
    if (d > 0) {
      above[r] = p; r++;
      if (d > dmax) { dmax = d; max = p; };
    } else if (d < 0) {
      below[s] = p; s++;
      if (d < dmin) { dmin = d; min = p; };
    }
  }

  *max_ = max;
  *min_ = min;
  *r_ = r;
  *s_ = s;
}
```

```
void quickhullStep3Manual (size_t n, const point * all, point left, point right, point top,
                           size_t *r_, point *lefts,
                           size_t *s_, point *rights,
                           point* top_left_, point* top_right_) {

  point top_left; point top_right; size_t r = 0, s = 0;

  double dleft = 0, dright = 0;

  vector v1 = { { top.X − left.X, top.Y − left.Y } };
  vector v2 = { { right.X − top.X, right.Y − top.Y } };

  for(int i = 0; i < n; i++) {
    const point p = all[i];
    point pv1 = diff(p, left);
    double d = distance(v1, pv1);
    if (d > 0) {
      lefts[r] = p; r++;
      if (d > dleft) { dleft = d ; top_left = p ; }
    } else {
      point pv2 = diff(p, top);
      double d = distance(v2, pv2);
      if(d > 0) {
        rights[s] = p; s++;
        if (d > dright) { dright = d ; top_right = p; }
      }
    }
  }

  *top_left_ = top_left;
  *top_right_ = top_right;
  *r_ = r;
  *s_ = s;
}
```

CLL is only used to compute the filterMap style operations. An external driver shared by both the C and the CLL^n implementations performs the recursive calls.

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.

Analysis The code deriving from CLL has much more branching than the explicit C code. This derives from the higher level of abstraction with which the linear code has been written. In particular:

1. Folds over option types.
2. Copies of option types. No specialized code exists for copying.
3. Input and output as sparse arrays. Although more efficient than the straightforward representation, the unnecessary sparsity still imposes overhead.

A good code generator should resort to an optimized `memcpy` when connecting two memory locations, instead of fully reading and writing the values.

On the other hand, the use of option types could be almost fully eliminated from this example with the implementation of Girard et al. [1992]'s bounded sizes.

7.4 Summary

The run times for the examples are summarized in Fig. 7.7. By applying cut elimination and a straightforward compilation scheme, it is possible 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/`.

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 7.7: *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.*

8 Discussion

What we present here is not a fully-fledged development environment, but a core language with good performance guarantees and minimal bureaucracy. This way, users of the language can focus on the more relevant trade-offs between work duplication and memory usage while safely ignoring the rest.

8.1 Distinctive features

There are two points that may result counterintuitive in view of previous work in linear logic and functional programming.

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 $\parr$-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.

8.2 Applications

We contemplate two possible venues for making CLL^n into a fully-fledged language:

Core language A straightforward application of the calculus is as the core language for a functional programming language. Duality makes for an economic design, and linearity reduces the need for heuristics for fusion optimizations and can give strong correctnes guarantees. Although it is not the focus of this work, the same linearity can be leveraged to model resource allocation and side effects, and ensure that they are respected by transformations. In this application, the designer of the extended language has a great deal of latitude in how much of the linear features of the core language will be exposed to the language user.

Preprocessor Bootstraping a language requires a big investment to develop libraries, documentation and toolchains.

However, when compiling the language, we have achieved a close correspondence between linear types and C types. Furthermore, linearity of the calculus allows for a predictable interplay between CLL^n code and low-level code; despite the reliance on side effects of the later.

Instead of developing the language from scratch, we can rely on low-level code and libraries for writing the smaller block of the language, and make CLL^n into a DSL that composes these blocks effectively.

8.3 Limitations and future work

Sizes The type checker is limited in the depth of the reasoning involving sizes that it can make. For example, it cannot deduce $1 \leq n$ from a chain of inequalities $1 \leq n_1$, …, $n_k \leq n$.

Even if we could improve our current solver to perform more general reasoning, integrating with an off-the-shelf SMT solver would be a longer-term solution.

Polymorphism The current implementation of the code generator does not support parametric *type* polymorphism. It does, however, support *size* polymorphism. The programmer can still use quantified types in their language, as long as they are introduced by fuse rules.

Sequences and parallelism The type of sequences forces a linear traversal of arrays. When converting between $\otimes$- and $⅋$-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.

Heterogeneous sequences Sequence arrays are a form of non-commutative connectives; unlike the case of tensor or $⅋$-arrays, the order of the elements cannot be changed.

As of Guglielmi [2004], a complete treatment of non-commutative linear logic has not yet been achieved in a sequent calculus. However, the use of sequent calculi are very advantageous, because their tree structure is very amenable for computation.

Our sequent calculus can express a range of programs involving sequences. An important limitation is that a sequence has to be processed in a single reduction step.

Because we have precise control over the number and position of the elements in a sequence, a natural extension to the calculus is allowing for elements to have different types depending on the index. If we restrict ourselves to the case where the range of types is finite, we obtain the following:

$\S[A_1^{n_1}; A_2^{n_2}; \ldots; A_k^{n_k}]$

Using this, we can embed a non-commutative binary operator $A \rhd B \stackrel{def}{=} \S[A; B]$, which fulfils the property $A \rhd A \stackrel{def}{=} \S_2\, A$.

Associativity and commutativity of binary and n-ary operators Binary operators ($\otimes$, $⅋$, $\oplus$, &) are provably associative within the system. When they are generalized to n-ary connectives, it is possible in both cases to reorder the elements by splitting the array into one or more chunks, and merging the pieces back together in a different order.

However, this does entail that n-ary operators do not commute with themselves; e.g. $\bigotimes_m \bigotimes_n A \bigotimes_n \S_m A$.

The reason for not adding this rule is that, in some models of the system (namely, the one that translates the terms into structured C code), commutativity is realized by shifting control to each of the operands. Each of these control-shifts could potentially create a new level of nesting in the code, or a stack frame at runtime. Commuting n-ary operators would require an unbounded number of levels of nesting, which is not desirable.

A solution to this problem is to generalize heterogeneous arrays into multi-level arrays,

$\bigotimes[A_1^{n_1}; [A_2^{n_2}; \ldots]; \ldots; A_k^{n_k}]$.

and to allow for arbitrary permutations both within the same level, and across levels. Because the array would be computed with coslice all at once, the issue of nested blocks does not arise.

This notation also brings formal properties that are easily realized with par and tensor arrays to the domain of sequences; for example, the isomorphism $\S_2\, \S_m\, A \;\; \S_{2m}\, A$

Cost metrics One technical complication arises from costs that depend on decisions made at runtime, such as branching, types dependent on sizes, and parametrically-polymorphic types. At the moment, a pessimistic approximation is made by taking the maximum cost of both options.

For more precision, we make use of an oracle for the values, whose answer is uniquely determined by the environment and input of the program, and whose answer will be preserved by cut elimination.

Bounded sequences Another limitation is that the rate at which the values are processed cannot be dependent on their values. Achieving more flexible schedules is possible, but would require a more sophisticated cut elimination algorithm.

The calculus can express bounds on sizes (see Sec. 3.2). However, in the current implementation, when accessing a sequence, the size must be known before the first element is produced. This limits the use of fusion between functions that may remove elements from a list, such as `filter`; requiring an approximation using sparse sequences (that is, sequences of type $\S_n(1 \oplus A)$).

Focusing The correspondence between focusing and normalization from Sec. 8.3 does not generalize to the n-ary fragment, for two core reasons:

- Arrays in the model can be arbitrary maps from the type of sizes to the type of values. These maps cannot always be represented as a COSLICE, which has a constant, fixed number of branches. Furthermore, the fact that a `Shift` is introduced means that the result will not necessarily be focused.
- When allocating state, sequentiality and boundedness of updates is not enforced. The constraints imposed by the sync rule are more strict.

A full correspondence requires harmonizing branching on indices and the representation of non-commutativity between the model and the calculus.

8.4 Related work

Optimization of compilers for both imperative and functional array languages is a field where extensive work has been done. We explain how the results achieved in CLL^n relate to common compiler optimizations.

8.4.1 Loop fusion

Each loop in a program incurs a certain amount of overhead to keep track of the indices, and branch on them.

Optimizing compilers often implement loop fusion, which combines several loops into a single one. That way the cost associated with loops, such as the test and branch, can be reduced.

Loop fusion is typically implemented using heuristics, just like most common compiler optimizations, which makes it hard for the programmer to understand exactly when it triggers.

In CLL^n, a loop consuming of type $\S_n A$ is equivalent to a value of type $(\S_n A)^\perp$.

Each application of traverse introduces one loop. After cut elimination, two sequences introduced at opposite sides of cut are guaranteed to be consumed by the a single traverse rule; i.e. in the same loop.

First, observe that we can then produce any number of "entangled" schedules; all of which will share the same counter; by traversing them together.

```
loopFusion ≡ s : (§n 1)⊥, t : (§n 1)⊥ ⊢
  traverse { s as s', t as t' ↦
    mix { yield to s'
        ; yield to t' } }
```

To implement loop fusion, observe that, whenever we have two sequences (for example, a sequence of type $\S_n A$ and a *schedule* of type $\S_n 1$), we can ensure that they will use the same counter by traversing them together.

```
scheduleSequence ≡ s : §_n 1, f : §_n A, fs : (§_n A)^⊥ ⊢
  traverse { s as s', f as f', fs as fs' ↦
             let ⋄ = s' ; f' ↔ fs' }

example ≡ f : §_n A^⊥, x : §_n A^⊥, g : §_n B^⊥, y : §_n B^⊥ ⊢
  cut { s' ↦ cut { t' ↦ loopFusion s' t' } }
      { s ↦  cut { x0' ↦ scheduleSequence s x x0' }
                 { x0 ↦  f ↔0 x } }
      { t ↦  cut { y0' ↦ scheduleSequence t y y0' }
                 { y0 ↦  g ↔0 y } }
```

For comparison, using two sequential applications of traverse will give two independent loop indices:

```
loopNoFusion ≡s : §_n ⅄, t : §_n ⅄ ⊢
  mix { traverse { s as s' ↦ yield to s' }
      ; traverse { t as t' ↦ yield to t' }
      }
```

Crucially, although the programmer has to explicitly apply loop fusion, the idiom can be encapsulated inside a combinator, and applied to any sequence; regardless of the program that consumes it.

Note that values of the type $\S_n 1 \parr \S_n 1$ contain no information. As such, uneliminated cuts over such types do not allocate any memory; the elimination of the cut only affects the number of loops in the generated code. This reflects how loop fusion does not remove any data, and is simply concerned with the control flow of programs.

Finally, observe that fusing loops will reduce total code size, but will also increase the amount of code run in each iteration. If the loop body is too big for the instruction cache, the penalty of the cache misses could nullify the benefits of loop fusion.

8.4.2 Loop fission

While loop fusion commutes MIX into TRAVERSE, loop fission does the exact opposite: commuting a MIX out of one of the branches of a traverse. This is in general only possible if the MIX is the first rule applied in the branch; which is not possible to guarantee using linearity.

We advocate writing programs using a compositional style, where the combination of two unrelated operations within the same loop should be avoided. In these situations, loop fission should be understood as an unrealized loop fusion.

8.4.3 Loop unswitching

A third optimization involving loops is loop unswitching. Branching is an expensive operation, not only because of cost of the comparison itself, but mainly because of the potential pipeline stalls.

Constant branches The easiest case is when the result of the branch is independent of the loop index or any variables that are modified by the loop body.

```
void maybe_negate(bool neg,
                  size_t n,
                  const double a[],
                  double b[]) {
  for(int i = 0; i < n; i++) {
    if (neg) {
      b[i] = 1 - a[i];
```

```
    } else {
      b[i] = 1 + a[i];
    }
  }
}
```

Optimizing compilers can detect this situation, and create one version of the loop for each possible result of the branch. The branch condition will only be checked once, and the appropiate instance of the loop will be run without further checks.

```
void maybe_negate(bool neg,
                  size_t n,
                  const double a[],
                  double b[]) {
  if (neg) {
    for(int i = 0; i < n; i++) {
      b[i] = 1 - a[i];
    }
  } else {
    for(int i = 0; i < n; i++) {
      b[i] = 1 + a[i];
    }
  }
}
```

In some sense, this optimization is not very meaningful in a linear language, because each variable can only be used once. Once the programmer creates copies of the branch condition for each of the tests, the fact that they all represent the same value is no longer straightforward to deduce.

But it is precisely the need to duplicate the branch condition explicitly that will discourage the programmer from writting unswitched code in the first place. Instead of using a heuristic, the programmer is nudged into doing the optimization himself.

Range branches In the general case, the branch condition may depend on the loop index. Then, the branch cannot be commuted in general. However, the branch condition sometimes depends on the loop index alone:

```
void append(size_t n, const double a[],
            size_t m, const double b[],
            double c[]) {
  for(int i = 0; i < n + m; i++) {
    if (0 <= i && i < n) {
      c[i] = a[i];
    } else {
      c[i] = b[i];
    }
  }
}
```

Then, the full range of indices can be partitioned into subranges where the value of the condition is constant. For example:

```
void append(size_t n, const double a[],
            size_t m, const double b[],
            double c[]) {
  for(int i = 0; i < n; i++) {
    c[i] = a[i];
  }
  for(int i = n; i < n+m; i++) {
```

```
    c[i] = b[i];
  }
}
```

In CLL^n, loop unswitching is a central consequence of the cut elimination process. After all cuts are eliminated, only sequences introduced by sync, LOOP remain, all of which are positive. If all the sequences in a traverse rule are positive, then the rule has full control on the loop index, and can generate unswitched code.

8.4.4 Data-flow Fusion

Lippmeier et al. [2013] 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 [Gill et al., 1993]. 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 [Turchin, 1986], deforestation [Wadler, 1990] and fixed point promotion [Ohori and Sasano, 2007]. 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.

Commutative arrows Liu et al. [2009] optimizes commutative arrows to a normal form with a single loop and one initial state. An analogous result follows directly from CLL^n without requiring additional compiler machinery: namely, the composition of sequences of any data type (including functions) reduces to a single traverse rule together with a tuple of initial values. Wave propagation Sec. 3.4 is one such example.

8.4.5 Recursive data structures

Optimization in functional languages deals with recursive data structures. For example, Wadler [1990] deals with lists, and trees in general, while Ohori and Sasano [2007] uses patterns arising from recursive function calls. Both approaches take a general language, and delimit a subset of where the optimizations apply.

Our approach takes a calculus where fusion properties are guaranteed (linear logic), and makes it applicable to a wider domain (array computations). Recursive data structure are one more step in the roadmap.

8.5 Conclusion

We have shown how CLL constitutes a versatile framework in which to express array computations. Being a small language, the ideas CLL^n can be applied in both low-level and user-facing settings.

CLL^n introduces a mirror-world of types to the sums, products and arrays of traditional functional programming. Concepts like iteration that are trivial in a functional require a subtler understanding in the dual world.

Once familiarized with the other side of the mirror, the programmer can leverage this understanding and enjoys essentially free higher-order functions and intermediate names. This duality allows for

a new, symmetric way of reasoning about programs: one that achieves a new optimal in the tradeoff between compositionality and performance.

Bibliography

L. Augustsson, S. Peyton Jones, et al. Pattern synonyms. 2011. URL `https://ghc.haskell.org/trac/ghc/wiki/PatternSynonyms`.

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.

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. doi: 10.1145/235815.235821.

U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. In B. Möller and J. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 117–137. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-65461-2. doi: 10.1007/3-540-49254-2_4. URL `http://dx.doi.org/10.1007/3-540-49254-2_4`.

J.-P. Bernardy and J. Svenningsson. Controlled array fusion using linear types. URL `http://www.cse.chalmers.se/~bernardy/controlled-array-fusion.pdf`.

T. M. Breuel. Lexical closures for c+. 1988.

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.

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

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

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. doi: 10.1016/0304-3975(87)90045-4.

J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1 – 66, 1992. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(92)90386-T.

A. Guglielmi. A system of interaction and structure. Technical report, ACM TRANSACTIONS ON COMPUTATIONAL LOGIC, 2004.

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

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

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

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. doi: 10.1145/1863543.1863582.

O. Laurent. A proof of the focalization property of linear logic. Note available on the author's web page., 2004. URL `http://perso.ens-lyon.fr/olivier.laurent/llfoc.pdf`.

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.

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

D. Miller and A. Saurin. From proofs to focused proofs: a modular proof of focalization in linear logic. In *CSL 2007: Computer Science Logic, volume 4646 of LNCS*, pages 405–419. Springer-Verlag, 2007.

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

F. Pfenning. Structural cut elimination. In *Logic in Computer Science, Symposium on*, page 156. IEEE Computer Society, 1995.

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

H. Thielemann. monoid-transformer: Monoid counterparts to some ubiquitous monad transformers. 2009. URL `https://hackage.haskell.org/package/monoid-transformer-0.0.3`.

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

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

P. Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA, 2012. ACM.

J. Westhoff. Quickhull. 2009. URL `https://de.wikipedia.org/wiki/Benutzer:Jakob.westhoff/QuickHull`.

List of Figures

Glossary

CLL Classical Linear Logic. 6, 7, 9, 11, 14, 33

floating-point is a fixed-length representation for real numbers in a computer.

A binary floating-point number $^1 1.m \cdot 2^e$ is represented as a sign bit b, a mantissa m, and an exponent e. Some codes are reserved to represent 0, $^1\infty$, and arithmetic exceptions.

The sizes of the mantissa and exponent are bounded statically when the program is compiled, and depend on the architecture.

Floating point numbers are cheap because processors have dedicated instructions to perform arithmetic on them, but lack mathematical properties such as associativity of addition (i.e. $a + (b + c) \neq (a + b) + c$ in some cases), which makes it hard to reason about errors. 1

FLOPS floating-point operations per second. 1

HPC high-performance computing. 1

intuitionistic logic Propositional logic without the axiom of excluded middle.. 6

meta-theoretical Refering to the surrounding theory in which a system is described.. 33

monoid is a tuple $(M, \cdot, 1)$ such that:

$$\cdot : M \to M \to M$$

$1 \in M$

For every $x, y, z \in M$:

$$x \cdot 1 = 1 \cdot x = x$$

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

16

SAT boolean satisfiability. 44

stencil is a map from a tensor to a tensor of the same dimensions and size (e.g. a vector (1 dimensional stencil), a matrix (2 dimensional stencil) and so on) where the value of each cell in the result depends on the values of a fixed number of neighbours on the input. 1

structural rule Structural rules are those which do not act on any particular connective, but on the structure of the context itself.. 7

the Curry-Howard isomorphism Correspondence between proofs in constructive logic and computer programs.. 5