Strong Sums in Focused Logic

Karl Crary
Carnegie Mellon University
crary@cs.cmu.edu

Abstract

A useful connective that has not previously been made to work in focused logic is the *strong sum,* a form of dependent sum that is eliminated by projection rather than pattern matching. This makes strong sums powerful, but it also creates a problem adapting them to focusing: The type of the right projection from a strong sum refers to the term being projected from, but due to the structure of focused logic, that term is not available.

In this work we confirm that strong sums can be viewed as a negative connective in focused logic. The key is to resolve strong sums' dependencies eagerly, before projection can see them, using a notion of *selfification* adapted from module type systems. We validate the logic by proving cut admissibility and identity expansion. All the proofs are formalized in Coq.

ACM Reference Format:
Karl Crary. 2018. Strong Sums in Focused Logic. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209108.3209145

1 Introduction

The power of a logical framework comes not from what can be written, but rather from what *cannot* be written. The theory of adequacy of the representation of a deductive system in a logical framework [10] works by drawing an isomorphism between expressions in the object language (the system being represented) and canonical forms in the logical framework. But this relies on the existence of strong canonical forms. It is essential that there not exist exotic terms that are not the representation of an object-language expression.

The classic example is the representation of lambda terms in the LF logical framework [10]. In it, terms are represented using a type `exp` and two constants:

```
lam : (exp -> exp) -> exp
app : exp -> exp -> exp
```

For instance, the lambda term $\lambda x.xx$ is represented $\texttt{lam}(\lambda x{:}\texttt{exp}.\,\texttt{app}\,x\,x)$. It is important that one cannot write something like $\texttt{lam}(\lambda x.\,\texttt{case}\,x\,\texttt{of}\ldots)$, because such a term is not within the range of the representation. (Note that the pattern match *itself* is the problem, not any ability to branch.)

Thus, the suitability of a logical framework stems from the code that cannot be written. In the case of LF, the only supported non-atomic type is a dependent function space, and pattern matching on atomic types is not supported. Thus, no types that involve pattern matching are supported.

Later, when LF was generalized to Linear LF [3], it added the linear function space ($\multimap$), additive conjunction ($\&$), and its unit ($\top$). However, it did not support other important types such as multiplicative conjunction ($\otimes$) or exponentials because they involve pattern matching and would therefore spoil the canonical forms.

In retrospect, we can observe that these early logical frameworks maintained strong canonical forms by supporting only negative types [9], types that are invertible on the right but not on the left. Positive types (the opposite) were excluded because left inversion is pattern matching.[1]

Nevertheless, there are good reasons to want to support positive types in a logical framework. One reason is that positive types are essential to some applications, such as forward-chaining logic programming, which gives a good treatment of the evolution of concurrent stateful systems [24, 29], or session types [27]. Another reason, which motivated this work, is we would like to integrate a logical framework with a functional programming language, and this requires richer programming facilities. (Existing approaches [8, 20, 21] have used separate logic and programming layers, rather than a single integrated language.)

Focusing Intuitionistic focused logic [1, 15, 25, 30] can allow negative and positive types to coexist in a logical framework without spoiling canonical forms. It derives from a restricted proof system for linear sequent calculus:

The idea is one alternates between inversion and focus stages. In an inversion stage, one applies all available invertible rules, decomposing negative connectives on the right and positive connectives on the left. Once no invertible steps are left (so the conclusion is positive or atomic, and all hypotheses are negative or atomic), one enters a focus stage, by choosing a hypothesis or the conclusion and *taking focus* on it. One then repeatedly decomposes the type under focus, which always leaves exactly one type under focus in each premise. This continues until the conclusion can be satisfied by a hypothesis, or until focus is lost, which happens when the type under focus has the wrong polarity for focus (*i.e.,* it is positive on the left, or negative on the right). At that point, one returns to inversion.

Certainly this proof search strategy is sound, since every rule is a valid sequent calculus rule. In its more general form, it is also complete: every true sequent has a focused derivation. (For our purposes—logical frameworks—it is preferable to restrict positive types even further using a lax modality [29].) But, even in its most general form, there are typically many fewer derivations, because at any point in the derivation, many fewer steps are permitted. To put this another way: a focused derivation has much more structure than an ordinary sequent calculus derivation (and a lax modality

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209145

[1] Strictly speaking, one must be in a linear (or other substructural) setting for negative and positive to align neatly with right-invertible and left-invertible. In a persistent setting, some types turn out to be invertible on the "wrong side" by accident.

gives it more still), and this turns out to make it suitable for use as a canonical form.

Suppose one is trying to derive $\Gamma \longrightarrow A^-$ in sequent calculus. One can do a little work on A^-, then a little work on a hypothesis in Γ, then on A^- some more, then on a different hypothesis in Γ, and so on. There are fewer choices than in natural deduction, but there remain many choices.

In a focused setting, one can prove $\Gamma \longrightarrow A^-$ in only one way. One must first apply invertible rules to A^- until no more apply, thereby obtaining an atomic proposition P^-. Then one must take focus on a hypothesis in Γ (or a constant) and work on that hypothesis exclusively until P^- is proved, or until focus is lost. If, additionally, we restrict positive types using a lax modality, one cannot lose left focus while deriving a negative goal, which eliminates the second possibility. This means that a proof term for P^- can have only one form: a head variable/constant with zero or more (negative) elims applied to it. We refer to such a proof term as a *path*.

A path corresponds precisely to an atomic object [12, 29] in a logical framework. When a path is then wrapped with zero or more negative intros, it is precisely a canonical form. Thus, the focused setting allows us to maintain a strong notion of canonical forms, which is essential for adequate representations [10], while also admitting positive types.

In a conventional setting, one writes a path like (for example): $(\pi_1\ (x\ V))\ W$. In focused logic, one separates the head from the rest of the path, which is called the *spine*. The spine is written in diagrammatic order, so the same path is written $x \cdot V; \pi_1; W$. Separating the path this way is important, because the spine can also be considered in isolation. A spine, not a full path, is the proof term for left focus.

Strong sums In logic there are (at least) two forms of dependent sum: the weak sum (a.k.a. existential type) and the strong sum [13].[2] The important difference between the weak and strong sum lies in their elimination form:

A weak sum (which we write using $\exists$) is eliminated by unpacking it for a particular scope, and the carrier (the sum's left-hand constituent) is not permitted to leak from that scope. This closed-scope elimination form makes the weak sum suitable for representing abstract types [18].

In contrast, a strong sum (which we write using Σ) is eliminated in the same open-scope fashion as a non-dependent product ($\&$ in a linear setting), using left and right projection operations. Crucially, the type of the right projection is permitted to refer to the left projection. Thus, if $M : \Sigma x{:}A.B$ then $\pi_2 M : [\pi_1 M/x]B$.

Both weak and strong sums have their uses. Weak sums are suitable for data abstraction. Also, they are suitable in a setting without dependent types, but that is not an issue here. On the other hand, they can be inconvenient. For example, consider an algebraic structure such as $\mathit{Semigroup} = \exists\alpha{:}\mathit{Type}.\Sigma m{:}(\alpha \to \alpha \to \alpha).\mathit{Associative}_m$. If G is a semigroup, one can only use G by unpacking it in a large enough scope to encompass all its uses. On the other hand, if $\exists$ is replaced by Σ, then one can simply obtain G's components by projection, whenever they are desired. (Data abstraction is compromised, but it can be restored with additional type theory [5].)

There are some additional merits to strong sums that are particular to logical frameworks: In the presence of a lax modality (which is needed to maintain strong canonical forms [28]), we will see that existential types cannot contribute directly to any term of negative type. If one wants a first-class dependent sum, one needs a strong sum. Strong sums are also useful in meta-logical frameworks that permit abstraction over types (such as Delphin [21] and Beluga [20]), since a theorem whose inputs rely on a single dependency can be instantiated with a strong sum to support multiple dependencies. Thus, for instance, a strong sum can play the role of a context (at least in the absence of linearity).

Using the language of polarity, we can characterize the two sums another way: The weak sum is invertible on the left (*i.e.,* eliminated using pattern matching), so it is a positive type. On the other hand, the strong sum seems to be focused on the left (*i.e.,* eliminated using projections that can appear in a path), so it should be a negative type.

In fact, the weak sum works quite well as a positive type in a focused logic [22, 24]. It poses no real difficulties beyond those that stem from dependent types generally.

Alas, the same is not true for the strong sum. The difficulty arises in the typing rule for spines (*i.e.,* the left focus judgement).

When S is a spine, we write $\Gamma \vdash S : A^- > U$ to mean that S decomposes a head of type A^- to derive a consequent U. (Think of a consequent as a goal of either polarity.) Thus we have rules like:

$$\frac{\Gamma \vdash V : A^+ \quad \Gamma \vdash S : B^- > U}{\Gamma \vdash V; S : (A^+ \to B^-) > U}$$

$$\frac{\Gamma \vdash S : A^- > U}{\Gamma \vdash \pi_1; S : (A^- \,\&\, B^-) > U} \qquad \frac{\Gamma \vdash S : B^- > U}{\Gamma \vdash \pi_2; S : (A^- \,\&\, B^-) > U}$$

For example, π_1 projects out the left constituent of a pair $A^- \,\&\, B^-$ and passes it on to the tail of the spine S, returning its result.

The left projection from a strong sum could work the same way:

$$\frac{\Gamma \vdash S : A^- > U}{\Gamma \vdash \pi_1; S : (\Sigma x{:}A^-.B^-) > U}$$

But, in the right projection, B^- could mention x and we have nothing to substitute for it:

$$\frac{\Gamma \vdash S : [???/x]B^- > U}{\Gamma \vdash \pi_2; S : (\Sigma x{:}A^-.B^-) > U}$$

If we knew the path R from which we are projecting, we would substitute $\pi_1 R$ for x. However, in a focused setting we must assign a type to a bare spine, without reference to any head variable or constant to which the spine might be attached.

At this point, it is natural to wonder whether we should abandon spines, and simply work with paths instead. This is how Concurrent LF was originally formulated [29], and strong sums have been shown to be workable in such a setting [4]. But there are good reasons to be dissatisfied with such a treatment: A system without left focus simply is not focused logic, and it sacrifices important advantages thereof.

One such advantage of focused logic (particularly relevant to the author's purposes) comes when the logic is combined with a metalogic capable of analyzing canonical forms. When analyzing a canonical form, one wants to begin with the most significant information—the head—and then proceed down the spine. This is what one gets in focused logic. In contrast, when analyzing paths one is forced to begin with the least important information—the end of the spine—and arrive at the head last of all. Also, for

[2] Howard referred to them as weak and strong existentials, but the term existential type has subsequently come to be identified [18] with the weak sum.

much the same reason, representing terms in spinal form results in a significant performance advantage in the implementation of a logical framework [16, 17]. Moreover, for one connective to drive the entire structure of the logic would seem to put the cart before the horse, and we show here that it is not necessary to do so.

Our approach In this paper, we show how to incorporate strong sums into focused logic, notwithstanding the difficulty discussed above. The key insight comes from a very different setting: the type theory of modules. In module calculi [7, 11, 14, 26], one can (in certain circumstances) rewrite the type of a module to refer the module itself. This process is whimsically referred to as *selfification* [7]. For example, when the module `M` has signature:

```
sig
  type t
  val x : t
  ...
end
```

it can be selfified to:

```
sig
  type t = M.t
  val x : M.t
  ...
end
```

Observe that after selfification, `x`'s type no longer depends on `t`.

The purpose of this is to circumvent a problem similar to our current problem. In the strong-sum typing rule, we cannot substitute the needed term for the dependency because the needed term is unavailable. In module calculi, one cannot (in general) substitute a module for a dependency because its type components are unavailable (since they may not fully determined at compile time).

Selfification allows one to write a module's signature in non-dependent form, which allows one to project fields from a module without needing to satisfy dependencies. We can do something similar here:

Suppose x has type $\Sigma y{:}A^-.B^-$. We can show that x has a more-specific, selfified type that we can think of as $A^- \,\&\, [\pi_1 x/y]B^-$.[3] We cannot give spine typing rules for Σ, but it is easy to give them for $\&$. Thus, when type-checking $x \cdot S$, we check the spine S against $(A^- \,\&\, [\pi_1 x/y]B^-) > U$, instead of $(\Sigma y{:}A^-.B^-) > U$. If S has the form $\pi_2; S'$, then S' is checked against $[\pi_1 x/y]B^- > U$.

In summary, we may not be able to resolve a strong sum's dependencies on-the-fly, but, using selfification, we can resolve them at the outset, when the head is still available. Unlike in module calculi, our selfification does not write in equality information (*e.g.,* `type t = M.t`), but we do not need that for our purposes.

In the remainder of this paper, we develop a focused logic that supports strong sums by using selfification. The issues of strong sums are largely orthogonal to linear logic, so for most of the paper we restrict our attention to intuitionistic persistent logic. To validate our logic, we prove cut admissibility and identity expansion. All the proofs are formalized in Coq.

[3] Precisely, x has the type $A^- \,\&\, [M/y]B^-$, where M is the identity expansion of $x \cdot \pi_1$.

neg. types	A^-	::=	$a \cdot S \mid \bigcirc A^+ \mid \Pi p{:}A^+.A^- \mid A^- \,\&\, A^- \mid \Sigma x{:}A^-.A^-$
pos. types	A^+	::=	$\downarrow A^- \mid 1 \mid A^+ \otimes A^+ \mid \exists p{:}A^+.A^+$
kinds	K	::=	$\mathsf{ty} \mid \Pi p{:}A^+.K$
contexts	Γ	::=	$\epsilon \mid \Gamma, x{:}A^-$
heads	h	::=	$x \mid c$
results	R	::=	$h \cdot S \mid \mathsf{ret}\ V$
values	V	::=	$\downarrow M \mid \star \mid V \otimes V$
spines	S	::=	$\mathsf{nil} \mid \mathsf{bind}\, p.R \mid V; S \mid \pi_1; S \mid \pi_2; S$
terms	M	::=	$\eta(R) \mid \mathsf{circ}\ R \mid \lambda p.M \mid \langle M, M \rangle$
patterns	p	::=	$\downarrow x \mid \star \mid p \otimes p$

Figure 1. Syntax

2 The logic

Our focused logic combines elements from Simmons [25] and Schack-Nielsen [22]. For now, and for most of the paper, we restrict our attention to persistent logic (*i.e.,* ordinary logic, which enjoys weakening, contraction, and exchange); we touch on what happens in the linear setting (not much) at the end. The syntax appears in Figure 1.

Focused notation can be verbose, so in some examples we use conventional, non-focused notation, for the sake of clarity. When we do so, we underline the expression to avoid confusion. For example, we might write $a \cdot \downarrow\eta(x \cdot \pi_1; \mathsf{nil}); \mathsf{nil}$ as $\underline{a\,(\pi_1 x)}$.

Types As usual, we categorize types as negative (A^-) or positive (A^+). Negative types include dependent function spaces ($\Pi p{:}A^+.B^-$), negative products ($A^- \,\&\, B^-$), and strong sums ($\Sigma x{:}A^-.B^-$). Positive types include positive unit (1), positive products ($A^+ \otimes B^+$), and weak sums ($\exists p{:}A^+.B^+$).

Although we are working in persistent logic, we use notation from linear logic to distinguish between positive and negative products, and between positive and negative unit. (The latter would be written $\top$, if we supported it.)

It is convenient to view the non-dependent function space $A^+ \to B^-$ as merely a degenerate form of the dependent function space. On the other hand, we view negative and positive products as distinct connectives from strong and weak sums, rather than as degenerate forms.

Negative types are injected into the positive types by the down-shift connective ($\downarrow A^-$), and positive types are injected into the negatives types—for limited uses—by the monad connective ($\bigcirc A^+$). We do not support an unrestricted upshift ($\uparrow A^+$) because it is incompatible with strong sums (as well as for other reasons).

Finally, negative types also include atomic propositions in the form of a path $a \cdot S$, where a is a type constant and S is a spine to which a is applied. Each type constant is given a kind K, which is either ty (the path is a type) or a dependent function space $\Pi p{:}A^+.K$.

Omitted connectives Our system omits a number of types that sometimes appear in focused logic. We omit upshift primarily because, as we show below, it is incompatible with strong sums. However, if not for that reason, we would probably still omit it because it spoils the canonical forms on which logical frameworks rely [28].

We also omit par ($\parr$) and negative void ($\bot$) because we are working in an intuitionistic setting.

We omit other common types for less fundamental reasons. We omit disjoint sums ($\oplus$) and positive void (0) because they would necessitate a multi-clause pattern-matching syntax, which in turn would require judgements expressing exhaustiveness and non-redundancy. We leave them out for the sake of simplicity. We omit positive atomic propositions because omitting them simplifies cut, and because we see no vital use for them in logical frameworks.

Finally, we omit top (a.k.a. negative unit) because its absence considerably simplifies one key lemma (Lemma 10). In a persistent setting this is no loss, because one can simply define a trivial atomic proposition. In a linear setting it is a loss, but since top causes severe practical problems for a linear logic framework [22], it is a loss we would suffer anyway. The Celf system [23] ameliorates the lack of top by supporting affine hypotheses and an affine modality [22].

Proof terms Our proof terms are broken into five syntactic classes. Terms (M) are the most general; they are the proof terms for right (negative) inversion. Of particular interest to this work is $\langle M, N\rangle$, which is the introduction form for negative conjunction and for strong sums. Patterns (p) are the proof term for left (positive) inversion; they arise whenever a positive type come into scope (such as in a lambda abstraction).

Results (R) are the proof term for when all available inversions have been performed, but a focus has not been selected. A result takes the form of a path ($h \cdot S$) or a return of a value. A path's type may be negative or positive, but a return's type is always positive. The head of a path (h) is either a variable (x) or a term constant (c). Note that, following Simmons [25], we distinguish between a result R and its inclusion as a term, which is either $\eta(R)$ (if R's type is negative) or $\mathsf{circ}\ R$ (if R's type is positive).

Values (V) are the proof term for right (positive) focus. We write the introduction form for positive products $V \otimes W$, and the introduction form for positive unit $\star$. The introduction form for $\downarrow A^-$ is written $\downarrow M$.

Spines (S) are the proof term for left (negative) focus. A spine is made of applications and projections, terminated by either nil or $\mathsf{bind}\, p.R$. In the latter case, the type under focus when the end is reached is $\bigcirc A^+$. Then a value of type A^+ is pattern-matched against p, focus is lost, and a new result is computed using the variables bound by p. For example, the result $h \cdot \pi_1; \mathsf{bind}\, p.R$ would be written in non-focused notation as $\underline{\mathsf{bind}\, p = \pi_1 h\ \mathsf{in}\ R}$.

A restricted form of spines (using application and nil only) are also used to form atomic propositions ($a \cdot S$).

2.1 Semantics

The logic's typing rules are made up of six judgements for proof terms, plus four more for types and kinds. We focus on the former here:

right inversion	$\Gamma \vdash M : A^-$
left inversion	$\vdash p : A^+ \Rightarrow \Gamma$
choose focus	$\Gamma \vdash R : U$
right focus	$\Gamma \vdash V : A^+$
left focus	$\Gamma \vdash S : A^- > U$
head typing	$\Gamma \vdash h : A^-$

The right inversion, right focus, and head typing judgements are self-explanatory. The left inversion judgement $\vdash p : A^+ \Rightarrow \Gamma$ means that when the pattern p is matched against a value of type A^+, variables become bound with the negative types given by Γ. The left focus judgement $\Gamma \vdash S : A^- > U$ means S can operate on a head belonging to A^-, producing a result belonging to U.

A result's type and a spine's codomain can be either negative or positive. To accommodate that, we have an additional syntactic class of *consequents* (U):

$$U \ ::= \ A^-\ \mathsf{tr}^- \mid A^+\ \mathsf{lax}^+$$

If $R : U$, the former indicates that R is a proof term for the truth of a negative proposition, and the latter that R is a proof term for the *lax* truth of a positive proposition.

A lax proof term can be used only to establish another lax judgement [19]. In our setting, the ultimate effect of this is negative proof terms cannot depend on variables introduced by $\mathsf{bind}\, p.R$. However, a negative proof term *can* depend on variables introduced by a lambda abstraction, because lambda-bound variables never pass through a lax judgement.

The full rules are given in Appendix A. They take as fixed a *signature* Ξ, which assigns negative types to term constants and kinds to type constants. In the results that follow, we implicitly assume that Ξ is well-formed, meaning that every K or A^- in Ξ's range is well-formed.

Of particular interest to our purposes are the typing rules for results and spines, all but one of which we give in Figure 2. (The remaining rule is central to our treatment of strong sums, and we will discuss it in the next section.) Observe that several of the rules are polymorphic over polarity; their consequent is arbitrary, so they can produce a negative or positive result. The ret and nil rules cannot be polymorphic this way, as they fundamentally work with positive and negative objects.

On the other hand, the bind rule could conceivably be polymorphic, if we replaced $B^+\ \mathsf{lax}^+$ with U. The fact that it is not polymorphic is how laxity (internalized by $\bigcirc$) is enforced. If it *were* polymorphic, we would want to rename $A^+\ \mathsf{lax}^+$ to $A^+\ \mathsf{tr}^+$ and $\bigcirc A^+$ to $\uparrow A^+$. We do not do so for two reasons: relaxing the laxity of positive types would break the canonical forms required for a logical framework [28], and—even more importantly for our purposes—it would break strong sums.

2.2 Substitution and cut

The logic's typing rules rely on substitution and cut in several places, including the typing rule for application given in Figure 2. In it, the spine $(V; S)$ accepts $\Pi p{:}A^+.B^-$, and $[V/p]B^+$ is passed on to the tail S. Here, $[V/p]B^+$ is a positive cut, in which the argument V is matched against the pattern p, and the resulting bindings are substituted into B^+.

Substitution and cut are defined by four interdependent definitions:

hereditary substitution	$[M/x]E = E'$
negative cut	$M \bullet S = R$
positive cut	$[V/p]E = E'$
commuting substitution	$[E \setminus p]R = E'$

$\boxed{\Gamma \vdash R : U}$

$$\frac{\Gamma \vdash h : A^- \quad \Gamma \vdash S : A^- > U}{\Gamma \vdash h \cdot S : U} \qquad \frac{\Gamma \vdash V : A^+}{\Gamma \vdash \mathsf{ret}\, V : A^+\ \mathsf{lax}^+}$$

$\boxed{\Gamma \vdash S : A^- > U}$

$$\frac{}{\Gamma \vdash \mathsf{nil} : A^- > A^-\ \mathsf{tr}^-} \qquad \frac{\vdash p : A^+ \Rightarrow \Gamma' \quad \Gamma, \Gamma' \vdash R : B^+\ \mathsf{lax}^+}{\Gamma \vdash \mathsf{bind}\, p.R : \bigcirc A^+ > B^+\ \mathsf{lax}^+}$$

$$\frac{\Gamma \vdash V : A^+ \quad \Gamma \vdash S : [V/p]B^- > U}{\Gamma \vdash V; S : \Pi p{:}A^+.B^- > U} \qquad \frac{\Gamma \vdash S : A^- > U}{\Gamma \vdash \pi_1; S : A^- \,\&\, B^- > U} \qquad \frac{\Gamma \vdash S : B^- > U}{\Gamma \vdash \pi_2; S : A^- \,\&\, B^- > U}$$

Figure 2. Result and spine typing

In each of these definitions, an expression (E) refers to any syntactic class other than a head or a pattern:

$$E \ ::= \ A^- \mid A^+ \mid K \mid \Gamma \mid R \mid V \mid S \mid M$$

A substitution or cut may be undefined. For example, a required cut may not exist, due to a type error. However, we will show (Theorem 14) that substitutions and cuts always exist for appropriately typed expressions.

Hereditary substitution [28], which corresponds to the right commutative cuts in cut elimination, is the most familiar, in that it largely resembles conventional substitution. It differs only in the case for substitution into a path:

$$\begin{array}{lcll} [M/x](x \cdot S) & = & M \bullet [M/x]S & \\ [M/x](h \cdot S) & = & h \cdot [M/x]S & (h \neq x) \end{array}$$

When substituting for the head variable of a path, conventional substitution would leave a term at the head, which is syntactically prohibited. Instead, we cut the term into the (substituted) spine to obtain a new result. This, in turn, may reinvoke substitution, which is why it is called hereditary.

Positive cut matches a value against a pattern, and carries out all the substitutions that implies:

$$\begin{array}{lcl} [\downarrow M/\downarrow x]E & = & [M/x]E \\ [\star/\star]E & = & E \\ [V \otimes W/p \otimes q]E & = & [W/q][V/p]E \end{array}$$

For technical reasons, it is important that the substitutions be carried out one-at-a-time, and not simultaneously.[4]

Negative cut operates on a term using a spine, thereby producing a result:

$$\begin{array}{lcl} \eta(R) \bullet \mathsf{nil} & = & R \\ \mathsf{circ}\, R_1 \bullet \mathsf{bind}\, p.R_2 & = & [R_1 \setminus p]R_2 \\ (\lambda p.M) \bullet V; S & = & [V/p]M \bullet S \\ \langle M, N \rangle \bullet \pi_1; S & = & M \bullet S \\ \langle M, N \rangle \bullet \pi_2; S & = & N \bullet S \end{array}$$

Finally, *commuting substitution*[5] corresponds to the left commutative cuts in cut elimination. In it, the substitutend (which must be a result or a spine) is "substituted" for a pattern in a result, to obtain the same syntactic class as the substitutend:

$$\begin{array}{lcl} [h \cdot S \setminus p]R & = & h \cdot [S \setminus p]R \\ [\mathsf{ret}\, V \setminus p]R & = & [V/p]R \\ [\mathsf{bind}\, p_1.R_1 \setminus p_2]R_2 & = & \mathsf{bind}\, p_1.[R_1 \setminus p_2]R_2 \\ [V; S \setminus p]R & = & V; [S \setminus p]R \\ [\pi_1; S \setminus p]R & = & \pi_1; [S \setminus p]R \\ [\pi_2; S \setminus p]R & = & \pi_2; [S \setminus p]R \end{array}$$

Note that the substitutend must ultimately end in a ret. This process can be understood as applying commuting conversions until one uncovers the ret, thereby creating the opportunity for a positive cut. For example, observe that $[h \cdot \pi_1; \mathsf{bind}\, p.R_1 \setminus q]R_2 = h \cdot \pi_1; \mathsf{bind}\, p.[R_1 \setminus q]R_2$. If we write this in non-focused notation, and write $[R \setminus q]R'$ as $\mathsf{let}\, q = R \,\mathsf{in}\, R'$, this equation is the commuting conversion:

$$\frac{\mathsf{let}\, q = (\mathsf{bind}\, p = \pi_1 h \,\mathsf{in}\, R_1) \,\mathsf{in}\, R_2}{=}$$
$$\mathsf{bind}\, p = \pi_1 h \,\mathsf{in}\, \mathsf{let}\, q = R_1 \,\mathsf{in}\, R_2$$

2.3 Suspension-normal consequents

Results can appear within terms either as $\eta(R)$ (for negative results) or as $\mathsf{circ}\, R$ (for positive results):

$$\frac{\Gamma \vdash R : (a \cdot S)\ \mathsf{tr}^-}{\Gamma \vdash \eta(R) : a \cdot S} \qquad \frac{\Gamma \vdash R : A^+\ \mathsf{lax}^+}{\Gamma \vdash \mathsf{circ}\, R : \bigcirc A^+}$$

The η rule corresponds to the atomic initial sequent in sequent calculus. In terms of proof terms, it means that proof terms for negative types must be given in eta-long (*i.e.,* fully applied) form. This makes them suitable for use in logical frameworks [10].

Reading from conclusions backward to premises, these are the only rules that bring consequents (U) into play. Thus, one might suppose that we need only consider consequents of the form $(a \cdot S)\ \mathsf{tr}^-$ and $A^+\ \mathsf{lax}^+$. However, as Simmons observes [25], it is very useful to be able to give types to incomplete results, even if incomplete results cannot appear within terms. Thus, most of our theorems involving consequents permit them to have the general form.

However, a few theorems apply only to the restricted form—cut admissibility being the most important. Suppose $M : A^-$ and recall that $\mathsf{nil} : A^- > A^-\ \mathsf{tr}^-$. The cut $M \bullet \mathsf{nil}$ is defined only if M has the form $\eta(R)$, which happens only when A^- is an atomic proposition.

[4] It makes no difference for well-formed expressions, but for ill-formed expressions, a sequential substitution can be well-defined when a simultaneous one would not be. For example:

$$[\downarrow(\lambda z.\eta(c \cdot \mathsf{nil})) \otimes \downarrow\eta(c \cdot \mathsf{nil}) \,/\, \downarrow x \otimes \downarrow y](x \cdot \downarrow\eta(y \cdot \pi_1; \mathsf{nil}); \mathsf{nil})$$

From this, one can obtain a counterexample to cut admissibility as it is stated in Theorem 14. To prove cut admissibility with simultaneous substitution would require stronger well-formedness conditions than it is convenient to maintain.

[5] A more common name is *leftist substitution*, so called because it is defined by induction on the syntax of the substitutend, rather than on the syntax of the expression being substituted into. But that terminology relies on the notation (it only makes sense when substitution is written prefix). We prefer a more descriptive term.

We say that a consequent is *suspension normal*[6] if it has the form $(a \cdot S)\,\mathsf{tr}^-$ or $A^+\,\mathsf{lax}^+$. We write $\bar{U}$ to range over suspension-normal consequents.

2.4 Expansion

Another important notion in the logic is expansion; its definition appears in Figure 3. Negative expansion ($\mathsf{Exp}(R : A^-)$) takes a result R belonging to $A^-\,\mathsf{tr}^-$ and eta-expands it into a term of type A^-. (In the base case, when A^- is atomic, it uses the primitive η form.) Positive expansion $\mathsf{Exp}(A^+)$ produces a pattern and value $p.V$, such that when a value is matched against p and the resulting substitution is applied to V, the original value is reconstructed.

Note that expansion relies only on the structure of the type, not on its dependencies. Therefore it is not necessary to maintain well-formedness of the type throughout the definition. For example, we define $\mathsf{Exp}(R : \Sigma x{:}A^-.B^-)$ to be the same as $\mathsf{Exp}(R : A^- \,\&\, B^-)$, even though this means that B^- may contain unbound variables.

Expansion utilizes the auxiliary notion of composition (written $R \circ S$ or $S \circ S'$), which appends a spine to the end of a path or another spine.

3 Strong sums

The introduction rule for strong sums is unproblematic:

$$\frac{\Gamma \vdash M : A^- \quad \Gamma \vdash N : [M/x]B^-}{\Gamma \vdash \langle M, N\rangle : \Sigma x{:}A^-.B^-}$$

We should observe, however, that the term "right inversion" is no longer wholly appropriate once this rule is added. If proof terms are omitted, this is the rule:

$$\frac{\Gamma \longrightarrow A^- \quad \Gamma \longrightarrow [M/x]B^-}{\Gamma \longrightarrow \Sigma x{:}A^-.B^-}$$

which is not invertible. Nevertheless, we retain the term "right inversion" to maintain the connection to the roots of focused logic.

The problem lies in the elimination rules. As we have seen, we could produce a left-projection rule, but in the right-projection rule we have nothing to substitute for the dependent variable:

$$\frac{\Gamma \vdash S : [???/x]B^- > U}{\Gamma \vdash \pi_2; S : (\Sigma x{:}A^-.B^-) > U}$$

Instead, we arrange things so that a spine never needs to see a strong sum in the first place. Suppose M has type A^-. We define the *selfification* of M at A^- as follows:

$$\begin{array}{lcl}
\mathsf{Self}(M : a \cdot S) & = & a \cdot S \\
\mathsf{Self}(M : \bigcirc A^+) & = & \bigcirc A^+ \\
\mathsf{Self}(\lambda p.M : \Pi p{:}A^+.B^-) & = & \Pi p{:}A^+.\mathsf{Self}(M : B^-) \\
\mathsf{Self}(\langle M, N\rangle : A \,\&\, B) & = & \mathsf{Self}(M : A) \,\&\, \mathsf{Self}(N : B) \\
\mathsf{Self}(\langle M, N\rangle : \Sigma x{:}A.B) & = & \mathsf{Self}(M : A) \,\&\, \\
 & & \mathsf{Self}(N : [M/x]B)
\end{array}$$

This has two important properties. First, $\mathsf{Self}(M : A^-)$ contains M:

Lemma 1. *Suppose* $\mathsf{Self}(M : A^-)$ *is defined. Then* $\Gamma \vdash M : A^-$ *if and only if* $\Gamma \vdash M : \mathsf{Self}(M : A^-)$.

[6] This terminology is justified in Simmons [25].

Second, $\mathsf{Self}(M : A^-)$ turns strong sums into non-dependent negative products. Selfification resolves the dependent variable in every strong sum with the actual term it represents. Any residual strong sums in a selfified type lie within positive types, where they cannot be reached without losing focus, so they pose no issue for spine typing.

Our strategy for typing a path $h \cdot S$ is to selfify h's type, and to use the strong-sum-free selfified type as the domain when typing S. Since the Self operation requires a term, rather than a head, we apply h to nil to obtain a result, and then we expand that to its equivalent term. This gives rise to the key rule that makes strong sums work, the selfifying rule for typing paths:

$$\frac{\Gamma \vdash h : A^- \quad \Gamma \vdash S : \mathsf{Self}(\mathsf{Exp}(h \cdot \mathsf{nil} : A^-) : A^-) > U}{\Gamma \vdash h \cdot S : U}$$

For example, suppose $a : \mathsf{ty}$ and $b : \underline{a \to \mathsf{ty}}$ and $C^- = \underline{\Sigma y{:}a.by}$ and $x : C^-$. Then the expansion $\mathsf{Exp}(x \cdot \underline{\mathsf{nil} : C^-})$ is $\langle \eta(x \cdot \pi_1; \mathsf{nil}), \underline{\eta(x \cdot \pi_2; \mathsf{nil})}\rangle$, and the selfification $\mathsf{Self}(\mathsf{Exp}(x \cdot \mathsf{nil} : C^-) : C^-)$ works out to $\underline{a \,\&\, b(\pi_1 x)}$. Certainly $(\pi_2; \mathsf{nil})$ takes $\underline{a \,\&\, b(\pi_1 x)}$ to $\underline{b(\pi_1 x)}$, so $(x \cdot \pi_2; \mathsf{nil})$ has type $\underline{b(\pi_1 x)}$, as one would expect.

It turns out that we also retain the old, non-selfifying rule for typing paths, for an important technical reason. In the downshift case of identity expansion (Theorem 15), to invoke the induction hypothesis we need to use the fact that $\Gamma, x{:}A^- \vdash x \cdot \mathsf{nil} : A^-\,\mathsf{tr}^-$. To show that fact using the selfification rule, we would need to know already that $\mathsf{Exp}(x \cdot \mathsf{nil} : A^-)$ is well-typed, but that is exactly what we are invoking the induction hypothesis to obtain. By retaining the non-selfifying rule, we can obtain that fact directly. Fortunately, having two rules for typing paths causes no problems.

3.1 The problem with upshift

For this strategy to work, we have to give up the upshift connective in favor of the weaker monad connective. For logical frameworks this is no loss, because one already must sacrifice upshift to maintain strong canonical forms [28], but for other applications it may be disappointing.

Suppose we had an upshift connective with the elimination rule:

$$\frac{\vdash p : A^+ \Rightarrow \Gamma' \quad \Gamma, \Gamma' \vdash R : U}{\Gamma \vdash \mathsf{bind}\, p.R : \uparrow A^+ > U}$$

(Observe that the $B^+\,\mathsf{lax}^+$ from the monad elimination rule is replaced here by the general U.) Also suppose that we define selfification for upshift the same way as for the monad (*i.e.,* $\mathsf{Self}(M : \uparrow A^+) = \uparrow A^+$).

Suppose $x : A^-$, where $A^- = \uparrow\downarrow(B^- \,\&\, C^-)$, and let $R_i = x \cdot \mathsf{bind}\downarrow y.(y \cdot \pi_i; \mathsf{nil})$ (that is, $\underline{\mathsf{bind}\downarrow y = x \;\mathsf{in}\; \pi_i y}$). It follows that R_1 has type B^- and R_2 has type $\underline{C^-}$.

However, suppose we replace the non-dependent product with a strong sum, so $A^- = \uparrow\downarrow(\Sigma z{:}B^-.D^-)$. Selfification does not reach into the upshift, so $\mathsf{Self}(\mathsf{Exp}(x \cdot \mathsf{nil} : A^-) : A^-) = A^-$. Consequently, the inner spine $(\pi_i; \mathsf{nil})$ sees a strong sum, so we cannot give a type to either R_1 or R_2. By the same token we cannot give a type to $\underline{\mathsf{bind}\downarrow y = x \;\mathsf{in}\; \langle \pi_1 y, \pi_2 y\rangle}$, which combines R_1 and R_2. This is fatal, because the latter is the identity expansion of x, so identity expansion fails to hold.

The problem is not merely an incompleteness in the rules. Yes, in principle R_1 could have type B^-, and—as usual—we could obtain that typing by adding a left projection rule for strong sums. But

$\boxed{\mathsf{Exp}(R : A^-) = M}$ (Negative expansion)

$$\begin{array}{lcll}
\mathsf{Exp}(R : a \cdot S) & = & \eta(R) & \\
\mathsf{Exp}(R : \bigcirc A^+) & = & \mathsf{circ}\,(R \circ \mathsf{bind}\,p.\,\mathsf{ret}\,V) & (\text{where } \mathsf{Exp}(A^+) = p.V) \\
\mathsf{Exp}(R : \Pi p'{:}A^+.B^-) & = & \lambda p.\,\mathsf{Exp}(R \circ (V; \mathsf{nil}) : B) & (\text{where } \mathsf{Exp}(A^+) = p.V) \\
\mathsf{Exp}(R : A^- \,\&\, B^-) & = & \langle \mathsf{Exp}(R \circ \pi_1; \mathsf{nil} : A^-), \mathsf{Exp}(R \circ \pi_2; \mathsf{nil} : B^-)\rangle & \\
\mathsf{Exp}(R : \Sigma x{:}A^-.B^-) & = & \mathsf{Exp}(R : A^- \,\&\, B^-) &
\end{array}$$

$\boxed{\mathsf{Exp}(A^+) = p.V}$ (Positive expansion)

$$\begin{array}{lcl}
\mathsf{Exp}(\downarrow A^-) & = & \downarrow x.\downarrow \mathsf{Exp}(x \cdot \mathsf{nil} : A^-) \\
\mathsf{Exp}(1) & = & \star.\star \\
\mathsf{Exp}(A^+ \otimes B^+) & = & p \otimes q.\, V \otimes W \quad (\text{where } \mathsf{Exp}(A^+) = p.V \text{ and } \mathsf{Exp}(B^+) = q.W \text{ and } BV(p) \cap BV(q) = \emptyset) \\
\mathsf{Exp}(\exists p'{:}A^+.B^+) & = & \mathsf{Exp}(A^+ \otimes B^+)
\end{array}$$

$\boxed{R \circ S = R' \qquad S \circ S' = S''}$ (Composition)

$$\begin{array}{llllllll}
h \cdot S \circ S' & = & h \cdot (S \circ S') & \mathsf{nil} \circ S & = & S & V; S \circ S' & = & V; (S \circ S') \\
\mathsf{ret}\,V \circ S & & \text{undefined} & \mathsf{bind}\,p.r \circ S & = & \mathsf{bind}\,p.(r \circ S) & \pi_1; S \circ S' & = & \pi_1; (S \circ S') \\
 & & & & & & \pi_2; S \circ S' & = & \pi_2; (S \circ S')
\end{array}$$

Figure 3. Expansion and composition

for R_2 there is nothing to write! We can say that $\underline{\pi_2 y}$ has type $[\underline{\pi_1 y}/z]D^-$, but the full expression R_2 is outside the scope of y, so it cannot have type $[\underline{\pi_1 y}/z]D^-$.

There is another way to look at the problem, though it ultimately amounts to the same thing: We might wish to change the definition of selfification for upshift to do something more useful. For instance, we might say something like $\mathsf{Self}(\eta(R) : \uparrow A^+) = \uparrow \mathsf{Self}(V : A^+)$. (Indeed, we *could* define $\mathsf{Self}(V : A^+)$, were it any use to us.) But what is the V? We could certainly obtain one if R had the form $\mathsf{ret}\,V$, but R could also be a path, and indeed that is the case of interest to us. If R were a path we would be stuck, because the only way to obtain the components of a positive path is to pattern-match on it, but any variables thus bound would be out of scope to us.

In retrospect, it seems significant that in the module type theories that inspired this work (particularly Stone and Harper [26]), all the connectives are negative. This work helps explain why a workable module type theory supporting both selfification and existential signatures (which would be useful for circumventing the "avoidance problem") was elusive [7].

4 Validation

We validate our logic by proving cut admissibility (Theorem 14) and identity expansion (Theorem 15). We summarize the proof here. All the proofs are formalized in Coq (Section 5).

The most delicate aspect of the proof comes from the interdependence of cut admissibility and identity expansion. Without dependent types, these proofs are typically separate. With dependent types, identity expansion depends on cut admissibility, because in the function case one must show that a substitution into the function's codomain is defined. With strong sums, cut admissibility also depends on identity expansion, since the selfification rule for path typing employs expansion. Thus, both cut admissibility and identity expansion need to be proven before the other.

We cut this knot by first proving that cut admissibility and identity expansion hold *up to simple types*. Each of those gives enough to prove the full version of the other. Thus we begin by defining a weak version of the static semantics, which ignores dependencies and uses only the structure of types.

Definition 2. *Two types, kinds, or consequents are* similar *(written $A^- \sim B^-$,* etc.*) if they differ only in dependencies and patterns. We also say that $A^- \,\&\, B^-$ is similar to $\Sigma x{:}A^-.B^-$ and $A^+ \otimes B^+$ is similar to $\exists p{:}A^+.B^+$.*

Definition 3. *We write judgements that hold modulo dependencies using $\Vdash$. The key rules are:*

$$\frac{\Gamma \vdash h : A^- \quad A^- \sim B^- \quad \Gamma \Vdash S : B^- > U}{\Gamma \Vdash h \cdot S : U}$$

for paths, and a similar rule for type paths. These allow dependencies to be altered as desired. The rules also omit all substitutions.

It will also prove to be convenient to have a notation for typing expressions of arbitrary sort:

Definition 4. *A generalized consequent J is given by the grammar:*

$$\begin{array}{rcl}
J & ::= & \bar{U} \mid A^+ \mid A^- > \bar{U} \mid A^- \\
 & \mid & \mathsf{type}^- \mid \mathsf{type}^+ \mid \mathsf{kind} \mid K > K' \mid \mathsf{context}
\end{array}$$

We define $\Gamma \vdash E : J$ and $\Gamma \Vdash E : J$ and substitution/cut into J in the obvious manner.

Now we can sketch the proof. We begin by establishing that substitutions commute:

Lemma 5. *Suppose $x \neq y$ and $x \notin BV(p)$ and $BV(p) \cap BV(q) = \emptyset$. Then each of the following equations hold, provided the inner cuts and/or substitutions are defined:*

- $[M/x](N \bullet S) = [M/x]N \bullet [M/x]S$
- $[M/x][V/p]E = [[M/x]V/p][M/x]E$
- $[M/x][N/y]E = [[M/x]N/y][M/x]E$
- $[M/x][E \setminus p]R = [[M/x]E \setminus p][M/x]R$
- $[V/q][E \setminus p]R = [[V/q]E \setminus p][V/q]R$
- $M \bullet [S \setminus p]R = [M \bullet S \setminus p]R$
- $[E \setminus p][R \setminus q]R' = [[E \setminus p]R \setminus q]R'$
- $[V/p][W/q]E = [[V/p]W/q][V/p]E$

Proof sketch. We define an auxiliary version of cut/substitution in which a derivation consumes one unit of fuel each time it substitutes for a path's head variable. It is easy to show that any cut/substitution can be given enough fuel to complete. We then show, by induction on the amount of fuel provided, that each of the equations holds, and moreover that the outer cuts/substitutions can complete using the same fuel as the inner ones. □

We also need to know that substitution commutes with selfification:

Lemma 6. *If* $\mathsf{Self}(M : A^-)$ *and* $[N/x]M$ *and* $[N/x]A^-$ *are defined, then* $[N/x]\mathsf{Self}(M : A^-) = \mathsf{Self}([N/x]M : [N/x]A^-)$.

Next we need to show that negative expansions are equivalent (in an appropriate sense) to the result being expanded, and that positive expansions are equivalent to the identity. For example, to show identity expansion for functions, we need to know that if $\mathsf{Exp}(A^+) = p.V$ then $[V/p]E = E$. To show these, we need to make some auxiliary definitions, and to strengthen the induction hypothesis considerably.

Definition 7. *A spine is* negative *if it ends in* nil *(not* bind*). A result is* negative *if it has the form* $h \cdot S$ *(not* $\mathsf{ret}\ V$*) and* S *is negative.*

Lemma 8. *If* $\Gamma \vdash S : A^- > B^-\ \mathsf{tr}^-$ *then* S *is negative. If* $\Gamma \vdash R : A^-\ \mathsf{tr}^-$ *then* R *is negative.*

Definition 9. *A* partial cut $M ;; S$ *is defined as follows:*

$$\begin{array}{lcl} M ;; \mathsf{nil} & = & M \\ (\lambda p.M) ;; V; S & = & [V/p]M ;; S \\ \langle M, N\rangle ;; \pi_1; S & = & M ;; S \\ \langle M, N\rangle ;; \pi_2; S & = & N ;; S \end{array}$$

Now we can state the expansion lemma:

Lemma 10 (Expansion substitution)**.**

- *If* $\mathsf{Exp}(A^+) = p.V$ *and* $[V/p]E$ *is defined, then* $[V/p]E = E$.
- *If* R *is negative and* $\mathsf{Exp}(R : A^-) \bullet S$ *is defined, then* $\mathsf{Exp}(R : A^-) \bullet S = R \circ S$.
- *If* $[\mathsf{Exp}(x \cdot \mathsf{nil} : A^-)/x]E$ *is defined, then* $[\mathsf{Exp}(x \cdot \mathsf{nil} : A^-)/x]E = E$.
- *If* $\mathsf{Exp}(A^+) = p.V$ *and* $[(R \circ \mathsf{bind}\, p.\mathsf{ret}\, V) \setminus q]R'$ *is defined, then* $[(R \circ \mathsf{bind}\, p.\mathsf{ret}\, V) \setminus q]R' = R \circ \mathsf{bind}\, q.R'$.
- *If* S *is negative and* $[M/x]\mathsf{Exp}(x \cdot S : A^-)$ *and* $[M/x]S$ *are defined, then* $[M/x]\mathsf{Exp}(x \cdot S : A^-) = M ;; [M/x]S$.
- *If* $\mathsf{Exp}(A^+) = p.V$ *and* $[W/p]V$ *is defined, then* $[W/p]V = W$.

Proof sketch. By induction on the size of A^+ or A^-. □

As remarked earlier, this lemma is false as stated if the logic includes negative unit ($\top$). The problem with negative unit is its expansion rule $\mathsf{Exp}(R : \top) = \langle\rangle$ throws away R entirely. This poses problems if R is ill-formed. For example, suppose $x : \top$. Then $[\langle\rangle/x]\mathsf{Exp}(x \cdot \pi_1; \mathsf{nil} : \top) = [\langle\rangle/x]\langle\rangle = \langle\rangle$ and $[\langle\rangle/x](\pi_1; \mathsf{nil}) = \pi_1; \mathsf{nil}$, but $\langle\rangle ;; \pi_1; \mathsf{nil}$ is undefined. This contradicts the fifth clause of the lemma.

This is fixable by adding well-formedness conditions to the lemma, but that would require cut admissibility to maintain stronger well-formedness conditions than it is convenient to maintain.

Next up are cut admissibility and identity expansion, modulo dependencies:

Lemma 11 (Weak cut admissibility)**.**

- *If* $\Gamma \Vdash M : A^-$ *and* $\Gamma, x{:}A^- \Vdash S : A^- > \bar{U}$*, then* $\Gamma \Vdash M \bullet S : \bar{U}$.
- *If* $\Gamma_1 \Vdash V : A^+$ *and* $\Vdash p : A^+ \Rightarrow \Gamma_2$ *and* $\Gamma_1, \Gamma_2, \Gamma_3 \Vdash E : J$*, then* $\Gamma_1, \Gamma_3 \Vdash [V/p]E : J$.
- *If* $\Gamma_1 \Vdash M : A^-$ *and* $\Gamma_1, x{:}A^-, \Gamma_2 \Vdash E : J$ *then* $\Gamma_1, \Gamma_2 \Vdash [M/x]E : J$.
- *If* $\Gamma \Vdash R : \mathsf{lax}^+A^+$ *and* $\Vdash p : A^+ \Rightarrow \Gamma'$ *and* $\Gamma, \Gamma' \vdash R' : B^+\ \mathsf{lax}^+$*, then* $\Gamma \Vdash [R \setminus p]R' : B^+\ \mathsf{lax}^+$.
- *If* $\Gamma \Vdash S : C > \mathsf{lax}^+A^+$ *and* $\Vdash p : A^+ \Rightarrow \Gamma'$ *and* $\Gamma, \Gamma' \vdash R : B^+\ \mathsf{lax}^+$*, then* $\Gamma \Vdash [S \setminus p]R : C > B^+\ \mathsf{lax}^+$.

Proof sketch. By induction on the size of A^- or A^+, with an inner induction on one of the derivations. □

Lemma 12 (Weak identity expansion)**.**

- *If* $\Gamma \Vdash R : \mathsf{tr}^-A^-$ *then* $\Gamma \Vdash \mathsf{Exp}(R : A^-) : A^-$.
- *If* $\mathsf{Exp}(A^+) = p.V$ *and* $\Vdash p : A^+ \Rightarrow \Gamma'$*, then* $\Gamma, \Gamma' \Vdash V : A^+$.

Proof sketch. By induction on the size of A^- or A^+. □

Now we can show the main selfification lemma:

Lemma 13. *If* $\Vdash \Gamma : \mathsf{context}$ *and* $\Gamma \vdash R : \mathsf{tr}^-A^-$*, then* $\Gamma \vdash R : \mathsf{Self}(\mathsf{Exp}(R : A^-) : A^-)\,\mathsf{tr}^-$ *and* $\Gamma \Vdash \mathsf{Self}(\mathsf{Exp}(R : A^-) : A^-)\,\mathsf{type}^-$.

Proof sketch. The proof combines two auxiliary selfification results:

- If $\Gamma \Vdash M : A^-$ and $\Gamma \Vdash A^- : \mathsf{type}^-$, then $\Gamma \Vdash \mathsf{Self}(M : A^-) : \mathsf{type}^-$.
- If $\Vdash \Gamma : \mathsf{context}$ and $\Gamma \vdash R : \mathsf{Self}(\mathsf{Exp}(R : A^-) : A^-)\,\mathsf{tr}^-$ and $\Gamma \vdash S : A^- > B^-\ \mathsf{tr}^-$, then $\Gamma \vdash S : \mathsf{Self}(\mathsf{Exp}(R : A^-) : A^-) > \mathsf{Self}(\mathsf{Exp}(R \circ S : B^-) : B^-)\,\mathsf{tr}^-$.

The former is proven by induction on the size of A^-, the latter by induction on the typing derivation of S. □

Whenever R's type is a strong sum, this lemma allows us to replace that strong sum with a negative product, which allows us to make progress in the identity expansion case for strong sums. Unfortunately, we cannot show that the new type is well-formed until we have identity expansion in hand; we have to settle for showing that it is well-formed modulo dependencies.

Finally we can establish our main results:

Theorem 14 (Cut admissibility)**.**

- *If* $\Gamma \vdash M : A^-$ *and* $\Gamma \vdash S : A^- > \bar{U}$*, then* $\Gamma \vdash M \bullet S : \bar{U}$.
- *If* $\Gamma_1 \vdash V : A^+$ *and* $\vdash p : A^+ \Rightarrow \Gamma_2$ *and* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash E : J$ *and* $[V/p]\Gamma_3$ *is defined and* $[V/p]J$ *is defined, then* $\Gamma_1, [V/p]\Gamma_3 \vdash [V/p]E : [V/p]J$.
- *If* $\Gamma_1 \vdash M : A^-$ *and* $\Gamma_1, x{:}A^-, \Gamma_2 \vdash E : J$ *and* $[M/x]\Gamma_2$ *is defined and* $[M/x]J$ *is defined, then* $\Gamma_1, [M/x]\Gamma_2 \vdash [M/x]E : [M/x]J$.
- *If* $\Gamma \vdash R : \mathsf{lax}^+A^+$ *and* $\vdash p : A^+ \Rightarrow \Gamma'$ *and* $\Gamma, \Gamma' \vdash R' : B^+\ \mathsf{lax}^+$*, then* $\Gamma \vdash [R \setminus p]R' : B^+\ \mathsf{lax}^+$.
- *If* $\Gamma \vdash S : C > \mathsf{lax}^+A^+$ *and* $\vdash p : A^+ \Rightarrow \Gamma'$ *and* $\Gamma, \Gamma' \vdash R : B^+\ \mathsf{lax}^+$*, then* $\Gamma \vdash [S \setminus p]R : C > B^+\ \mathsf{lax}^+$.

Proof sketch. By induction on the size of A^- or A^+, with an inner induction on one of the derivations. □

Observe that we do not assume that the contexts or the concluding types are well-formed. This formulation, due to Watkins, *et al.* [28] appears to be the "sweet spot." In each case we have just enough well-formedness to push the proof through. If we make any additional assumptions, the assumptions necessary to push the

induction through snowball rapidly, and consume the bulk of the effort.

Theorem 15 (Identity expansion). *Suppose* $\Vdash \Gamma$: context. *Then:*

- *If* $\Gamma \vdash R : \mathsf{tr}^- A^-$ *then* $\Gamma \vdash \mathsf{Exp}(R : A^-) : A^-$.
- *If* $\Gamma \Vdash A^+ : \mathsf{type}^+$ *and* $\mathsf{Exp}(A^+) = p.V$ *and* $\vdash p : A^+ \Rightarrow \Gamma'$, *then* $\Gamma, \Gamma' \vdash V : A^+$.

Proof sketch. By induction on the size of A^- or A^+. □

5 Formalization

All the results in this paper are formalized using Coq (version 8.4). We implemented binding using deBruijn indices. The formalization can be found at:

www.cs.cmu.edu/~crary/papers/2018/sigma.tgz

The full development is about 15.5k lines (counting comments and whitespace). It takes just over 2 minutes to check using four cores on a 3.4GHz PC with 8GB of RAM.

6 Conclusion

Although the roots of focused logic are in linear logic, we have not said much here about linear logic, other than by way of motivation. The reason is that strong sums do not seem to interact with linear logic in an interesting way. In most linear logics, types are not permitted to refer to linear resources. Suppose $\langle M, N\rangle : \Sigma x{:}A^-.B^-$. Since x stands for M and can appear within the type B^-, that means M cannot contain linear resources. But strong sums are a dependent form of additive product, so M and N must use the same resources. That means that linear resources cannot appear within N either.

Thus, strong sums cannot contain linear resources. In an adjoint logic [2], it would be reasonable to put strong sums at the persistent layer, for that reason. On the other hand, it does seem possible that strong sums could have more interesting substructural behavior in a strict logic (where there is contraction but no weakening), since in strict logic types *can* refer to resources, but we have not explored this.

Zeilberger [31] conjectures that a notion of concurrent equality similar to CLF's [28] could be used to draw a connection between equivalence of focused proofs and normalization-by-evaluation [6] for weak and strong sums. This would be interesting to explore, but there are significant differences between Zeilberger's logic and ours (beyond strong sums). Most significantly, his logic is classical while ours is intuitionistic.

In this work we have confirmed that strong sums can be viewed faithfully as a negative type connective in a focused setting. This has long been suspected, of course, but a workable type system has been elusive. The key innovation is to adapt the concept of selfification from module calculi, thereby eliminating strong sums before spines ever see them.

The connection sheds light in the other direction as well: it seems that polarity may explain why some connectives work in module calculi and others do not. Perhaps a focused type system may provide a way forward for positive connectives in module calculi.

A Typing rules

$\boxed{\vdash p : A^+ \Rightarrow \Gamma}$

$$\frac{}{\vdash \downarrow x : \downarrow A^- \Rightarrow x{:}A^-} \qquad \frac{}{\vdash \star : 1 \Rightarrow \epsilon}$$

$$\frac{\vdash p : A^+ \Rightarrow \Gamma \quad \vdash q : B^+ \Rightarrow \Gamma' \quad BV(p) \cap BV(q) = \emptyset}{\vdash p \otimes q : A^+ \otimes B^+ \Rightarrow \Gamma, \Gamma'}$$

$$\frac{\vdash p : A^+ \Rightarrow \Gamma \quad \vdash q : B^+ \Rightarrow \Gamma' \quad BV(p) \cap BV(q) = \emptyset}{\vdash p \otimes q : \exists p{:}A^+.B^+ \Rightarrow \Gamma, \Gamma'}$$

$\boxed{\Gamma \vdash h : A^-}$

$$\frac{(x : A^-) \in \Gamma}{\Gamma \vdash x : A^-} \qquad \frac{(c : A^-) \in \mathit{Const}}{\Gamma \vdash c : A^-}$$

$\boxed{\Gamma \vdash R : U}$

$$\frac{\Gamma \vdash h : A^- \quad \Gamma \vdash S : A^- > U}{\Gamma \vdash h \cdot S : U}$$

$$\frac{\Gamma \vdash h : A^- \quad \Gamma \vdash S : \mathsf{Self}(\mathsf{Exp}(h \cdot \mathsf{nil} : A^-) : A^-) > U}{\Gamma \vdash h \cdot S : U}$$

$$\frac{\Gamma \vdash V : A^+}{\Gamma \vdash \mathsf{ret}\, V : A^+ \,\mathsf{lax}^+}$$

$\boxed{\Gamma \vdash V : A^+}$

$$\frac{\Gamma \vdash M : A^-}{\Gamma \vdash \downarrow M : \downarrow A^-} \qquad \frac{}{\Gamma \vdash \star : 1}$$

$$\frac{\Gamma \vdash V : A^+ \quad \Gamma \vdash W : B^+}{\Gamma \vdash V \otimes W : A^+ \otimes B^+} \qquad \frac{\Gamma \vdash V : A^+ \quad \Gamma \vdash W : [V/p]B^+}{\Gamma \vdash V \otimes W : \exists p{:}A^+.B^+}$$

$\boxed{\Gamma \vdash S : A^- > U}$

$$\frac{}{\Gamma \vdash \mathsf{nil} : A^- > A^- \,\mathsf{tr}^-} \qquad \frac{\vdash p : A^+ \Rightarrow \Gamma' \quad \Gamma, \Gamma' \vdash R : B^+ \,\mathsf{lax}^+}{\Gamma \vdash \mathsf{bind}\, p.R : \bigcirc A^+ > B^+ \,\mathsf{lax}^+}$$

$$\frac{\Gamma \vdash V : A^+ \quad \Gamma \vdash S : [V/p]B^- > U}{\Gamma \vdash V; S : \Pi p{:}A^+.B^- > U}$$

$$\frac{\Gamma \vdash S : A^- > U}{\Gamma \vdash \pi_1; S : A^- \,\&\, B^- > U} \qquad \frac{\Gamma \vdash S : B^- > U}{\Gamma \vdash \pi_2; S : A^- \,\&\, B^- > U}$$

$\boxed{\Gamma \vdash M : A^-}$

$$\frac{\Gamma \vdash R : (a \cdot S)\,\mathsf{tr}^-}{\Gamma \vdash \eta(R) : a \cdot S} \qquad \frac{\Gamma \vdash R : A^+ \,\mathsf{lax}^+}{\Gamma \vdash \mathsf{circ}\, R : \bigcirc A^+}$$

$$\frac{\vdash p : A^+ \Rightarrow \Gamma' \quad \Gamma, \Gamma' \vdash M : B^-}{\Gamma \vdash \lambda p.M : \Pi p{:}A^+.B^-}$$

$$\frac{\Gamma \vdash M : A^- \quad \Gamma \vdash N : B^-}{\Gamma \vdash \langle M, N\rangle : A^- \,\&\, B^-}$$

$$\frac{\Gamma \vdash M : A^- \quad \Gamma \vdash N : [M/x]B^-}{\Gamma \vdash \langle M, N\rangle : \Sigma x{:}A^-.B^-}$$

$\boxed{\Gamma \vdash S : K > K'}$

$$\frac{}{\Gamma \vdash \mathsf{nil} : K > K} \qquad \frac{\Gamma \vdash V : A^+ \quad \Gamma \vdash S : [V/p]K > K'}{\Gamma \vdash V; S : \Pi p{:}A^+.K > K'}$$

$\boxed{\Gamma \vdash A^- : \mathsf{type}^-}$

$$\frac{(a : K) \in \mathit{Const} \quad \Gamma \vdash S : K > \mathsf{ty}}{\Gamma \vdash a \cdot S : \mathsf{type}^-} \qquad \frac{\Gamma \vdash A^+ : \mathsf{type}^+}{\Gamma \vdash \bigcirc A^+ : \mathsf{type}^-}$$

$$\frac{\Gamma \vdash A^+ : \mathsf{type}^+ \quad \vdash p : A^+ \Rightarrow \Gamma' \quad \Gamma, \Gamma' \vdash B^- : \mathsf{type}^-}{\Gamma \vdash \Pi p{:}A^+.B^- : \mathsf{type}^-}$$

$$\frac{\Gamma \vdash A^- : \mathsf{type}^- \quad \Gamma \vdash B^- : \mathsf{type}^-}{\Gamma \vdash A^- \otimes B^-}$$

$$\frac{\Gamma \vdash A^- : \mathsf{type}^- \quad \Gamma, x{:}A^- \vdash B^- : \mathsf{type}^-}{\Gamma \vdash \Sigma x{:}A^-.B^-}$$

$\boxed{\Gamma \vdash A^+ : \mathsf{type}^+}$

$$\frac{\Gamma \vdash A^- : \mathsf{type}^-}{\Gamma \vdash \downarrow A^- : \mathsf{type}^+} \qquad \frac{}{\Gamma \vdash 1 : \mathsf{type}^+}$$

$$\frac{\Gamma \vdash A^+ : \mathsf{type}^+ \quad \Gamma \vdash B^+ : \mathsf{type}^+}{\Gamma \vdash A^+ \otimes B^+ : \mathsf{type}^+}$$

$$\frac{\Gamma \vdash A^+ : \mathsf{type}^+ \quad \vdash p : A^+ \Rightarrow \Gamma' \quad \Gamma, \Gamma' \vdash B^+ : \mathsf{type}^+}{\Gamma \vdash \exists p{:}A^+.B^+ : \mathsf{type}^+}$$

$\boxed{\Gamma \vdash K : \mathsf{kind}}$

$$\frac{}{\Gamma \vdash \mathsf{ty} : \mathsf{kind}}$$

$$\frac{\Gamma \vdash A^+ : \mathsf{type}^+ \quad \vdash p : A^+ \Rightarrow \Gamma' \quad \Gamma, \Gamma' \vdash K : \mathsf{kind}}{\Gamma \vdash \Pi p{:}A^+.K : \mathsf{kind}}$$

$\boxed{\Gamma \vdash \Gamma' : \mathsf{context}}$

$$\frac{}{\Gamma \vdash \epsilon : \mathsf{context}} \qquad \frac{\Gamma \vdash \Gamma' : \mathsf{context} \quad \Gamma, \Gamma' \vdash A^- \ \mathsf{tr}^-}{\Gamma \vdash \Gamma', x{:}A^- : \mathsf{context}}$$

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

[2] Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Eighth International Workshop on Computer Science Logic*, September 1994.

[3] Iliano Cervesato and Frank Pfenning. A linear logical framework. In *Eleventh IEEE Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996.

[4] Karl Crary. A syntactic account of singleton types via hereditary substitution. In *2009 Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Montreal, 2009.

[5] Karl Crary. Modules, abstraction, and parametric polymorphism. In *Forty-Fourth ACM Symposium on Principles of Programming Languages*, Paris, France, January 2017.

[6] Olivier Danvy. Type-directed partial evaluation. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

[7] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Thirtieth ACM Symposium on Principles of Programming Languages*, pages 236–249, New Orleans, Louisiana, January 2003.

[8] Andrew Gacek. The Abella interactive theorem prover (system description). In *International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*. Springer, August 2008.

[9] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59(3), 1993.

[10] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[11] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[12] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1), 2005.

[13] W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[14] Xavier Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

[15] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logic. *Theoretical Computer Science*, 410(46), 2009.

[16] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Proceedings of the Workshop on the lambda-Prolog Programming Language*, July 1992.

[17] Spiro Michaylov and Frank Pfenning. Higher-order logic programming as constraint logic programming. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, April 1993.

[18] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[19] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[20] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*. Springer, 2010.

[21] Adam Poswolsky and Carsten Schürmann. System description: Delphin — a functional programming language for deductive systems. In *Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Pittsburgh, Pennsylvania, June 2008.

[22] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, Copenhagen, Denmark, January 2011.

[23] Anders Schack-Nielsen and Carsten Schürmann. Celf — a logical framework for deductive and concurrent systems. In *International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*. Springer, 2008.

[24] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, November 2012.

[25] Robert J. Simmons. Structural focalization. *ACM Transactions on Computational Logic*, 15(3), 2014.

[26] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, Boston, January 2000. Extended version published as CMU technical report CMU-CS-99-155.

[27] Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *Thirteenth ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, Odense, Denmark, July 2011.

[28] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, School of Computer Science, 2002. Revised May 2003.

[29] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2004. Papers from the Third International Workshop on Types for Proofs and Programs, April 2003, Torino, Italy.

[30] Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1–3), 2008.

[31] Noam Zeilberger. Polarity and the logic of delimited continuations. In *Twenty-Fifth IEEE Symposium on Logic in Computer Science*, 2010.