The power of detachment: Programming in multiplicative linear logic with control

by

Alex Yi

SB, Electrical Engineering and Computer Science, MIT, 2025.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026

Authored by: Alex Yi
Department of Electrical Engineering and Computer Science
May 22, 2026

Certified by: Adam Chlipala
Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

The power of detachment: Programming in multiplicative linear logic with control

by

Alex Yi

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2026 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Prior work (Aschieri and Genco 2019) has demonstrated considerable strides in bridging the gap between Girard's linear logic and functional interpretations of calculi. I present a simple-yet-powerful extension to their language $\lambda_{\otimes}$, `detach`, that enables a wide variety of patterns to be expressed concisely. I discuss the ability to use this to create compositional operations for easier programming; in particular I explore the ability to use this extension to implement a control operator and several linear connectives.

Thesis supervisor: Adam Chlipala
Title: Professor of Electrical Engineering and Computer Science

Contents

List of Figures

Chapter 1

Background

One might say the marriage of typed lambda calculi and intuitionistic logic is one of the greater romances of the 20th century; locutors of other languages and proprietors of other proof calculi alike often see the easy correspondence between terms and proofs of Curry-Howard and think to themselves: "I want that kind of relationship for me."

Like: Curry-Howard is pretty great, right? You can write a proposition you want to prove as a type, then find a term satisfying that type; if you succeed at finding that term, the proof is sound. Conversely, you can pick up any term you like and wave whatever type inference algorithm you like over it; if the algorithm judges it well-typed and types it with a proposition, that proposition is automatically true. (Whether it is of any use is less-straightforward, granted.)

Unfortunately, the logic described by Curry-Howard—intuitionistic logic—is strictly weaker than classical logic; among other failings, it generally fails to provide computational interpretations of classical-logic laws like the law of the excluded middle ($A \vee \neg A$) and double-negation elimination ($\neg\neg A \to A$). Much ink has been shed on this topic, to little avail; control operators like `call/cc` and $\mathfrak{C}$ recover some aspects, including a functional excluded middle, though double-negation elimination remains elusive (not a very sound proof principle, if the continuation expecting an A can be discarded at will.)

Girard's calculus of linear types has proved promising but difficult to tame; in particular it has lent itself well to linearly typing the terms of the π-calculus, the so-called linear session types. Still, Girard's original formulation introduces several behaviors that have proven difficult to provide computational interpretations for: in particular, an involutive type-level "negation" $(\cdot)^\top$ (that is, it has the property that $(A^\top)^\top \equiv A$). This negation gives rise to four combinators rather than two (as in intuitionistic logic), in dual pairs; while disjoint-sum $\oplus$ finds its dual easily in the consumer-choice sum $\&$, the Cartesian product operator $\otimes$'s dual does not admit such an easy computational interpretation.

Aschieri and Genco 2019 has made strides forward in establishing a Curry-Howard-like correspondence in a linear-typed space; we present a syntactic extension to their work in the form of a new operator, `detach`. This operator allows us to express a term-level semantics for the involutive negation and duality of connectives.

The rest of the thesis is structured as follows: Section 2 describes prior work and the conventions used in this thesis; Section 3 discusses the semantics of our new operator; Section 4 discusses applications of this operator; Section 5 describes limitations and future work.

Chapter 2

Setup

2.1 A brief review of Curry-Howard

One of the nice points of the usual Curry-Howard is that connectives at the type level, expressing concepts like implication, conjunction and disjunction, have strong correlations with terms; furthermore, the computational interpretations of these terms line up nicely with their values.

For example, given terms $a : A$ (read: "a of type A") and $b : B$, we can construct a new term $(a, b) : A \times B$ representing, at the value level, a pair holding terms a and b; and at the type level, the conjunction of proofs of proposition A and proposition B. Expressing this construction more formally in sequent calculus:

$$\frac{\Gamma \Rightarrow a : A \qquad \Gamma \Rightarrow b : B}{\Gamma \Rightarrow (a, b) : A \times B} \times I$$

This sequent is composed of three *type judgements*, each of the form $\Gamma \Rightarrow t : T$, with Γ (and its siblings Δ, Σ, Θ, when necessary) naming a context (a mapping of terms to types), t naming a single term, and T naming a single type; in the usual interpretation of Curry-Howard, this example judgement means "given context Γ, we can judge term t to have type T." The annotation to the right, $\times I$, means that this sequent is the Introduction rule (as opposed to an Elimination rule) for the connective $\times$; in other words, that this rule is used to introduce the $\times$ combinator at the type level.

This intro rule has two interpretations, corresponding to Curry-Howard's duality; one is that a term of type A and a term of type B combine into a term of type $A \times B$. The other, more proof-oriented interpretation is that a proof of proposition A and a proof of proposition B can be conjoined into a proof of $A \times B$, the conjunction of the propositions.

Similarly, the elimination rules for $\times$ ("projection") look as follows:

$$\frac{\Gamma \Rightarrow p : A \times B}{\Gamma \Rightarrow \pi_1(p) : A} \times E_1 \qquad \frac{\Gamma \Rightarrow p : A \times B}{\Gamma \Rightarrow \pi_2(p) : B} \times E_2$$

At the evaluation/term level, these rules suggest that a term of type $A \times B$ can yield a term of type A and/or a term of type B; at the proof-oriented level, these rules suggest that a term representing the conjunction $A \times B$ can be used to prove A or B.

It should be noted that Γ has two meanings, then; in the term-level reading, it comprises a mapping of terms to types that those terms have; and at the proof level, it is instead a list of propositions that are "known" in a given context.

We also generally add *reduction rules*; for example,

$$\pi_1(a, b) \rightsquigarrow_\times a$$
$$\pi_2(a, b) \rightsquigarrow_\times b,$$

which link π_i and $(\cdot, \cdot)$; what one introduces, the other eliminates.

2.2 Prior work in the linear-types space

As aforementioned, general notions of intuitionistic logic fail to admit theories like double-negation elimination; Griffin 1989 presents a formulation that permits these types (and which inspired the definition of callCC in this thesis) but introduces unsoundness. Work has been done to reintroduce these constructs in a linear context (Mazurak and Zdancewic 2010), but that work does not admit the same type of Curry-Howard correspondence.

Many papers have much to say on Girard's linear types as a system for the π-calculus: Honda, Vasconcelos, and Kubo 1998 linked Girard-style linear types to the π-calculus in a program of "session types," though choosing embeddings of connectives that defy logical interpretation (e.g. making $A \otimes B$ represent "output A on this channel, then continue as B," which is obviously not commutative); and Abramsky 1994 and Bellin and Scott 1994 present theories more-directly linking linear connectives to program data directly; but which cause various problems with evaluation. More work over the years culminated in the paper that directly inspired this thesis (Aschieri and Genco 2019): presenting a variant of π-calculus embedding linear λ-calculus and admitting far more expressivity at the evaluation level than its predecessors.

In this thesis, I extend the calculus presented in Aschieri and Genco 2019 in a way that permits the flexibility and expressivity of the constructs depicted in Mazurak and Zdancewic 2010.

2.3 $\lambda_{⅋}$ & co.

The terms in this thesis will generally follow the presentation of $\lambda_{⅋}$ by Aschieri & Genco; we will be extending their calculus with one term that has an (almost) globally relevant reduction rule and then introducing a multitude of other syntactic forms or applicatives (i.e. functions) that make use of this rule (directly or indirectly) to express interesting computations, generally at local scope.

We reproduce a record of their syntax here, for reference, in Figures 2.1 and 2.2.

The core judgement of the sequent calculus is similar to the presentation in the last section, with $\Gamma \;\; \Delta$, with Γ representing a set of (variable) terms and their corresponding types in a context; only that rather than Δ representing a single term-type pair, it can now represent any number of pairs—the net effect being that all these terms exist simultaneously at runtime. Additionally, the judgement rule for close$(\cdot)$ (dual to that of $\circ$) contains terms

$$
\begin{array}{lr}
u, v ::= \circ & \text{(terminated process)} \\
\quad \mid \overline{x}.u & \text{(when applied to a term, output it on } x \text{ and continue with } u\text{)} \\
\quad \mid \overline{xy}\, u & \text{(output } u \text{ on } x, y \text{ and terminate)} \\
\quad \mid u \mid b & \text{(parallel composition)} \\
\quad \mid \mathrm{close}(u) & \text{(close terminated process)} \\
\quad \mid x & \text{(variable)} \\
\quad \mid uv & \text{(term application)} \\
\quad \text{with syntactic sugar:} & \\
\quad \mid \lambda x.u ::= \overline{x}.u &
\end{array}
$$

Figure 2.1: Terms for $\lambda_{⅋}$.

without types; these still participate in evaluation at the term level, but represent the *lack* of a type at the type level— these terms are our only way to eliminate appearances of $\bot$ (which represents a "terminated" process).

Of note is that $\lambda_{⅋}$ is a syncretization of linear λ-calculus and a variation on π-calculus presented in Kokke, Montesi, and Peressotti 2018. Variables (e.g. x) represent places where terms can be "sent" by a corresponding send form, usually of the form $\overline{x}.e$ or $\overline{xy}\, e$, and sending happens immediately (as opposed to in π-calculus, where send forms block until a corresponding "receive" form in another thread synchronizes). Unlike in linear λ-calculus, a send and receive need not be colocated one within the other; this change is the major innovation of Aschieri and Genco 2019 over prior work.

Linear λ-calculus is embedded in this system: the λ symbol is sugar over the unary "send" primitive; and term application follows λ ordering, with $(\overline{x}.e)u$ sending the term u to variable x, then reducing to the term e.

Of interest are the $x \mid y$ and $\overline{xy}\, e$ terms; the former introduces a term of type $⅋$, and the latter eliminates it.

2.4 A word on definitions

Aschieri & Genco's presentation of $\lambda_{⅋}$ was rather-theoretical; I hope to present a more-practical view of their constructions in this thesis. In particular I will be introducing some functional forms and syntactic constructs as definitions that have free variables on their right-hand sides, for example:

$$\eta_+\, x \triangleq \lambda a.xa$$

However, crucially $\lambda_{⅋}$ omits any sort of machinery to deal with identical variables of different types, instead choosing to enforce that all identifiers in a source program are unique! This fact is problematic for us because a perfectly innocuous term:

$$\eta_+\, a$$

$$x : A \;\Rightarrow\; x : A$$

$$\frac{\Gamma, x : A \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \lambda x.t : A \multimap B, \Delta} \multimap I \qquad \frac{\Gamma, x : A \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \overline{x}.t : A \multimap B, \Delta} \multimap I$$

$$\frac{\Gamma \Rightarrow s : A \multimap B, \Delta \qquad \Sigma \Rightarrow t : A, \Theta}{\Gamma, \Sigma \Rightarrow st : B, \Delta, \Theta} \multimap E \qquad \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \circ : \bot, \Delta} \bot I$$

$$\frac{\Gamma \Rightarrow t : \bot, \Delta}{\Gamma \Rightarrow \mathrm{close}(t), \Delta} \bot E \qquad \frac{\Gamma \Rightarrow s : A, t : B, \Delta}{\Gamma \Rightarrow s \mid t : A \parr B, \Delta} \parr I$$

$$\frac{\Gamma \Rightarrow t : A \parr B, \Delta \qquad \Sigma_1, x : A \Rightarrow \Theta_1 \qquad \Sigma_2, y : B \Rightarrow \Theta_2}{\Gamma, \Sigma_1, \Sigma_2 \Rightarrow \overline{xy}\, t : \bot, \Delta, \Theta_1, \Theta_2} \parr E$$

Figure 2.2: Typing judgements for $\lambda_{\parr}$

.

might, under naive expansion of functional form, resolve to an unfortunate ambiguity:

$$\lambda a.aa$$

There are of course several ways to go about resolving this problem, but we take the following principle for convenience: **Free variables on the right-hand side of a definition (i.e. not mentioned on the left-hand side) may be freely α-renamed at substitution time** (to evade name collisions, which are not allowed by $\lambda_{\parr}$).

Because $\lambda_{\parr}$ is an extension of linear λ-calculus, these terms are unable to be duplicated (i.e. single use), so renaming them poses no risk; in fact, all these substitutions can be done "at compile time" by iteratively preprocessing uses of definitions and generating fresh symbols where needed (e.g. renaming all a to a_i for numerically incrementing i). Since programs are finite, we will not run out of bonus symbols; and due to the single-use condition, our symbols also will not duplicate and cause more typing-judgement problems for e.g. preservation.

Chapter 3

Enter `detach`

So with all that out of the way, we present an extension to the language: an extension to the grammar of terms, $u ::= \texttt{detach}(u)$, with the following type judgement:

$$\frac{\Gamma \Rightarrow u : A \parr \bot, \Delta}{\Gamma \Rightarrow \texttt{detach}(u) : A, \Delta} \parr E_r$$

...and the following runtime semantics:

$$C[\texttt{detach}(u \mid v)] \rightsquigarrow C[u] \mid \text{close}(v),$$

when and only when C is the top-level context (i.e. not a subterm of any larger term).

In other words, `detach` is a "local" elimination rule for a $\parr$ term, but only when the right-hand term in the $\parr$ is a computation typed $\bot$ (i.e. one that will terminate properly). Semantically, it expresses "yielding" the $\bot$-typed term to a convenient local scheduler; at the type level, it expresses (one half of) the fact that $\bot$ is an identity term with respect to $\parr$.

Since `detach` is by its nature asymmetrical, we choose as a convention to "keep" its left operand and "discard" its right; it would be pretty trivial to introduce equivalent semantics for an operator that does the opposite (or present it as a matched pair of operators, $\texttt{detach}_l$ and $\texttt{detach}_r$), but it would clutter up the presentation, so we omit it.

3.1 Lowering & proof of soundness

It should be noted that `detach` does not actually give us any more expressive power! Any program that uses `detach` can be rewritten as a program without it—due to the linear nature of variables (i.e. that one can only appear twice in a matched pair); we can trivially lower any well-typed program with `detach` to a program without it by introducing fresh variables "in a definition," as follows:

$$C[\texttt{detach}(x)] \triangleq C[u] \mid \overline{uv}\, x \mid \text{close}(v)$$

This latter program trivially bisimulates the semantics of `detach` we have already given; execution proceeds under context C equivalently, since reduction rules are valid under all contexts, and when x takes the form $a \mid b$, applying our `detach` rule on the left-hand side

is exactly equivalent to applying the relevant two-channel reduction rule on the right. (If we had a way to duplicate terms, e.g. via a ! operator, we would need a more sophisticated proof; but $\lambda_{\invamp}$ omits them, so we shall follow its lead in laziness.)

Soundness, then, falls from the typing rule for `detach`(x); the type of u in this presentation is the same as the type of the left-hand side in the $\invamp$-pair for x, and the type of v is $\bot$ and thus trivially eliminable by the close operator.

However, these lowered programs do not make it immediately obvious how control is meant to flow! The benefit of `detach` is not additional computational power but *expressive* power; with this extra power in hand, programs can be structured far more logically.

In particular, at the type level, what `detach` expresses is that processes that terminate parsimoniously may be "eliminated"; at the semantic level this elimination is done by shifting the process body up to the top level—analogous to spawning a thread, then allowing the language runtime or OS scheduler to manage its running to completion. In fact, this characteristic ability to "set and forget" a thread makes `detach` more favorable to use than vanilla $\lambda_{\invamp}$; imagine having to specify every thread that your program was going to use, at a global declaration site, before the program even begins!

3.2 Connections to real-world semantics

The ability for `detach` to express "spawning a thread," similar to C's `fork` or JavaScript's `setTimeout`/`setImmediate`, posits the existence of a runtime construct a bit more. As Aschieri & Genco note, most programs can be written in the form of a bunch of terms connected by |'s; our abstract semantics are nondeterministic and make no distinction on what order our reductions should happen (even after we impose a call-by-value semantics), but generally in the real world we have the notion of a "scheduler," whether it exists at the OS level being provided by process-level parallelism or a language runtime's event loop driving things forward. With `detach`, then, we express a sort of "dynamism" with that notion of a scheduler; instead of being a preset list of terms determined at program start, it allows us to add a term to the scheduler's work queue in a measured way, trusting it to get the work in that term done parsimoniously.

Chapter 4

Examples

4.1 Calling and control

As an astute reader may notice, the behavior of `detach` somewhat resembles a control operator; in particular the ability to raise an arbitrary term to the "top level." It may be unsurprising, then, that we can recover a limited form of a classical control operator directly as a term: a sort of "callCC."

In fact, it is trivial to generate a value-continuation pair in the semantics of $\lambda_{\invamp}$; if x is given type A, then $\bar{x}.\circ$ naturally has dual type $A \multimap \bot$. In $\lambda_{\invamp}$ this duality is a special case of communication between processes; with `detach`, this duality allows us to write more classically compositional programs.

Note that the traditional type judgement for callCC, i.e. $((A \multimap \bot) \multimap A) \multimap A$, is uninhabitable in our language in the general case. This fact is seen easily with polarity; with two As in negative position and one in positive, there exists no way to create a term with this type without sourcing an additional A from somewhere.

However, we can still generate a function with similar semantics, $\text{callCC} : ((A \multimap \bot) \multimap \bot) \multimap A$. This function can be implemented as follows:

$$\text{callCC}\, k \triangleq \texttt{detach}(u \mid k(\overline{u}.\circ))$$

In other words, our continuation-accepting thunk is simply applied to a function sending a "returned" value along u "on a separate thread"; returning $\bot$ is necessary to ensure that no resources are leaked by the end of computation. Meanwhile, our is simply the opaque value u, pausing further computation in immediately enclosing terms until the matching $\overline{u}$ term resolves. Indeed, from the point of view of the continuation-accepting thunk k, its argument "is" the context in which it was called, in that passing in a value resumes the context using the value it was called with (albeit not destroying the current context).

This behavior does have broader implications in the sense that it simulates the semantics of a C-style "function call;" with u in our definition representing a kind of "return address," and the term yielded to the scheduler the "stack frame" on which computation should be done. The continuation passed into the term, then, is a sort of reified `return` form, returning an object representing the termination of the subroutine when applied, rather than needing to float the result up to the top level. This semantics is notably more compatible with how many modern languages express `return` in function bodies!

This analogy does, however, raise the question of generalization; after all, with the tools of parallelism in this language, we need not wait on a single term, nor must our sequence of calls at any given point form a simple stack structure.

One such thing to notice is that, if we do not assume the context outside callCC waits on it to finish evaluation and instead proceeds apace, callCC can be seen as a sort of "future" constructor, placing k on a separate "thread" to run, then replacing the floating u term when the computation finishes.

By the way, note also that the type of our callCC is exactly $(A^\top)^\top \multimap A$; that is to say, linear double-negation elimination!

This fact turns out to give us a lot of ground with regards to duality, in the sense of "inverting" a term; if a term x is typed A, its typed dual $A^\top$ is simply termed $\overline{x}.\circ$; to recover a value A from a term $k : A^\top$, we can use callCC($\overline{k}.\circ$).

As we see in the following sections, this notion of duality is not just a parlor trick; it in fact allows us to recover connectives from one another.

4.2 Converting between $\multimap$ and $\parr$

Girard identifies the linear implication operator $A \multimap B$ with $A^\top \parr B$; while vanilla $\lambda_\parr$ entails this equivalence in the forward direction, `detach` allows us to express it in the reverse as follows:

$$\text{impl2par}(t : A \multimap B) \triangleq \lambda x.\circ \mid tx : A^\top \parr B$$
$$\text{par2impl}(t : A^\top \parr B) \triangleq \lambda x.\texttt{detach}(v \mid \texttt{detach}(xu \mid \texttt{detach}(\overline{uv}\, t))) : A \multimap B$$

These functions are inverses modulo reduction; impl2par and par2impl compose to identity.

4.3 Duality of sums

If we extend $\lambda_\parr$ with one of the & and $\oplus$ combinators, with the usual semantics, it turns out that `detach` is enough to recover the other combinator entirely!

Without loss of generality, let us use $\oplus$ (since it is somewhat more common). We add:

- $\tau ::= \tau \oplus \tau$ to our grammar of types;
- $e ::= \text{inl}\, e \mid \text{inr}\, e \mid \text{case}\, e\, \text{of inl}\, u \to e_1; \text{inr}\, v \to e_2\, \text{end}$ to our grammar of terms (with u and v being variable names and e_i being arbitrary terms);
- reduction rules $\text{case inl}\, e\, \text{of inl}\, u \to e_1; \text{inr}\, v \to e_2\, \text{end} \rightsquigarrow (\lambda u.e_1)e$ and $\text{case inr}\, e\, \text{of inl}\, u \to e_1; \text{inr}\, v \to e_2\, \text{end} \rightsquigarrow (\lambda v.e_2)e$,
- and typing judgements as follows:

$$\frac{\Gamma \Rightarrow t : A, \Delta}{\Gamma \Rightarrow \text{inl}\, t : A \oplus B, \Delta} \oplus I_l \qquad \frac{\Gamma \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \text{inr}\, t : A \oplus B, \Delta} \oplus I_r$$

$$\frac{\Gamma \Rightarrow t : A \oplus B, \Delta \qquad \Sigma, u : A \Rightarrow e_1 : R, \Theta \qquad \Sigma, v : B \Rightarrow e_2 : R, \Theta}{\Gamma, \Sigma \Rightarrow \ \mathrm{case}\, t\, \mathrm{of\, inl}\, u \to e_1; \mathrm{inr}\, v \to e_2\, \mathrm{end} : R, \Delta, \Theta} \oplus E$$

These additions comprise a bog-standard implementation of a linear disjoint sum; you can go read another paper on linear types if $\oplus$ is unfamiliar. That said, the magic trick I will show off looks like so:

We define syntax sugar $A \& B$ to be $(A \multimap \bot \oplus B \multimap \bot) \multimap \bot$ and want to define term $\mathrm{case\, outl} \to e_1; \mathrm{outr} \to e_2\, \mathrm{end}$ to introduce $A \& B$ and terms $\mathrm{outl}\, e$ and $\mathrm{outr}\, e$ to eliminate it.

Intuitively, our definition of $A \& B$ is a de Morgan's-law term; to introduce it, we have to be able to eliminate either a $A \multimap \bot$ or a $B \multimap \bot$; and to eliminate it, we have to introduce one of these to pass to the eliminator we just created. Luckily, we just introduced the tool for the job—that is, callCC—in the previous section!

We thus define our syntax as such:

$$\begin{aligned} \mathrm{case\, outl} \to e_1; \mathrm{outr} \to e_2\, \mathrm{end} &\triangleq \lambda k.\, \mathrm{case}\, k\, \mathrm{of\, inl}\, k_a \to k_a e_1; \mathrm{inr}\, k_b \to k_b e_2\, \mathrm{end} \\ \mathrm{outl}\, e &\triangleq \mathrm{callCC}\, \lambda k. e(\mathrm{inl}\, k) \\ \mathrm{outr}\, e &\triangleq \mathrm{callCC}\, \lambda k. e(\mathrm{inr}\, k) \end{aligned}$$

As it turns out, these constructs behave exactly as a "language-level" implementation of & would, but eliding the need to write separate judgement rules: in particular, we find that

$$\begin{aligned} & \mathrm{outl}(\mathrm{case\, outl} \to e_1; \mathrm{outr} \to e_2\, \mathrm{end}) \\ \rightsquigarrow\ & \mathrm{callCC}(\lambda k.(\mathrm{case\, outl} \to e_1; \mathrm{outr} \to e_2\, \mathrm{end})(\mathrm{inl}\, k)) \\ \rightsquigarrow_{\mathrm{def}}\ & \mathrm{callCC}(\lambda k.(\lambda k_0.\, \mathrm{case}\, k_0\, \mathrm{of\, inl}\, k_a \to k_a e_1; \mathrm{inr}\, k_b \to k_b e_2\, \mathrm{end})(\mathrm{inl}\, k)) \\ \rightsquigarrow_{\beta}\ & \mathrm{callCC}(\lambda k.\, \mathrm{case\ inl}\, k\, \mathrm{of\, inl}\, k_a \to k_a e_1; \mathrm{inr}\, k_b \to k_b e_2\, \mathrm{end}) \\ \rightsquigarrow_{\oplus E}\ & \mathrm{callCC}(\lambda k.(\lambda k_a.k_a e_1)k) \\ \rightsquigarrow_{\beta}\ & \mathrm{callCC}(\lambda k.k e_1) \\ \rightsquigarrow^*\ & e_1) \end{aligned}$$

as desired (and similarly for e_2); and the typing judgements for $\& I$, $\& E_l$, $\& E_r$ (dual to those for $\oplus I_l$, $\oplus I_r$, $\oplus E$) are provable just by similar expansion (and are sound "for free," since they are not language-level axioms).

This construction is interesting for several reasons! In particular, it expresses the duality between $\oplus$ and & at a computational level; having one recovers the other in a de Morgan fashion. Of note, too, is that despite our callCC (and thus implicitly `detach`) technically lifting a term to a global state, this "raised" term resolves immediately; we instantly call the continuation k with e_1, reducing to a close($\circ$) at the top level.

Still, this construction would be impossible to well-type without `detach` or another control-like operator; in this way, it acts as a sort of "magic camel" to express an interesting control flow while keeping the type system happy.

4.4 Duality of products

Following in our success with sums, we ask: can we recover $\otimes$, the tensor (Cartesian product) from the $\lambda_{\otimes}$ we have natively? The answer is yes!

As before, we define $A \otimes B$ to be $(A \multimap \bot \parr B \multimap \bot) \multimap \bot$; we then seek to find definitions for terms (u, v) and let $(x, y) = u$ in v such that let $(x, y) = (a, b)$ in u is confluent with $(\lambda x.\lambda y.u)ab$, and the following type judgements are satisfied:

$$\frac{\Gamma \Rightarrow u : A, \Delta \qquad \Sigma \Rightarrow v : B, \Theta}{\Gamma, \Sigma \Rightarrow (u, v) : A \otimes B, \Delta, \Theta} \otimes I$$

$$\frac{\Gamma \Rightarrow u : A \otimes B, \Delta \qquad \Sigma, x : A, y : B \Rightarrow v : C, \Theta}{\Gamma, \Sigma \Rightarrow \text{let } (x, y) = u \text{ in } v : C, \Delta, \Theta} \otimes E$$

It turns out one possible implementation is as follows:

$$\begin{aligned} (u, v) &\triangleq \texttt{detach}(\texttt{detach}(\lambda k.\overline{xy}\, k \mid xu) \mid yv) \\ \text{let } (x, y) = u \text{ in } v &\triangleq \texttt{detach}(v \mid u(\overline{x}.\circ \mid \overline{y}.\circ)) \end{aligned}$$

This implementation of course satisfies the preconditions we wanted, by computation; more interestingly, it does "parallelize" evaluation of its arguments (as specified in its typing rule), even with a call-by-value eval strategy, since xu and yv are on separate sides of a $\parr$ and lifted to the top level.

Note also that, at least in this formulation, those top-level evaluation points are precisely where the arguments are "stored," blocking application of x and y (and subsequently termination) until resolved at a destructuring let.

With this section and the last, though, we have shown that it is possible to recover all four non-exponential connectives in $\lambda_{\parr}$ from only two! How exciting!

Chapter 5

Future work

While already very functional, `detach` is somewhat of a syntactically janky construct; in particular, eliminating multi-branch $\lambda_{\parr}$ expressions needs n `detach` calls for $n+1$ branches. It would be interesting to generalize this to a less jank system.

In Section 4.1, we discuss the capability for callCC to express program call structures beyond simple stacks; in fact, it mimics several primitives like futures (from a local point of view) and green threads (from a more global perspective). Integrating this type system into preexisting structures using these constructs would be a worthwhile pursuit.

Additionally, it seems very possible that Girard's exponential connectives, ? and !, can be dualized in a similar manner to our other connectives; and that we can indeed have an entirely functional λ-style computational interpretation of Girard's linear types, as foretold. Hooray!

Also, all this theory is nice and all, but it would be really nice to have an interpreter verifying all this. Just a thought.

Chapter 6

Acknowledgements

I would like to thank Prof. Adam Chlipala for his insight and guidance on this project; without his support this project would not have been possible. Thank you for your patience in revising this thesis document.

Thank you to Prof. Ankush Das at Boston University for providing valuable direction and feedback on my research.

Thank you, Cora, Lucy, & Maddy; without your friendship & companionship I never would have made it this far.

Thank you to Luna and John, Annie and Francisco, and Diana; you were the best undergraduate friends I could have asked for.

Thank you to my mother and father for supporting me through all this. I'm so lucky to have had the opportunity to do this thesis, and I couldn't have done any of it without you.

Bibliography

Aschieri, F. and F. A. Genco (Dec. 2019). "Par means parallel: multiplicative linear logic proofs as concurrent functional programs." *Proc. ACM Program. Lang.*, **4**(POPL), Dec. 2019. DOI: 10.1145/3371086. URL: https://doi.org/10.1145/3371086.

Griffin, T. G. (1989). "A formulae-as-type notion of control." In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. San Francisco, California, USA: Association for Computing Machinery, pp. 47–58. ISBN: 0897913434. DOI: 10.1145/96709.96714. URL: https://doi.org/10.1145/96709.96714.

Mazurak, K. and S. Zdancewic (2010). "Lolliproc: to concurrency from classical linear logic via curry-howard and control." In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: Association for Computing Machinery, pp. 39–50. ISBN: 9781605587943. DOI: 10.1145/1863543.1863551. URL: https://doi.org/10.1145/1863543.1863551.

Honda, K., V. Vasconcelos, and M. Kubo (Mar. 1998). "Language Primitives and Type Discipline for Structured Communication-Based Programming." In: vol. 1381, pp. 122–138. ISBN: 978-3-540-64302-9. DOI: 10.1007/BFb0053567.

Abramsky, S. (1994). "Proofs as processes." *Theoretical Computer Science*, **135**(1), 1994, pp. 5–9. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(94)00103-0. URL: https://www.sciencedirect.com/science/article/pii/0304397594001030.

Bellin, G. and P. Scott (1994). "On the π-calculus and linear logic." *Theoretical Computer Science*, **135**(1), 1994, pp. 11–65. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(94)00104-9. URL: https://www.sciencedirect.com/science/article/pii/0304397594001049.

Kokke, W., F. Montesi, and M. Peressotti (2018). *Better Late Than Never: A Fully Abstract Semantics for Classical Processes*. arXiv: 1811.02209 [cs.LO]. URL: https://arxiv.org/abs/1811.02209.