

Intrinsic Verification of Parsers and Formal Grammar Theory in Dependent Lambek Calculus (Extended Version)

STEVEN SCHAEFER, University of Michigan, USA

NATHAN VARNER, University of Michigan, USA

PEDRO HENRIQUE AZEVEDO DE AMORIM, University of Oxford, United Kingdom

MAX S. NEW, University of Michigan, USA

We present Dependent Lambek Calculus (Lambek^D), a domain-specific dependent type theory for verified parsing and formal grammar theory. In Lambek^D , linear types are used as a syntax for formal grammars, and parsers can be written as linear terms. The linear typing restriction provides a form of intrinsic verification that a parser yields only valid parse trees for the input string. We demonstrate the expressivity of this system by showing that the combination of inductive linear types and dependency on non-linear data can be used to encode commonly used grammar formalisms such as regular and context-free grammars as well as traces of various types of automata. Using these encodings, we define parsers for regular expressions using deterministic automata, as well as examples of verified parsers of context-free grammars.

We present a denotational semantics of our type theory that interprets the linear types as functions from strings to sets of abstract parse trees and terms as parse transformers. Based on this denotational semantics, we have made a prototype implementation of Lambek^D using a shallow embedding in the Agda proof assistant. All of our examples parsers have been implemented in this prototype implementation.

CCS Concepts: • **Theory of computation** → *Denotational semantics*; **Grammars and context-free languages**; • **Software and its engineering** → *Domain specific languages*; **Parsers**; Formal software verification.

1 Introduction

Parsing structured data from untrusted input is a ubiquitous task in computing. Any formally verified software system that interacts with the outside world must contain some parsing component, and unverified parsers can undermine the overall correctness theorem for the system. For example, in an extensive experiment finding bugs in C compilers [Yang et al. 2011], an early version of CompCert, the formally verified C compiler, was found to have bugs only in the then unverified parsing component [Leroy 2009]. The correctness of the compiler only showed that the abstract syntax tree would be compiled correctly, but this is not very useful if it did not correctly correspond to the actual source program. Eventually, CompCert was updated to use a verified parser that implements an LR(1) grammar [Jourdan et al. 2012].

It is entirely understandable from an engineering perspective *why* verified parsing was not part of the initial releases of CompCert: parsing algorithms are a complex area, featuring a variety of domain-specific formalisms such as context-free grammars and various automata. These formalisms have little relation to the main components of a verified compiler. For this reason, it is advantageous for verified parsers to be implemented using a reusable verified library, just as parser generators and regular expression matchers have done for many decades in unverified software.

Prior approaches to verified parsing focus on verification of a particular grammar formalism, such as non-left-recursive grammars or LL(1) grammars [Danielsson 2010; Edelmann et al. 2020; Lasser et al. 2021]. This has led to a series of isolated solutions, as each new grammar formalism is extended with its own independent verified implementation.




This work is licensed under a Creative Commons Attribution 4.0 International License.

In this work, we present the design of Dependent Lambek Calculus (Lambek^D), a domain-specific language for formal verification of parsers. A key property is that Lambek^D is an *extensible* framework for verification of parsers in that it supports the definition of grammar formalisms of unrestricted complexity. That is, Lambek^D is not a system for verifying *one* type of grammar formalism, but instead is a domain-specific language in which many grammar formalisms and their verified parsers can be implemented. For example, Lambek^D itself is not a verified parser generator compiling regular expressions to deterministic finite automata but is instead a domain specific language in which we can write such a verified parser generator.

The design of Lambek^D is an extension of Joachim Lambek’s *syntactic calculus* [Lambek 1958]. Lambek calculus is a grammar formalism equivalent in expressive power to context-free grammars that in modern terminology would be considered a kind of *non-commutative linear logic* — a version of linear logic where the tensor product is not commutative, reflecting the obvious property that the relative ordering of characters is significant in parsing problems. We extend non-commutative linear logic with two key components that increase its power to support arbitrarily powerful grammar formalisms: inductive linear¹ types, as well as dependency of linear types on non-linear data. The resulting system has two kinds of types: non-linear types which model sets and linear types which model formal grammars. Crucially, the non-linear types and linear types are allowed to be dependent on non-linear types, but not on linear types. This combination has been used previously in the “linear-non-linear dependent” type theory with *commutative* linear logic to model imperative programming [Krishnaswami et al. 2015].

The substructural nature of Lambek^D is well-aligned with the requirements intrinsic to parsing and the theory of formal languages, where strings constitute a clear notion of resource that cannot be duplicated, reordered, or dropped. Moreover, Lambek^D ensures that parsers written in the calculus are *correct-by-construction*. That is, our type system is rich enough that typing derivations carry intrinsic proofs of parser correctness. Parsers written in Lambek^D take on a linear functional style, which makes them familiar to write and amenable to compositional verification techniques.

To show the feasibility of our design, we have implemented Lambek^D as a shallowly embedded domain-specific language in the Cubical Agda proof assistant [Vezzosi et al. 2019]. We have implemented many example grammars and parsers in our system including regular expressions, non-deterministic and deterministic automata, as well as some example LL(1) context-free grammars and parsers using stack-based automata. Throughout this paper  will mark results that are mechanized in our Agda development and provide a link to their implementation.

Our Agda prototype is based on a *denotational semantics* of Lambek^D. The core idea of the denotational semantics stems from an observation of Elliott: an abstract notion of formal grammar can be given by a “proof-relevant” predicate on strings [Elliott 2021]. That is, a *formal grammar* A is a function $\mathbf{String} \rightarrow \mathbf{Set}$ such that for a string w , $A\ w$ is the set of “proofs” showing that w belongs to the language recognized by A . We show that all linear types in Lambek^D can be so interpreted as an abstract formal grammar in this sense, and that linear terms can be interpreted as a kind of *parse transformer*, a function that takes a parse tree from one grammar to a parse tree in a different grammar but over the same underlying string.

Our contributions are then:

- The design of Dependent Lambek Calculus (Lambek^D): A dependent linear-non-linear type theory for building verified parsers, which extends prior work on dependent linear-non-linear type theory to support inductive linear types.
- A demonstration of how to encode many common grammar and parser formalisms (regular expressions, (non-)deterministic automata, context-free grammars) within our type theory.

¹Throughout, we shall use “linear types” to refer to the non-commutative linear types.

- A prototype implementation of Lambek^D in Agda with all examples mechanized.
- A denotational semantics for Lambek^D that shows that the parsers are in fact verified to be correct, and the equational theory is sound.

This paper begins in Section 2 by studying small example programs from Lambek^D to build intuition. From there, in Section 3 we provide the syntax, typing, and equational theory of Dependent Lambek Calculus. In Section 4 we demonstrate the applicability of Lambek^D for relating familiar grammar and automata formalisms as well as building concrete parsers. Then in Section 5, we give a denotational semantics that makes precise the connection between Lambek^D syntax and formal grammars. Finally, in Section 6 we discuss related and future work.

2 Dependent Lambek Calculus by Example

To gain intuition for working in Lambek^D , we begin with some illustrative examples drawn from the theory of formal languages. Each of our examples will be defined for strings over the three character alphabet $\Sigma = \{a, b, c\}$.


Finite Grammars. First consider finite grammars — those built from base types via disjunctions and concatenations. The base types comprise characters drawn from the alphabet, the empty string, and the empty grammar. For each character a in the alphabet we have a type ' a ' which has a single parse tree for the string " a " and no parse trees at any other strings. The grammar I has a single parse tree for the empty string $\epsilon = ""$ and no parses for any other strings. The final base type, the empty grammar \emptyset , has no parses for any string. We use type-theoretic syntax to represent disjunction \oplus and concatenation \otimes of grammars. Over an input string w , a parse of the disjunction $A \oplus B$ is either a parse of A over the string w or a parse of B over the string w , along with a tag inl or inr indicating which case was taken. A parse of $A \otimes B$ for w is a splitting of w into two strings w_A and w_B with parses for A and B , respectively.

A derivation of a word w in a grammar A is given by a term in our calculus that satisfies the typing $[w] \vdash e : A$, where $[w]$ is a context with one variable for each character of w . The term $e : A$ represents a *parse tree* of w for the grammar A . For example, to define a parse tree for " ab ", we use the context $["ab"] = a : 'a', b : 'b'$. In Figure 1, we give a lambda term and its typing derivation to define a parse for a finite grammar.

For this interpretation of parse trees as terms to make sense, our calculus cannot allow for *any* of the usual structural rules of type theory: weakening, contraction and exchange. Weakening allows for variables to go unused, which would allow for a character in an input string to be ignored, yielding the erroneous parse tree $a : 'a', b : 'b' \not\vdash a : 'a'$. Contraction allows for the same variable to be used twice, yielding the erroneous parse $a : 'a', b : 'b' \not\vdash (a, a) : 'a' \otimes 'a'$. Finally, the ordering of characters in a string cannot be ignored while parsing, so we omit the exchange rule because it would allow for variables in the context to be reordered. This prevents the erroneous derivation, $a : 'a', b : 'b' \not\vdash (b, a) : 'b' \otimes 'a'$.

```
f : ↑('a' ⊗ 'b' → ('a' ⊗ 'b') ⊕ 'c')
f (a , b) = inl (a , b)
```

$$\frac{\frac{a : 'a' \vdash a : 'a' \quad b : 'b' \vdash b : 'b'}{a : 'a', b : 'b' \vdash (a, b) : 'a' \otimes 'b'}}{a : 'a', b : 'b' \vdash f := \text{inl}(a, b) : ('a' \otimes 'b') \oplus 'c'}$$

Fig. 1.  " ab " is parsed by $('a' \otimes 'b') \oplus 'c'$

```

data A* : L where
  nil : ↑(A*)
  cons : ↑(A → A* → A*)

```

Fig. 2. Kleene Star as an inductive type

$$\begin{array}{c}
\frac{}{\cdot \vdash \text{cons} : \uparrow ('a' \multimap 'a'^* \multimap 'a'^*)} \quad \frac{}{\cdot \vdash \text{nil} : \uparrow ('a'^*)} \\
\frac{}{\cdot \vdash \text{cons} : 'a' \multimap 'a'^* \multimap 'a'^*} \quad \frac{}{a : 'a' \vdash a : 'a'} \quad \frac{}{\cdot \vdash \text{nil} : 'a'^*} \\
\frac{}{a : a \vdash \text{cons } a : 'a'^* \multimap 'a'^*} \quad \frac{}{\cdot \vdash \text{nil} : 'a'^*} \\
\frac{}{a : 'a' \vdash \text{cons } a \text{ nil} : 'a'^*} \quad \frac{}{b : 'b' \vdash b : 'b'} \\
\frac{}{a : 'a', b : 'b' \vdash (\text{cons } a \text{ nil}, b) : 'a'^* \otimes 'b'} \\
\frac{}{a : 'a', b : 'b' \vdash g := \text{inl}(\text{cons } a \text{ nil}, b) : ('a'^* \otimes 'b') \oplus 'c'}
\end{array}$$

```

g : ↑((('a' ⊗ 'b') → ('a'* ⊗ 'b')) ⊕ 'c')
g (a , b) = inl (cons a nil , b)

```

Fig. 3. $(\text{inl}(\text{cons } a \text{ nil}, b))$ "ab" is parsed by $('a'^* \otimes 'b') \oplus 'c'$

Regular Expressions. Regular expressions can be encoded as types generated by base types, \oplus , and \otimes , and the Kleene star $(\cdot)^*$. For a grammar A , we define the Kleene star A^* as a particular *inductive linear type* of linear lists, as shown in Fig. 2. Here $A^* : L$ means we are defining A^* to be a *linear type*. A^* has two constructors: `nil`, which builds a parse of type A^* from nothing; and `cons`, which linearly consumes a parse of A and a parse of A^* and builds a parse of A^* . This linear consumption is defined by the linear function type \multimap . The linear function type $A \multimap B$ defines functions that take in parses of A as input, *consume* the input, and return a parse of B as output. The arrow, \uparrow , wrapping these constructors means that the constructors are not consumed upon usage, and so are *non-linear* values themselves, which are not small. That is, the names `nil` and `cons` are function symbols that may be reused as many times as we wish.

Through repeated application of the Kleene star constructors, Fig. 3 gives a derivation that shows "ab" is parsed by the regular expression $('a'^* \otimes 'b') \oplus 'c'$. The leaves of the proof tree that mention the arrow \uparrow describe a cast from a non-linear type to a linear type. For instance, the premise of the leaf involving `nil` views `nil : ↑('a'^*)` as the name of a constructor, and a constructor should be nonlinearly valued because we may call it several times (or not at all). However, the conclusion of this leaf views `nil : 'a'^*` as a linear value, which in our syntax is an implicit coercion from a nonlinear value to a linear value. After we call the constructor it "returns" a value that may only be used a single time.

We may also have derivations where the term in context is not simply a string of literals. In Fig. 4 we show that every parse of the grammar $(A \otimes A)^*$ induces a parse of A^* for an arbitrary grammar A . The context $(A \otimes A)^*$ does not correspond directly to a string, so it is not quite appropriate to think of a linear term here as a parse *tree*. The context $a : (A \otimes A)^*$ does not contain concrete data to be parsed; rather, there may be many choices of string underlying the parse tree captured by the variable a . Thus, the term h from Fig. 4 is not a parse of a string, and it is more appropriate to think of it as a parse *transformer* — a function from parses of $(A \otimes A)^*$ to parses of A^* .

We define h by recursion on terms of type $(A \otimes A)^*$. This recursion is expressed in the derivation tree by invoking the elimination principle for Kleene star, written as `fold`. The parse transformer

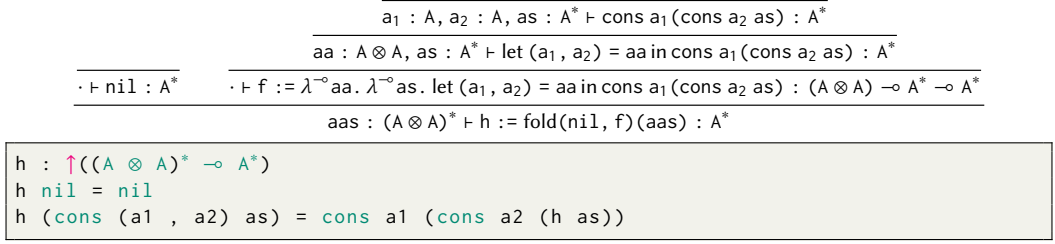


Fig. 4. A parse transformer for abstract grammars

h is more intuitively presented in the pseudocode of Fig. 4 by pattern matching on the input and making an explicit recursive call in the body of its definition.

Non-deterministic Finite Automata. Regular expressions are a compact formalism for defining a formal grammar, but an expression such as $('a'^* \otimes 'b') \oplus 'c'$ does not give an operationalized method for parsing. For this reason, most parsers are implemented by compiling a grammar to a corresponding automaton, which is readily implemented. To implement these algorithms in Lambek^D , we represent automata as types, just as we did with regular expressions.

Finite automata are precisely the class of machines that recognize regular expressions. Fig. 5 shows a non-deterministic finite automaton (NFA) for the regular expression $('a'^* \otimes 'b') \oplus 'c'$, along with a type Trace , an *indexed* inductive linear type of traces through this automaton. Defining an indexed inductive type can be thought of as defining a family of mutually recursive inductive types, one for each element of the indexing type. Here Trace uses an index $s : \text{Fin } 3$ which picks out which state in the automaton a trace begins at — where $\text{Fin } 3$ is the finite type containing inhabitants $\{0, 1, 2\}$. We can think of this as defining three mutually recursive inductive types $\text{Trace } 0$, $\text{Trace } 1$, and $\text{Trace } 2$. There are three kinds of constructors for Trace : (1) those that terminate traces, (2) those that correspond to transitions labeled by a character, and (3) those that correspond to transitions labeled by the empty string ϵ . The constructor stop terminates a trace in the accepting state 2. The constructors $1\text{to}1$, $1\text{to}2$, $0\text{to}2$ each define a labeled transition through the NFA, and each of these consumes a parse of the label's character and a trace beginning at the destination of a transition to produce a trace beginning at the source of a transition. The constructor $0\text{to}1$ behaves similarly, except its transition is labeled with the empty string ϵ . Therefore, $0\text{to}1$ takes in a trace beginning at state 1 and returns a trace beginning at state 0 corresponding to the same underlying string. Lastly, we give a λ term that constructs an accepting trace starting at the initial state for the string "ab". Later in Section 4, we will show that we can actually construct mutually inverse functions between the regular expression $('a'^* \otimes 'b') \oplus 'c'$ and its corresponding NFA traces ($\text{Trace } 0$) demonstrating that the regular expression and the automaton capture the same language. Further, since the functions are mutually inverse, this shows they are *strongly equivalent* as grammars.

3 Syntax and Typing for Dependent Lambek Calculus

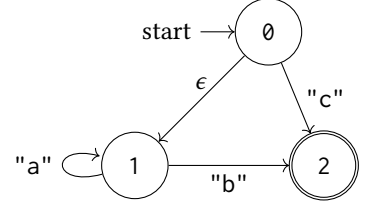
The design of Lambek^D is based on the dependent linear-non-linear calculus (LNL_D) and Lambek calculus, also known as non-commutative linear logic [Krishnaswami et al. 2015; Lambek 1958]. As in LNL_D , Lambek^D includes both non-linear dependent types, as well as linear types, which are allowed to depend on the non-linear types, but not on other linear types. The main point of departure from LNL_D 's design is that, as in Lambek calculus [Lambek 1958], the linear typing is

```

data Trace : (s : Fin 3) → L where
  stop : ↑(Trace 2)
  1to1 : ↑('a' → Trace 1 → Trace 1)
  1to2 : ↑('b' → Trace 2 → Trace 1)
  0to2 : ↑('c' → Trace 2 → Trace 0)
  0to1 : ↑(Trace 1 → Trace 0)

k : ↑(('a' ⊗ 'b') → Trace 0)
k (a , b) = 0to1 (1to1 a (1to2 b stop))

```

Fig. 5. (NFA) NFA for $(a^* \otimes b) \oplus c$ and its corresponding type

$$\begin{array}{c}
 \boxed{\Gamma \text{ ctx}} \quad \frac{}{\cdot \text{ ctx}} \quad \frac{\Gamma \text{ ctx} \quad \Gamma \vdash X \text{ type}}{\Gamma, x : X \text{ ctx}} \quad \boxed{\Gamma \vdash \Delta \text{ lin. ctx.}} \quad \frac{}{\Gamma \vdash \cdot \text{ lin. ctx.}} \quad \frac{\Gamma \vdash \Delta \text{ lin. ctx.} \quad \Gamma \vdash A \text{ lin. type}}{\Gamma \vdash \Delta, a : A \text{ lin. ctx.}}
 \end{array}$$

Fig. 6. Context well-formedness rules

non-commutative — i.e., that exchange is not an admissible structural rule. Furthermore, we add a general-purpose indexed inductive linear type connective, as well as an *equalizer* type, which we will show allows us to perform inductive proofs of equalities between linear terms. Finally, while LNL_D was enhanced with special connectives inspired by separation logic to model imperative programming, we instead add base types and axioms to the system specifically to model formal grammars and parsing.

The formation rules for the judgments of Lambek^D are shown in Figures 6 to 8. Γ stands for non-linear contexts; X, Y, Z stand for non-linear types; M, N stand for non-linear terms, these act as in an ordinary dependent type theory; Δ stands for linear contexts; A, B, C for linear types; and, e, f, g for linear terms. These contexts, types and terms are allowed to depend on an ambient non-linear context Γ , but note that linear types A cannot depend on any *linear* variables in Δ . We include definitional equality judgments for both kinds of type and term judgments as well. Additionally, we have judgments $\Gamma \vdash X$ small and $\Gamma \vdash A$ small lin. which are used in the definition of universe types.

3.1 Non-linear Typing

We present a selection of the non-linear type constructors in Figure 7 and provide the rest in Figs. 20 and 21 of Appendix A. First, we include universe types U of small non-linear types and L of linear types. These are defined as universes “ala Coquand” in that we define judgments saying when non-linear and linear types are *small* and define the universes to internalize precisely this judgment [Coquand 2013; Gratz et al. 2021]. The definition of smallness is simply that all types are small as long as their sub-formulae are, with the exception of the two universe types themselves. A formal description of smallness is given in Figs. 18 and 19 of Appendix A. These universe types are needed so that we can define non-linear and linear types by recursion on natural numbers. Next, we include standard Σ , Π , empty, unit, Boolean, and natural number types. More complex inductive types — such as sum types, list types, and $\text{Fin } n$ — can be defined in terms of these primitives, and we give their encoding in Appendix A.

We use an *extensional* equality type $M =_X N$ with introduction form refl , but no elimination form. Instead we have the equality reflection rule which allows us to conclude a definitional equality $M =_X N$ from an arbitrary typal equality proof P . The usage of an extensional equality

$\frac{\Gamma \text{ ctx}}{\Gamma \vdash X \text{ type}}$	$\frac{}{\Gamma \vdash U \text{ type}}$	$\frac{}{\Gamma \vdash L \text{ type}}$	$\frac{}{\Gamma \vdash 1 \text{ type}}$	$\frac{}{\Gamma \vdash \perp \text{ type}}$	$\frac{}{\Gamma \vdash \text{Bool type}}$	$\frac{}{\Gamma \vdash \text{Nat type}}$
$\frac{\Gamma \vdash X \text{ type} \quad \Gamma, x : X \vdash Y \text{ type}}{\Gamma \vdash \prod_{x:X} Y \text{ type}}$			$\frac{\Gamma \vdash X \text{ type} \quad \Gamma, x : X \vdash Y \text{ type}}{\Gamma \vdash \sum_{x:X} Y \text{ type}}$			
$\frac{\Gamma \vdash M : U}{\Gamma \vdash [M] \text{ type}}$	$\frac{\Gamma \vdash A \text{ lin. type}}{\Gamma \vdash \uparrow A \text{ type}}$	$\frac{\Gamma \vdash X \text{ type} \quad \Gamma \vdash M : X \quad \Gamma \vdash N : X}{\Gamma \vdash M =_X N \text{ type}}$				
$\frac{\Gamma \vdash X \text{ type} \quad \Gamma \vdash Y \text{ type}}{\Gamma \vdash X \equiv Y \text{ type}}$	$\frac{\Gamma \vdash X \equiv Y : U}{\Gamma \vdash X \equiv Y \text{ type}}$	$\frac{\Gamma \vdash X \text{ lin. type}}{\Gamma \vdash \lfloor [X] \rfloor \equiv X \text{ lin. type}}$	$\frac{\Gamma \vdash X \text{ type}}{\Gamma \vdash X \text{ small}}$			
$\frac{\Gamma \vdash X \text{ type}}{\Gamma \vdash M : X}$	$\frac{\Gamma \vdash X \text{ small}}{\Gamma \vdash [X] : U}$	$\frac{\Gamma \vdash A \text{ small lin.}}{\Gamma \vdash [A] : L}$	$\frac{\Gamma; \cdot \vdash e : A}{\Gamma \vdash e : \uparrow A}$	$\frac{\Gamma \vdash M \equiv N : X}{\Gamma \vdash \text{refl} : M =_X N}$		
$\frac{\Gamma \vdash M : X \quad \Gamma \vdash N : X}{\Gamma \vdash M \equiv N : X}$			$\frac{\Gamma \vdash P : M =_X N}{\Gamma \vdash M \equiv N : X}$			

Fig. 7. Non-linear Formation and Typing Rules (selection)

type matches our implementation, which interprets both judgmental and typal equality as Cubical Agda’s Path type, and so naturally supports the equality reflection rule. Extensional equality makes type checking of our syntax as such undecidable [Hofmann 1997] because the conversion rule may require an arbitrarily complex equality proof with no explicit proof term. However, in our Agda implementation we must provide all of these equalities manually, so the extensionality does not raise any issues. This makes Lambek^D into an extensional theory, supporting function extensionality and the uniqueness of identity proofs. The development could be ported to an intensional type theory in the future, possibly requiring the use of setoids to handle function extensionality.

Lastly, we include a non-linear type $\uparrow A$ where A is a linear type. The intuition for this type is that its elements are the linear terms that are “resource free”: its introduction rule says we can construct an $\uparrow A$ when we have a linear term of type A with no free linear variables. Semantically, this is the type of parses of the empty string. This type is used extensively in our examples, playing a similar role to the $!$ modality of ordinary linear logic or the persistence modality \Box of separation logic [Girard 1987; Jung et al. 2016].

3.2 Linear Typing

We give the rules for linear type formation in Fig. 8 and the definition of linear terms in Fig. 9. The equational theory for these types is straightforward $\beta\eta$ equivalence and is included in Fig. 22 of Appendix B.

First, the linear variable rule says that a linear variable can be used if it is the *only* variable in the context. Next, we cover the “multiplicative” connectives of non-commutative linear logic. The linear unit (I) and tensor product (\otimes) are standard for a non-commutative linear logic: when we construct a linear unit we cannot use any variables and when we construct a tensor product, the two sides must use disjoint variables, and the variables the left side of the product uses must be to

$\frac{\Gamma \text{ ctx}}{\Gamma \vdash A \text{ lin. type}}$	$\frac{}{\Gamma \vdash I \text{ lin. type}}$	$\frac{c \in \Sigma}{\Gamma \vdash 'c' \text{ lin. type}}$	$\frac{\Gamma \vdash A \text{ lin. type} \quad \Gamma \vdash B \text{ lin. type}}{\Gamma \vdash A \otimes B \text{ lin. type}}$
$\frac{\Gamma \vdash A \text{ lin. type} \quad \Gamma \vdash B \text{ lin. type}}{\Gamma \vdash A \multimap B \text{ lin. type}}$	$\frac{\Gamma \vdash A \text{ lin. type} \quad \Gamma \vdash B \text{ lin. type}}{\Gamma \vdash A \multimap B \text{ lin. type}}$	$\frac{\Gamma, x : X \vdash A \text{ lin. type}}{\Gamma \vdash \bigoplus_{x:X} A \text{ lin. type}}$	
$\frac{\Gamma, x : X \vdash A \text{ lin. type}}{\Gamma \vdash \bigotimes_{x:X} A \text{ lin. type}}$	$\frac{\Gamma \vdash f : \uparrow (A \multimap B) \quad \Gamma \vdash g : \uparrow (A \multimap B)}{\Gamma \vdash \{a \mid f a = g a\} \text{ lin. type}}$	$\frac{\Gamma \vdash M : L}{\Gamma \vdash [M] \text{ lin. type}}$	$\frac{\Gamma \vdash A \text{ lin. type}}{\Gamma \vdash A \text{ small lin.}}$
$\frac{\Gamma \vdash A \text{ lin. type} \quad \Gamma \vdash B \text{ lin. type}}{\Gamma \vdash A \equiv B \text{ lin. type}}$	$\frac{\Gamma \vdash A \equiv B : L}{\Gamma \vdash A \equiv B \text{ lin. type}}$	$\frac{\Gamma \vdash A \text{ lin. type}}{\Gamma \vdash [A] \equiv A \text{ lin. type}}$	

Fig. 8. Linear Type Formers, Type Equivalence

the left in the context of the variables used by the right side of the tensor product. The elimination rules for unit and tensor are given by pattern matching. The pattern matching rules split the linear context into three pieces $\Delta_1, \Delta_2, \Delta_3$: the middle Δ_2 is used by the scrutinee of the pattern match, and in the continuation this context is replaced by the variables brought into scope by the pattern match. This ensures that pattern matches maintain the proper ordering of resource usage.

Because we are non-commutative, there are two function types: $A \multimap B$ and $B \multimap A$, which have similar λ introduction forms and application elimination forms. The difference between these is that the introduction rule for $A \multimap B$ adds a variable to the right side of the context, whereas the introduction rule for $B \multimap A$ adds a variable to the left side of the context. In our experience, because by convention parsing algorithms parse from left-to-right, we rarely need to use the $B \multimap A$ connective. As we have already seen, the \multimap connective is frequently used in conjunction with the \uparrow connective so that we can abstract non-linearly over linear functions.

Next, we cover the “additive” connectives. First, we use the non-linear types to define *indexed* versions of the additive disjunction \oplus and additive conjunction $\&$ of linear logic, which can be thought of as linear versions of the Σ and Π connectives of ordinary dependent type theory, respectively. The indexed $\&$ is defined by a λ that brings a *non-linear* variable into scope and eliminated using projection where the index specified is given by a non-linear term. The rules for indexed \oplus are analogous to a “weak” Σ type: it has an injection introduction rule σ , but its elimination rule is given by *pattern matching* rather than first and second projections. We can define the more typical nullary and binary versions of these connectives by using indexing over the empty and boolean type respectively. We will freely use \emptyset to refer to this empty disjunction and \top to refer to the empty conjunction, and use infix $\oplus/\&$ for binary disjunction/conjunction.

Lastly, we include a type $\{a \mid f a = g a\}$ that we call the *equalizer* of linear functions f and g . We think of this type as the “subtype” of elements of A that satisfy the equation $f a \equiv g a$. Note that it is important here that f, g themselves are non-linearly used functions, as linear values cannot be used in a type. Equalizer types are not needed for non-linear types since they can be constructed using the equality type as $\sum_{x:X} f x =_Y g x$, but this construction cannot be used for linear types because it uses a *dependent* version of the equality type, which we cannot define as a linear type. While the equalizer type is not used directly in defining any of our parsers or formal grammars, it is used for several proofs, allowing for inductive arguments about our indexed inductive types.

In addition to these type-theoretic principles, we need two additional axioms that do not generally hold in systems based on linear logic. First, we need that additive conjunction *distributes* over additive disjunction — e.g., in the finitary case that $0 \& A \cong 0$ and $(A + B) \& C \cong (A \& C) + (B \& C)$. More generally, we assume Axiom 3.1.

AXIOM 3.1 (Distributivity). For any $A : (x : X) \rightarrow Y(x) \rightarrow L$, the definable function distributing conjunction over disjunction
$$f : \bigoplus_{x : X} Y(x) \rightarrow \&_{x : X} A x \rightarrow \&_{x : X} \bigoplus_{y : Y(x)} A x y$$
 has an inverse.

The following corollary is a well known consequence of distributivity [Cockett 1993], which we use in Lemma 4.7 to prove that unambiguous binary sums have unambiguous summands.

COROLLARY 3.2. Distributivity implies that the constructors $\text{inl} : A \rightarrow A \oplus B$, $\text{inr} : B \rightarrow A \oplus B$ of a binary sum are injective — i.e. if $\text{inl } a \equiv \text{inl } a'$, then $a \equiv a'$.

Our primary use of distributivity is to define an equivalence that expresses a linear type A as a sum over which character it starts with, if any, $A \cong (A \& I) \oplus \bigoplus_{c : \Sigma_0} (A \& ('c' \otimes \top))$. We use this equivalence when building a parser for the traces of the lookahead automaton in Fig. 15.

Second, we need that the different constructors of \bigoplus are disjoint. That is, we want to enforce that $\sigma x a$ and $\sigma x' a'$ of type $\bigoplus_{x : X} A x$ are not equal whenever $x \neq x'$. However, because these are linear terms we cannot state their disequality directly. Instead, we encode the disequality via a function out of an equalizer,

AXIOM 3.3 (σ -Disjointness). For any $A : X \rightarrow L$ and $x \neq x' : X$ there is a function,

$$\uparrow (\{b \mid (\sigma x \circ \pi_1) b = (\sigma x' \circ \pi_2) b\} \rightarrow 0)$$

where $b : A(x) \& A(x')$. That is, the grammar of pairs of an $a : A(x)$ and an $a' : A(x')$ such that $\sigma x a = \sigma x' a'$ is empty.

We use σ -disjointness in Lemma 4.7 to prove that unambiguous sums have disjoint summands.

3.3 Indexed Inductive Linear Types

Next, we introduce the most complex and important linear type constructors of our development, *indexed inductive linear types*. We encode these by adding a mechanism for constructing initial algebras of strictly positive functorial type expressions, following prior work on inductive types [Altenkirch et al. 2015; Nakov and Nordvall Forsberg 2022]. The syntax is given in Figure 10. First, we add a non-linear type $\text{SPF } X$ of *strictly positive functorial* linear type expressions indexed by a non-linear type X . We think of the elements of this type as syntactic descriptions of linear types that are parameterized by X -many variables standing for linear types that are only used in strictly positive positions. Accordingly, the $\text{SPF } X$ type supports an operation el that interprets it as such a type constructor, as well as an operator map that defines a functorial action on parse transformers. The $\text{SPF } X$ type supports constructors for a reference $\text{Var } x$ to one of the linear type variables, a constant expression that does not mention any type variables K , as well as tensor products and additive conjunction and disjunction of type expressions. Further, we add equations in Fig. 17 of Appendix A that say that the el/map operations correspond to these descriptions of the constructors.

Next, given a family of X -many strictly positive linear type expressions $F : X \rightarrow \text{SPF } X$, we define a family $\mu F : X \rightarrow L$ of X -many mutually recursive inductive types. The introduction rule for this is roll , which constructs an element of $\mu F x$ from the one-level of the x th type expression. The elimination principle is defined by a mutual fold operation: given a family of output types A indexed by X , we can define a family of functions from $\mu F x \rightarrow A x$ if you specify how to interpret all of the constructors as operations on A values. We add $\beta\eta$ equations that specify that this makes the family

$\frac{\Gamma \vdash \Delta \text{ lin. ctx.} \quad \Gamma \vdash A \text{ lin. type}}{\Gamma; \Delta \vdash e : A}$	$\frac{}{\Gamma; a : A \vdash a : A}$	$\frac{\Gamma \vdash M : \uparrow A}{\Gamma; \cdot \vdash M : A}$	$\frac{\Gamma; \Delta \vdash e : B \quad \Gamma \vdash A \equiv B \text{ lin. type}}{\Gamma; \Delta \vdash e : A}$
$\frac{}{\Gamma; \cdot \vdash () : I}$		$\frac{\Gamma; \Delta_2 \vdash e : I \quad \Gamma; \Delta_1, \Delta_3 \vdash e' : C}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{let } () = e \text{ in } e' : C}$	
$\frac{\Gamma; \Delta \vdash e : A \quad \Gamma; \Delta' \vdash e' : B}{\Gamma; \Delta, \Delta' \vdash (e, e') : A \otimes B}$		$\frac{\Gamma; \Delta_2 \vdash e : A \otimes B \quad \Gamma; \Delta_1, a : A, b : B, \Delta_2 \vdash e' : C}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{let } (a, b) = e \text{ in } e' : C}$	
$\frac{\Gamma; \Delta, a : A \vdash e : B}{\Gamma; \Delta \vdash \lambda^{\circ} a. e : A \multimap B}$		$\frac{\Gamma; \Delta \vdash e : A \multimap B \quad \Gamma; \Delta' \vdash e' : A}{\Gamma; \Delta, \Delta' \vdash e e' : B}$	
$\frac{\Gamma; a : A, \Delta \vdash e : B}{\Gamma; \Delta \vdash \lambda^{\circ} a. e : B \multimap A}$		$\frac{\Gamma; \Delta \vdash e : A \quad \Gamma; \Delta' \vdash e' : B \multimap A}{\Gamma; \Delta, \Delta' \vdash e' \multimap e : B}$	
$\frac{\Gamma, x : X; \Delta \vdash e : A}{\Gamma; \Delta \vdash \lambda^{\&}_x. e : \&_{x:X} A}$		$\frac{\Gamma; \Delta \vdash e : \&_{x:X} A \quad \Gamma \vdash M : X}{\Gamma; \Delta \vdash e. \pi M : A\{M/x\}}$	
$\frac{\Gamma \vdash M : X \quad \Gamma; \Delta \vdash e : A\{M/x\}}{\Gamma; \Delta \vdash \sigma M e : \bigoplus_{x:X} A}$		$\frac{\Gamma; \Delta_2 \vdash e : \bigoplus_{x:X} A \quad \Gamma, x : X; \Delta_1, a : A, \Delta_3 \vdash e' : C}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{let } \sigma x a = e \text{ in } e' : C}$	
$\frac{\Gamma; \Delta \vdash e : A \quad \Gamma; \Delta \vdash f e \equiv g e}{\Gamma; \Delta \vdash \langle e \rangle : \{a \mid f a = g a\}}$		$\frac{\Gamma; \Delta \vdash e : \{a \mid f a = g a\}}{\Gamma; \Delta \vdash e. \pi : A}$	

Fig. 9. Linear terms

μF into an *initial algebra* for the functor $\text{el}(F)$. That is, the β rule says that a fold applied to a roll is equivalent to mapping the fold over all the sub-expressions, which means that fold interprets all of the constructors homomorphically using the provided interpretation f . Then the η rule says that fold is the *unique* such homomorphism, i.e. anything that satisfies the recurrence equation of the fold is equal to it.

This definition as an initial algebra is well-understood semantically but the η principle in particular is somewhat cumbersome to use directly in proofs. In dependent type theory, we would have a dependent *elimination* principle, which can be used to implement functions by recursion as well as proofs by induction. Unfortunately, since linear types do not support dependency on linear types, we cannot directly adapt this approach. However, if we are trying to prove that two morphisms out of a mutually recursive type are equal, we can use the *equalizer* type to prove their equality by induction. That is, if our goal is to prove two functions $f, g : \uparrow (\mu F x \multimap A x)$ equal, it suffices to implement a function $\text{ind} : \uparrow (\mu F x \multimap \{a \mid f a = g a\})$ such that $\text{ind}(a) \equiv a$. Then an inductive-style proof can be implemented by constructing ind using a fold. This can all be justified using only the $\beta\eta$ principles for equalizers and inductive types, and this is how our most complex inductive proofs are implemented in the Agda formalization.

3.4 Grammar-specific Additions

So far, our calculus is a somewhat generic combination of dependent types with non-commutative linear types. In order to carry out formal grammar theory and define parsers, we need only add a few grammar-specific constructions.

$$\begin{array}{c}
 \frac{\Gamma \vdash X \text{ type}}{\Gamma \vdash \text{SPF } X \text{ type}} \qquad \frac{\Gamma \vdash X \text{ small}}{\Gamma \vdash \text{SPF } X \text{ small}} \qquad \text{el} : \prod_{x:U} \text{SPF } X \rightarrow (X \rightarrow L) \rightarrow L \\
 \\
 \text{map} : \prod_{x:U} \prod_{F:\text{SPF } X \rightarrow L} \prod_{B:X \rightarrow L} \left(\prod_{x:X} \uparrow ([A x] \multimap [B x]) \right) \rightarrow \uparrow ([\text{el}(F)(A)] \multimap [\text{el}(F)(B)]) \qquad \text{Var} : \prod_{x:U} X \rightarrow \text{SPF } X \\
 \\
 K : \prod_{x:U} L \rightarrow \text{SPF } X \qquad \bigoplus : \prod_{x:U} \prod_{Y:U} (Y \rightarrow \text{SPF } X) \rightarrow \text{SPF } X \qquad \& : \prod_{x:U} \prod_{Y:U} (Y \rightarrow \text{SPF } X) \rightarrow \text{SPF } X \\
 \\
 \otimes : \prod_{x:U} \text{SPF } X \rightarrow \text{SPF } X \rightarrow \text{SPF } X \qquad \text{roll} : \prod_{x:U} \prod_{F:X \rightarrow \text{SPF } X} \prod_{x:X} \uparrow (\text{el}(F x)(\mu F)) \\
 \\
 \text{fold} : \prod_{x:U} \prod_{F:X \rightarrow \text{SPF } X} \prod_{A:X \rightarrow L} \left(\prod_{x:X} \uparrow ([\text{el}(F x)([A])] \multimap [A x]) \right) \rightarrow \prod_{x:X} \uparrow (\mu F x \multimap A x) \\
 \\
 \frac{\Gamma \vdash f : \prod_{x:X} \uparrow (\text{el}(F x)(A) \multimap A x) \quad \Gamma; \Delta \vdash e : \text{el}(F x)(\mu F)}{\Gamma; \Delta \vdash \text{fold } F f x (\text{roll } e) \equiv f x (\text{map}(F x) (\text{fold } F f)) : A x} \text{IND}\beta \\
 \\
 \frac{\Gamma \vdash f : \prod_{x:X} \uparrow (\text{el}(F x)(A) \multimap A x) \quad \Gamma \vdash e : \prod_{x:X} \uparrow (\mu F x \multimap A x)}{\Gamma, x : X; a : \text{el}(F x)(\mu F) \vdash e x (\text{roll } a) \equiv f x (\text{map}(F x) e) : A x} \text{IND}\eta \\
 \\
 \text{fold } F f \equiv e' : \prod_{x:X} \uparrow (\mu F x \multimap A x)
 \end{array}$$

Fig. 10. Strictly positive functors and indexed inductive linear types

Lambek^D is parameterized by a fixed, finite alphabet Σ from which we build our strings. For each character $c \in \Sigma$, we add a corresponding linear type ' c '. We can then define a non-linear type Char as the disjunction of all of these characters, and define a type String as the Kleene star of Char , i.e. as an inductive linear type. Then we add a function $\text{read} : \uparrow (\top \multimap \text{String})$ that intuitively “reads” the input string from the input and makes it available. It is important that the input type of read is \top , which can control any amount of resources, and not I which controls no resources.

AXIOM 3.4. $\lambda s. \text{read}(! (s)) \equiv \lambda s. s$ where $!$ is the unique function $\uparrow (\text{String} \multimap \top)$.

If we have a string, but then throw it away and read it from the input, then we, in fact, recover the original string. This ensures that the elements of the String type always stand for the actual input string in our reasoning. In the next section, we will show how these basic principles are enough to provide a basis for verified parsing and formal grammar theory.

4 Formal Grammar Theory in Dependent Lambek Calculus

This section explores the applications of Lambek^D to formal grammar theory. We demonstrate that several classical notions and constructions integral to the theory of formal languages are faithfully represented in Dependent Lambek Calculus.

In the theory of formal grammars, there are two different notions of equivalence: up to weak generative capacity, meaning just which strings are accepted by the grammar; and up to *strong* generative capacity, when the parse trees of the two grammars is isomorphic [Chomsky 1963]. Using linear types as grammars, we can define both of these notions of equivalence in Lambek^D.

Definition 4.1. Grammars A and B are *weakly equivalent* if there exist parse transformers $f : \uparrow (A \multimap B)$ and $g : \uparrow (B \multimap A)$. A is a *retract* of B if they are weakly equivalent and $\lambda a. g(f(a)) \equiv \lambda a. a$. They are *strongly equivalent* if further the other composition is the identity, i.e., $\lambda b. f(g(b)) \equiv \lambda b. b$.

A formal grammar A is ambiguous if there are multiple parse trees for the *same* string w . For example, $a \oplus a$ is ambiguous because there are two parses of " a ", constructed using inl and inr . On the other hand, a formal grammar is unambiguous when there is at most one parse tree for any input string. We can capture this notion as a type in Lambek^D as follows:

Definition 4.2. A grammar A is *unambiguous* if for every linear type B , $f : \uparrow (B \multimap A)$, and $g : \uparrow (B \multimap A)$ then $f \equiv g$.

Definition 4.2 can be read more intuitively as stating that A is unambiguous if there is at most one way to transform parses of any other grammar B into parses of A . This notion of an unambiguous type is the analog for linear types of the definition of a (homotopy) *proposition* in the terminology of homotopy type theory [Univalent Foundations Program 2013]. The most basic unambiguous types are \top and \emptyset , and in a system of classical logic all unambiguous types would have to be equivalent to one of these, but with our axioms we can show also that I and literals ' c ' are unambiguous. To see this, first, we establish two useful properties of unambiguity.

LEMMA 4.3 (Lambek^D). *If B is unambiguous and A is a retract of B then A is unambiguous.*

LEMMA 4.4 (Lambek^D). *As a consequence of Corollary 3.2, if a binary disjunction $A \oplus B$ is unambiguous then A and B are each unambiguous.*

From Lemma 4.3, we can prove that String is unambiguous, since it is a retract of \top . In fact, observe that if A is a retract of B and B is unambiguous, then in fact A and B are strongly equivalent, as the equation $\lambda b. f(g(b)) \equiv \lambda b. b$ follows because B is unambiguous. Therefore String is also strongly equivalent to \top . Next, since String is defined as a Kleene star, we can easily show that $\text{String} \cong I \oplus \text{Char} \oplus (\text{Char} \otimes \text{Char} \otimes \text{String})$. Then by Lemma 4.4, we have that I and Char are unambiguous as well. Using the finiteness of the alphabet Σ and the unambiguity of Char , we have that each literal ' c ' is likewise unambiguous.

We now turn to our main task, which is using our linear type system to implement verified parsers. Given a grammar defined as a linear type A , a first attempt at defining a parser would be to implement a function $\uparrow (\text{String} \multimap A)$. But since our linear functions must be total, this means that we can construct an A parse for *every* input string, which is impossible for most grammars of interest. Instead we might try to write a partial function as a $\uparrow (\text{String} \multimap (A \oplus \top))$ using the "option" monad. This allows for the possibility that the input string does not parse, but is far too weak as a specification: we can trivially implement a parser for any type by always returning inr . The correct notion of a parser should be one that allows for failure, but only in the case that a parse cannot be constructed.

Definition 4.5. Linear types A and B are *disjoint* if there is a function $\uparrow (A \& B \multimap \emptyset)$.

Definition 4.6. A parser for a linear type A is the choice a type A_- , disjoint from A and function $\uparrow (\text{String} \multimap A \oplus A_-)$.

Here we replace \top in our partial parser type with a type A_- that we can think of as a negation of A . The function $\uparrow (A \& A_- \multimap \emptyset)$ ensures that it is impossible for A and A_- to parse the same input string. This means that in defining a parser, we will need to define a kind of negative grammar for strings that do not parse.

Fortunately, we will see that deterministic automata naturally support such a notion with no additional effort: the negative grammar is simply the grammar for traces that end in a rejecting state. This follows from the following principle, a consequence of Axiom 3.3.

LEMMA 4.7 (⌘). *If $\bigoplus_{x:\mathcal{X}} A x$ is unambiguous, then for $x \neq x'$, $A x$ and $A x'$ are disjoint. In particular, if the binary product $A \oplus A_{\neg}$ is unambiguous, then A and A_{\neg} are disjoint.*

PROOF. If $\bigoplus_{x:\mathcal{X}} A x$ is unambiguous, then all functions into it from $A x \& A x'$ are equal. In particular, $\sigma x \circ \pi_1 \equiv \sigma x' \circ \pi_2$ so there is a function $\uparrow (A x \& A x' \multimap \{b \mid (\sigma x \circ \pi_1) b = (\sigma x' \circ \pi_2) b\})$. We then compose with the function in Axiom 3.3 to prove that $A x$ and $A x'$ are disjoint. \square

Writing a parser as a linear term intrinsically verifies the *soundness* of the parser for free from the typing: any inl parse that we return *must* correspond to a parse tree of the input string. Further, if we verify the disjointness property of Definition 4.6 we then also get the *completeness* of the parser as well, that when the parser rejects the input that there are no valid parses.

Our main method for constructing verified parsers is to show that a grammar A is weakly equivalent to a grammar for a deterministic automaton. Parsers for deterministic automata are simple to implement by stepping through the states of the automaton, with the rejecting traces serving as the negative grammar. This is sufficient due to the following:

LEMMA 4.8 (⌘). *If A is weakly equivalent to B then any parser for A extends to a parser for B .*

Here we need both directions of the weak equivalence. We need $A \multimap B$ to extend the parser from $\text{String} \multimap A \oplus A_{\neg}$ to $\text{String} \multimap B \oplus A_{\neg}$, but then we also need $B \multimap A$ to establish that A_{\neg} is disjoint from B .

4.1 Regular Expressions and Finite Automata

Next, we describe how to construct an intrinsically verified parser for regular expressions by compiling it to an NFA and then a DFA. That is, for each regular expression A , we construct an NFA $N(A)$ and a corresponding DFA $D(A)$ such that A is strongly equivalent to the traces of $N(A)$ and weakly equivalent to the accepting traces of $D(A)$. Then we can easily construct a parser for traces of $D(A)$ and apply Lemma 4.8 to get a verified regular expression parser.

A regular expression in Lambek^D is a linear type constructed using only the connectives ' c ', \emptyset , \oplus , I , \otimes , and Kleene star. In Section 2, we saw one particular NFA and its corresponding type of traces. More generally, in Fig. 11 we define a linear type of traces through an arbitrary NFA N .

Trace_N is an inductive type indexed by the starting state of the trace $s : N.\text{states}$, and it may be built through one of three constructors. We may terminate a trace at an accepting state with the constructor nil . Here we use an Agda-style Unicode syntax for $\&$, as well as using the function arrow to mean a non-dependent version of $\&$. If we had a trace beginning at the destination state of a transition, then we may use the cons constructor to combine that trace with a parse of the label of the transition to build a trace beginning at the source of the transition. Finally, if we had a trace beginning at the destination of an ϵ -transition then we may use ϵcons to pull it back along the ϵ -transition and construct a trace beginning at the source of the ϵ -transition. As a shorthand, we write Parse_N for the accepting traces out of $N.\text{init}$.

Trace_D , the linear type of traces through D , is given next. Unlike traces for an NFA, we parameterize this type additionally by a boolean which says whether the trace is accepting or rejecting. These traces may be terminated in an accepting state s with the nil constructor. The cons constructor builds a trace out of state s by linearly combining a parse of some character c with a trace out of the state $D.\delta c s$. The trace built with cons is accepting if and only if the trace out of $D.\delta c s$ is accepting.

Because DFAs are deterministic, we are able to prove that their types of traces are unambiguous and define a parser for them directly. In particular we show that for any start state s , $\bigoplus_{b:\text{Bool}} \text{Trace}_D s b$ is a retract of String and apply Lemma 4.3 to derive unambiguity. That is, we first construct a function parse_D which is a parser for $\text{Trace}_D s \text{ true}$ with $\text{Trace}_D s \text{ false}$ being

```

data TraceN : (s : N .states) → L where
  nil : ↑(&[ s : N .states ] N .isAcc s → TraceN s)
  cons : ↑(&[ t : N .transitions ] ('N .label t' → TraceN (N .dst t)
                                     → TraceN (N .src t)))
  econs : ↑(&[ t : N .etransitions ] (TraceN (N .edst t) → TraceN (N .esrc t)))
data TraceD : (s : D .states) (b : Bool) → L where
  nil : ↑(&[ s : D .states ] TraceD s D .isAcc s)
  cons : ↑(&[ c : Σ ] &[ s : D .states ]
           &[ b : Bool ] ('c' → TraceD (D .δ c s) b → TraceD s b))

```

Fig. 11. Traces of an NFA N and a DFA D

```

parseD : ↑(String → &[ s : D .states ] ⊕[ b : Bool ] TraceD s b)
parseD String .nil s = σ (D .isAcc s) (TraceD .nil s)
parseD (String .cons (σ c a) w) s = let σ b t = parse w (D .δ c s) in
                                     σ b (TraceD .cons c s b a t)
printD : (s : D .states) → ↑((⊕[ b : Bool ] TraceD s b) → String)
printD s (σ b (TraceD .nil .s)) = String .nil
printD s (σ b (TraceD .cons c (D .δ c s) b a trace)) =
  String .cons (σ c a) (printD (D .δ c s) (σ b trace))

```

Fig. 12. Parser/printer for DFA traces

the disjoint type used, and disjointness follows from the unambiguity of $\bigoplus_{b:\text{Bool}} \text{Trace}_D s b$ by Lemma 4.4.

The parser, parse_D , is defined by recursion on strings in Fig. 12. If this string is empty, then parse_D terminates a trace at the input state s . If the string is nonempty, then parse_D walks forward in D from the input state s by the character at the head of the string. The inverse, print_D is defined by recursion on traces. If the trace is defined via nil , then print_D returns the empty string. Otherwise, if the trace is defined by cons then print_D appends the character from the most recent transition to the output string and recurses. We prove this is a retraction by induction on traces.

THEOREM 4.9 (\mathcal{U}). parse_D is a parser for $\text{Trace}_D s$ true.

Working backwards, we can then show the traces of an NFA are weakly equivalent to the traces of a DFA implementing a variant of Rabin and Scott’s classic powerset construction [Rabin and Scott 1959]. Here we note that this is *only* a weak equivalence and not a strong equivalence, as the DFA is unambiguous even if the NFA is not.

CONSTRUCTION 4.10 (Determinization, \mathcal{U}). Given an NFA N , we construct a DFA D such that Parse_N is weakly equivalent to Parse_D .

PROOF. Define the states of D to be $\mathbb{P}_\epsilon(N.\text{states})$ — the type of ϵ -closed² subsets of $N.\text{states}$. A subset is accepting in D if it contains an accepting state from N . The initial state of D is the ϵ -closure of $N.\text{init}$. Lastly, the transition function of D sends the subset X under the character c to the ϵ -closure of all the states reachable from X via a transition labeled with the character c .

We demonstrate the weak equivalence between Parse_N and Parse_D by constructing parse transformers between the two grammars. To build the parse transformer $\uparrow(\text{Parse}_N \multimap \text{Parse}_D)$, we strengthen our inductive hypothesis to quantify over every start state and build a term $N\text{to}D : \uparrow(\text{Trace}_N s \text{ true} \multimap \&_{X:D.\text{states}} \&_{s\text{In}X:X\>S} \text{Trace}_D X \text{ true})$ that maps a trace in N from an

²A subset of states X is ϵ -closed if for every $s \in X$ and ϵ -transition $s \xrightarrow{\epsilon} s'$ we have $s' \in X$.

arbitrary state s to a trace in D that may begin at any subset of states X that contains s . $NtoD$ may then be instantiated at $s = N.init$ and $X = D.init$ to get the desired parse transformer.

To construct a term from DFA traces to NFA traces, we similarly strengthen our inductive hypothesis and build a parse transformer that ranges over arbitrary ϵ -closed subsets $X : \mathbb{P}_\epsilon N.states$, $DtoN : \uparrow (\text{Trace}_D X \text{ true} \multimap \bigoplus_{s:N.states} \bigoplus_{s \text{ in } X : X \ni s} \text{Trace}_N s \text{ true})$. Because the states of D are ϵ -closed subsets of $N.states$, if two states s and s' belong to the same ϵ -closed subset $X : \mathbb{P}_\epsilon (N.states)$ then there exists a ϵ -path in N between the two, but we do not necessarily know which one. Similarly, the data contained in the type $\text{Trace}_D X \text{ true}$ is the *existence* of an accepting trace in N beginning at some state in D .

To define $DtoN$, we need a choice function that extracts out a trace in N from the mere existence of one. We achieve this by choosing the smallest trace through N subject to an ordering on the non-linear types $N.states$, $N.transitions$, and $N.\epsilon transitions$. In essence, the given orderings specify a global disambiguation strategy. \square

Finally, given any regular expression we can construct a *strongly* equivalent NFA. While only weak equivalence is required to construct a parser, proving the strong equivalence shows that other aspects of formal grammar theory are also verifiable in Lambek^D.

CONSTRUCTION 4.11 (Thompson's Construction [Thompson 1968], \hookrightarrow). Given a regular expression R , we build an NFA N such that R is strongly equivalent to $\text{Trace}_N (N.init)$.

COROLLARY 4.12 (\hookrightarrow). We may build a parser for every regular expression R .

PROOF. We combine the strong equivalence of Construction 4.11 with the weak equivalence of Construction 4.10 to show that R is weakly equivalent to the traces of its determinized automaton. Then, we use Lemma 4.8 to extend the parser from Theorem 4.9 with respect to this weak equivalence. \square

4.2 Context-free grammars

Next, we give two examples for parsing context-free grammars (CFGs). CFGs can be encoded in our type theory similarly to regular expressions, as CFGs are equivalent to the formalism of μ -regular expressions, where the Kleene star is replaced by an arbitrary fixed point operation [Leiß 1992].

A simple example of a CFG is the Dyck grammar of balanced parentheses, which we define in Figure 13. Dyck is a grammar over the alphabet $\{"(", ")\}"$. The `nil` constructor shows that the empty string is balanced, and the `bal` constructor builds a balanced parse by wrapping an already balanced parse in an additional set of parentheses then following it with another balanced parse. We construct a parser for Dyck by building a deterministic automaton M such that Parse_M is strongly equivalent to Dyck.

We implement the parser for the Dyck language using an *infinite* state deterministic automaton, in Figure 14. Here the state is a “stack” counting how many open parentheses have been seen so far. Functions `parseM` and `printM` for this automaton can be defined analogously to those for DFAs, and so $\bigoplus_{s:M.states} \bigoplus_{b:Bool} \text{Trace}_M s b$ is likewise unambiguous.

THEOREM 4.13 (\hookrightarrow). Dyck and Parse_M are strongly equivalent, so we may build a parser for Dyck.

```
data Dyck : L where
  nil : ↑ Dyck
  bal : ↑(' ( → Dyck → ') ' → Dyck → Dyck)
```

Fig. 13. The Dyck grammar as an inductive linear type

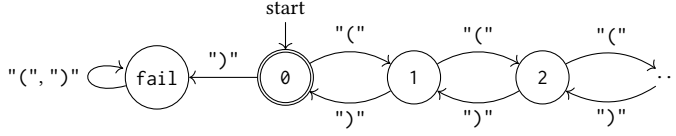


Fig. 14. Automaton M for the Dyck grammar

Our final example is of a simple grammar of arithmetic expressions with an associative operation. Here we take the alphabet to be $\{ "(", ")", "+", "NUM" \}$. In Figure 15 we define the expression grammar using two mutually recursive types, corresponding to the two non-terminals we would use in a CFG syntax. The syntactic structure encodes that the binary operation is right associative. In the same figure, we define the traces of an automaton with one token of lookahead. The automaton³ has four different “states”, each with access to a natural number “stack”. The “opening” state 0 expects either a left paren, in which case it increments the stack and stays in the opening state, or sees a number and proceeds to the D state. The “done opening” state D is where lookahead is used: if the next token will be a right paren, then we proceed to C; otherwise, we proceed to A. In the “closing” state C if we observe a right paren, then we decrement the count and continue to the D state. In the “adding” state A, we succeed if the string ends and the count is 0; otherwise, if we see a plus we continue to the 0 state. Additionally, since the automaton need parse all of the incorrect strings, we add all of the failing cases.

THEOREM 4.14 (👉). We construct a parser for Exp by showing it is weakly equivalent to $O \emptyset \text{ true}$.

With Axiom 3.1, it is straightforward to implement a parser for this lookahead automaton, generalizing the approach for deterministic automata. Without access to distributivity, we may define a rudimentary lookahead operation via the chain of equivalences,

$$A \text{ n } b \cong (A \text{ n } b) \& \text{T} \cong (A \text{ n } b) \& ((')' \otimes \text{T}) \oplus \text{NotStartsWithRP}$$

When defining a parser, if the lookahead character is a right paren we would like to apply lookAheadRP ; otherwise, apply lookAheadNot . However, without distributivity we have no means of relating the bit of information learned by looking ahead to the input $A \text{ n } b$ parse. That is, to choose which constructor to apply we would need the end of this chain of equivalences to be a binary sum rather than a binary product, necessitating the distributivity axiom.

4.3 Unrestricted Grammars

While we have shown only examples for context-free grammars, in fact arbitrarily complex grammars are encodable in Lambek^D. To demonstrate this, we show that for any *non-linear* function $P : \mathbf{String} \rightarrow U$, where here **String** is the *non-linear* type of strings over the alphabet, we can construct a grammar whose parses correspond to P . Reify $P = \bigoplus_{w:\mathbf{String}} \bigoplus_{x:P \ w} [w]$ where $["] = I$ and $[c :: w] = 'c' \otimes [w]$.

This reification operation on functions $\mathbf{String} \rightarrow U$ is very expressive, as it sidesteps our linear typing connectives and utilizes the whole of nonlinear dependent type theory to define a grammar. For example, given a Turing machine T one may define a non-linear predicate $\text{accepts} : \mathbf{String} \rightarrow U$ that encodes that T halts and accepts an input string. Then, Reify accepts is a linear type that captures precisely the string the same language as T . In general, $\text{Lang}(\text{Reify accepts})$ is recursively enumerable — the most general class of languages in the Chomsky hierarchy.

³In the automaton definition, NotStartsWithLP is defined as $I \oplus ('(' \oplus '+' \oplus \text{'NUM'}) \otimes \text{T})$. Similarly, NotStartsWithRP is defined as $I \oplus ('(' \oplus '+' \oplus \text{'NUM'}) \otimes \text{T})$.


```

data Exp : L where
  done : ↑(Atom → Exp)
  add : ↑(Atom → '+' → Exp → Exp)
data Atom : L where
  num : ↑('NUM' → Atom)
  parens : ↑('(' → Exp → ')') → Atom

data O : Nat → Bool → L where
  left : ↑(&[ n : Nat ] &[ b : Bool ] '(' → O (n + 1) b → O n b)
  num : ↑(&[ n : Nat ] &[ b : Bool ] 'NUM' → O n b → O n b)
  unexpected : ↑(&[ n : Nat ] (NotStartsWithLP → O n false))
data D : Nat → Bool → L where
  lookAheadRP : ↑(&[ n : Nat ] &[ b : Bool ] ((')' ⊗ T) & C n b) → D n b)
  lookAheadNot : ↑(&[ n : Nat ] &[ b : Bool ] (NotStartsWithRP & A n b) → D n b)
data C : Nat → Bool → L where
  closeGood : ↑(&[ n : Nat ] &[ b : Bool ] ')') → D n b → C (n + 1) b)
  closeBad : ↑('(') → C 0 false)
  unexpected : ↑(&[ n : Nat ] NotStartsWithRP → C n false)
data A : Nat → Bool → L where
  doneGood : ↑(A 0 true)
  doneBad : ↑(&[ n : Nat ] A (n + 1) false)
  add : ↑(&[ n : Nat ] &[ b : Bool ] '+' → O n b → A n b)
  unexpected : ↑(&[ n : Nat ] (('' ⊕ ')') ⊕ 'NUM') → T → A n false)

```

Fig. 15. Associative arithmetic expressions and a corresponding lookahead automaton

CONSTRUCTION 4.15 (). For any Turing machine T , we can construct a grammar in Lambek^D that accepts the same language as T .

5 Denotational Semantics and Implementation

To justify our assertion that Lambek^D is a syntax for formal grammars and parse transformers, we will now define a *denotational semantics* that makes this mathematically precise by defining a notion of formal grammar and parse transformer then showing that our type theory can be soundly interpreted in this model. We then discuss how this denotational semantics provides the basis for our prototype implementation in Agda.

5.1 Formal Grammars and Parse Transformers

The most common definition of a formal grammar is as generative grammars, defined by a set of non-terminals, a specified start symbol and set of production rules. We instead use a more abstract formulation that is closer in spirit to the standard definition of a formal *language* [Elliott 2021]:

Definition 5.1. A *formal language* L is a function from strings to propositions. A (small) *formal grammar* A is a function from strings to (small) sets.

We think of the grammar A as taking a string to the set of all parse trees for that string. However since A could be any function whatsoever there is no requirement that an element of $A(w)$ be a “tree” in the usual sense. This definition provides a simple, syntax-independent definition of a grammar that can be used for any formalism: generative grammars, categorial grammars, or our own type-theoretic grammars. Note that the definition of a formal grammar is a generalization of the usual notion of formal language since a proposition can be equivalently defined as a subset of a one-element set. Then the difference between a formal grammar and a formal language is that formal grammars can be *ambiguous* in that there can be more than one parse of the same string.

Even for unambiguous grammars, we care not just about *whether* a string has a parse tree, but *which* parse tree it has, i.e., what the structure of the element of $A(w)$ is. To interpret our universes U, L we assume we have a universe of *small* sets. In the remainder, all formal grammars are assumed to be small.

We then interpret linear *terms* as *parse transformers*:

Definition 5.2. Let A_1, A_2 be formal grammars. Then a parse transformer f from A_1 to A_2 is a function assigning to each string w a function $f_w : A_1(w) \rightarrow A_2(w)$.

Just as formal grammars generalize formal languages, parse transformers generalize formal language inclusion: if $A_1(w), A_2(w)$ are all subsets of a one-element set, then a parse transformer is equivalent to showing that $A_1(w) \subseteq A_2(w)$. In our denotational semantics, linear terms will be interpreted as such parse transformers, and the notions of unambiguous grammar, parsers, disjointness, etc, introduced in Section 4 can be verified to correspond to their intended meanings under this interpretation.

Parse transformers can be composed: given two parse transformers f and g , their composition is defined pointwise, i.e. $(f \circ g)_w = f_w \circ g_w$. Furthermore, given a formal grammar A , its identity transformer is $\text{id}_w = \text{id}_{A(w)}$, where $\text{id}_{A(w)}$ is the identity function on the set $A(w)$. This defines a *category*.

Definition 5.3. Define **Gr** to be the category whose objects are formal grammars and morphisms are parse transformers.

This category is equivalent to the slice category **Set**/ Σ^* and is very well-behaved. It is complete, co-complete, Cartesian closed and carries a monoidal biclosed structure. We will use these structures to model the linear types, terms and equalities in Lambek^D .

5.2 Semantics

We now define our denotational semantics.

Definition 5.4 (Grammar Semantics). We define the following interpretations by mutual recursion on the judgments of Lambek^D :

- (1) For each non-linear context Γ ctx, we define a set $\llbracket \Gamma \rrbracket$.
- (2) For each non-linear type $\Gamma \vdash X$ type, and element $\gamma \in \llbracket \Gamma \rrbracket$, we define a set $\llbracket X \rrbracket \gamma$.
- (3) For each linear type $\Gamma \vdash A$ lin. type and element $\gamma \in \llbracket \Gamma \rrbracket$, we define a formal grammar $\llbracket A \rrbracket \gamma$. We similarly define a formal grammar $\llbracket \Delta \rrbracket \gamma$ for each linear context $\Gamma \vdash \Delta$ lin. ctx..
- (4) For each non-linear term $\Gamma \vdash M : X$ and $\gamma \in \llbracket \Gamma \rrbracket$, we define an element $\llbracket M \rrbracket \gamma \in \llbracket X \rrbracket \gamma$.
- (5) For each linear term $\Gamma; \Delta \vdash e : A$ and $\gamma \in \llbracket \Gamma \rrbracket$, we define a parse transformer from $\llbracket \Delta \rrbracket \gamma$ to $\llbracket A \rrbracket \gamma$.

And we verify the following conditions:

- (1) If $\Gamma \vdash X$ small, then $\llbracket X \rrbracket \gamma$ is a small set.
- (2) If $\Gamma \vdash X \equiv X'$ then for every γ , $\llbracket X \rrbracket \gamma = \llbracket X' \rrbracket \gamma$.
- (3) If $\Gamma \vdash A \equiv A'$ then for every γ , $\llbracket A \rrbracket \gamma = \llbracket A' \rrbracket \gamma$.
- (4) If $\Gamma \vdash M \equiv M' : X$ then for every γ , $\llbracket M \rrbracket \gamma = \llbracket M' \rrbracket \gamma$.
- (5) If $\Gamma; \Delta \vdash e \equiv e' : A$ then for every γ , $\llbracket e \rrbracket \gamma = \llbracket e' \rrbracket \gamma$.

The interpretation of dependent types as sets is standard [Hofmann 1997]. We present the concrete descriptions of the semantics of linear types, as well as our non-standard non-linear types in Figure 16. The grammar for a literal c has a single parse precisely when the input string consists of the single character. The grammar for the unit similarly has a single parse for the empty string. A parse of the tensor product $A \otimes B$ consists of a *splitting* of the empty string into a prefix w_1 and suffix

$$\begin{array}{ll}
\llbracket c \rrbracket \gamma w = \{c \mid w = c\} & \llbracket A \otimes B \rrbracket \gamma w = \{(w_1, w_2, a, b) \mid w_1 w_2 = w \wedge a \in \llbracket A \rrbracket \gamma w_1 \wedge b \in \llbracket B \rrbracket \gamma w_2\} \\
\llbracket I \rrbracket \gamma w = \{() \mid w = \epsilon\} & \llbracket \bigoplus_{x:X} A \rrbracket \gamma w = \prod_{x \in \llbracket X \rrbracket \gamma} \llbracket A \rrbracket (\gamma, x) w \\
\llbracket \uparrow A \rrbracket \gamma = \llbracket A \rrbracket \gamma \epsilon & \llbracket \&_{x:X} A \rrbracket \gamma w = \prod_{x \in \llbracket X \rrbracket \gamma} \llbracket A \rrbracket (\gamma, x) w \\
\llbracket A \multimap B \rrbracket \gamma w = \prod_{w'} \llbracket A \rrbracket \gamma w' \rightarrow \llbracket B \rrbracket \gamma w w' & \llbracket \{a \mid f a = g a\} \rrbracket \gamma w = \{a \in \llbracket A \rrbracket \gamma w \mid \llbracket f \rrbracket \gamma w a = \llbracket g \rrbracket \gamma w a\} \\
\llbracket B \multimap A \rrbracket \gamma w = \prod_{w'} \llbracket A \rrbracket \gamma w' \rightarrow \llbracket B \rrbracket \gamma w' w & \llbracket L \rrbracket \gamma = \mathbf{Gr}_0 \\
\llbracket \text{el}(F) \rrbracket \gamma G = \llbracket F \rrbracket \gamma G & \llbracket \text{SPF } X \rrbracket \gamma = \text{DepPolyFunc}(\llbracket X \rrbracket \gamma \times \Sigma^*, \Sigma^*) \\
\llbracket \text{map}(F) \rrbracket \gamma f = \llbracket F \rrbracket \gamma f & \\
\llbracket \mu A \rrbracket \gamma = \mu(\llbracket A \rrbracket \gamma) &
\end{array}$$

Fig. 16. Grammar Semantics

w_2 along with an A parse of w_1 and B parse of w_2 . A parse of $\bigoplus_{x:X} A$ is a pair of an element of the set X and a parse of $A(x)$, while dually a parse of $\&_{x:X} A$ is a *function* taking any $x : X$ to a parse of $A(x)$. A w -parse of $A \multimap B$ is a function that takes an A parse of some other string w' to a B parse of ww' , and $B \multimap A$ is the same except the B parse is for the reversed concatenation $w'w$. The set $\uparrow A$ is the set of parse for the empty string for A . This definition means that $\llbracket \uparrow (A \multimap B) \rrbracket$ (or $\llbracket \uparrow (B \multimap A) \rrbracket$) is equivalent to the set of parse transformers: $\llbracket \uparrow (A \multimap B) \rrbracket \gamma = \llbracket A \multimap B \rrbracket \gamma \epsilon = \prod_{w'} \llbracket A \rrbracket \gamma w' \rightarrow \llbracket B \rrbracket \gamma w' w$.

Next, a parse in the equalizer $\{a \mid f a = g a\}$ is defined as a parse in $\llbracket A \rrbracket$ that is mapped to the same parse by the parse transformers $\llbracket f \rrbracket$ and $\llbracket g \rrbracket$. The universe L of linear types is interpreted as the set of all small grammars. The most complex part of the semantics is the interpretation of strictly positive functors and indexed inductive linear types. We interpret a strictly positive functor as a *dependent polynomial functor* on the category of sets, also sometimes called an *indexed container* [Altenkirch et al. 2015; Gambino and Hyland 2003].

Definition 5.5. Let I and O be sets. A (dependent) *polynomial* (of sets) from I to O consists of a set of shapes S , a set of positions P and functions $f : P \rightarrow I$, $g : P \rightarrow S$ and $h : S \rightarrow O$. The *extension* of a polynomial is a functor $\mathbf{Set}/I \rightarrow \mathbf{Set}/O$ defined as the composite

$$\mathbf{Set}/I \xrightarrow{f^*} \mathbf{Set}/P \xrightarrow{\Pi_g} \mathbf{Set}/S \xrightarrow{\Sigma_h} \mathbf{Set}/O$$

Where f^* is the pullback functor along f ; Π_g is the dependent product operation and Σ_h is the dependent sum operation, which are, respectively, the right and left adjoint of their pullback functors g^* and h^* . A dependent polynomial functor from I to O is a functor $\mathbf{Set}/I \rightarrow \mathbf{Set}/O$ that is naturally isomorphic to the extension of a dependent polynomial from I to O .

With this interpretation of F as a polynomial functor, $\text{el}(F)$ and $\text{map}(F)$ are just interpreted as the action of the functor on objects and morphisms, respectively. We interpret the constructors K , Var , etc. on functors in the obvious way that matches the definitional behavior of el and map . The non-trivial part of the construction is verifying that such constructions are closed under being polynomial. The details are tedious but straightforward extension of prior work on dependent polynomials and indexed containers and we have verified the construction in Agda.

We use dependent polynomials functors on sets as these are guaranteed to have initial algebras. Further, these initial algebras are readily constructed in our Agda implementation as an inductive type of IW trees which are already available in the Cubical library of Agda [The Agda Community 2024]. Then an element $F \in \llbracket X \rightarrow \text{SPF } X \rrbracket \gamma$ is an $\llbracket X \rrbracket \gamma$ -indexed family of polynomial functors from $\llbracket X \rrbracket \gamma \times \Sigma^*$ to Σ^* , and taking the product of these constructs a polynomial functor from $\llbracket X \rrbracket \gamma \times \Sigma^*$ to

itself. Then $\llbracket \mu F \rrbracket \gamma$ is defined to be the initial algebra of this functor, and the initial algebra structure is used to interpret `roll`, `fold` and the corresponding axioms.

The remaining details of the interpretation of linear terms as parse transformers is a relatively straightforward extension of existing semantics for linear logic in monoidal categories [Seely 1989]. In Appendix B, we include the denotations of linear terms and prove that our semantics respects the equational theory of Lambek^D — as well as Axioms 3.1, 3.3 and 3.4.

5.3 Agda Implementation

This denotational semantics in grammars and parse transformers serves as the basis for our prototype implementation of Lambek^D in Cubical Agda [Schaefer et al. 2025]. The implementation is a shallow embedding, meaning that rather than formalizing a syntax of Lambek^D types and terms, we interpret the non-linear universe U as Cubical Agda’s universe of (homotopy) sets (at some universe level ℓ), $\mathbf{hSet} \ell$, and the linear universe L as the function type $\mathbf{String} \rightarrow \mathbf{hSet} \ell$. We use $\mathbf{hSet} \ell$ as it ensures that uniqueness of identity proofs holds for the interpretation of all types in Lambek^D , as it does in any extensional type theory. Then we implement each of the type and term constructors of Lambek^D as combinators on formal grammars or parse transformers, and use Cubical Agda’s equality type to model the term equalities. Cubical Agda is convenient for this purpose as it has built-in support for function extensionality which is convenient for the verification of equality rules. Axioms such as distributivity of $\oplus/\&$ and disjointness of constructors are then provable directly in Agda, and we are careful to only construct grammars, terms and proofs using constructs from Lambek^D .

The main difference between our shallow embedding and Lambek^D is that our linear terms are written in a combinator-style, without being able to use named variables in linear terms. For example, the function `h` from Fig. 4 would be written with the fold combinator applied to subexpressions for the `nil` and `cons` cases, `h = fold nil (cons ∘ id ⊗ cons ∘ assoc-1)`. Notice in the `cons` case of the fold, we manually reassociate `('a' ⊗ 'a') ⊗ 'a'` to `'a' ⊗ ('a' ⊗ 'a')`, then act on the left by the identity and on the right by `cons` to get a parse `'a' ⊗ 'a'`, and finally we act again by `cons` to finally produce a parse of `'a'`. Even in the small example from Fig. 4 we must manage additional complexity — such as manually reassociations — and this only grows more complex in larger programs. On the other hand, a benefit of this shallow embedding is that the parsers are immediately available to a larger Agda development, as they are just normal Agda code. In future work we will look to implement a type checker for a syntax closer to the presentation in this paper, while maintaining the easy integration with an existing proof assistant.

6 Related and Future Work

6.1 Related Work

Grammars as (Linear) Types. Lambek’s original syntactic calculus [Lambek 1958] describes a logical system for linguistic derivations, and it can be given semantics inside of a non-commutative biclosed monoidal category [Lambek 1988]. This led to many uses of non-commutative linear logic and lambda calculi in linguistics [Buszkowski 2003] — including mechanized categorial grammar parsers [Guillaume et al. 2024; Ranta 2011]. This style of grammar formalism has gone by the names Lambek calculus or *categorial grammar*, and it is equal in expressivity to context-free grammars. The existing works on categorial grammar are different in nature to our approach: they are based on non-commutative linear logic, but their terms do not include elimination rules, and so can only express parse trees, not verified parsers.

The most similar prior work to our own is Luo’s Lambek calculus with dependent types [Luo 2018]. They present two systems: one like ours where linear types may only depend on non-linear

ones, and another that allows linear types to depend on other linear types and supports directed Π and Σ types that have no analog in our system. They do not provide a semantics for these connectives, and it is unclear how to interpret their connectives in our grammar semantics. Further, we provide several examples showing that Lambek^D is a practical system for describing grammar formalisms and parsers, and it is unclear if these could be implemented in their calculus.

Frisch and Cardelli introduced the use of a simple type system to reason about regular expressions up to weak equivalence [Frisch and Cardelli 2004]. Henglein and Nielsen build on this type-theoretic view of grammars, pointing out that the values of the types correspond precisely to the parse trees [Henglein and Nielsen 2011]. Lambek^D extends this view of grammars as types to a much broader class of grammars, as well as providing a syntax and equational theory for *parse transformers*, showing that the terms can be viewed not just as parse trees, but as intrinsically verified parsers.

Elliott [2021] uses essentially the same denotational semantics as ours, interpreting regular expressions as type-valued functions on strings. Our denotational semantics of Lambek^D extends this to a much broader class of grammars.

Dependent Linear Types. Our syntax for non-commutative dependent linear type theory is based on the dependent commutative linear type theory of Krishnaswami et al. [2015], itself an extension of Benton’s linear-non-linear calculus [Benton 1994]. A distinct feature of these systems is that linear types and non-linear types are distinct sorts, so the linear logic $!$ operator is not a primitive, but is instead definable, for instance in Lambek^D as $!A = \bigoplus_{-} \uparrow_A I$. Vákár develops a dependent linear type theory where instead the non-linear types are accessed using the $!$ modality, similar to Girard’s original approach [Girard 1987; Vákár 2015]. Additionally, Vákár develops a general categorical semantics for their system, whereas we have only developed a single denotational model. It should be straightforward to adapt Vákár’s general semantics approach to a non-commutative variant that would apply to Lambek^D . This may have applications in finding alternative models, or developing logical relations proofs categorically.

Relation to Separation Logic. Lambek^D is similar in spirit to separation logic [Reynolds 2002]. Semantically, they are closely related: linear types in Lambek^D denote families of sets indexed by a monoid of strings, whereas separation logic formulae typically denote families of predicates indexed by an ordered partial commutative monoid of worlds [Jung et al. 2016]. The monoidal structure in both cases is an instance of the category-theoretic notion of Day convolution monoidal structure [Day 1970]. From a separation-logic perspective, our notion of memory is very primitive: a memory shape is just a string of characters and the state of the memory is never allowed to evolve.

This semantic connection to separation logic suggests an avenue of future work: to develop a program logic based on non-commutative separation logic for verifying imperative implementations of parsers. This could be implemented by modifying an existing separation logic implementation or embedding the logic within Lambek^D .

6.2 Future Work

In this work we have demonstrated feasibility of Lambek^D for the verification of formal grammar theory and sound-by-construction parsers. In future work, we aim to extend Lambek^D to be a practical tool for developing verified parsing components of larger verified software systems.

Verified Parser Generators. We aim to extend our work on parsing to verify other algorithms. For instance, our regular expression parser makes somewhat arbitrary choices when the grammar is ambiguous. In the future we aim to verify Frisch and Cardelli’s *greedy* algorithm for regular expression disambiguation. Defining this algorithm in Lambek^D would provide soundness by construction, but it is not obvious if the greediness property could be verified easily as well.

We also aim to adapt the approach used for our context-free grammar parsers to LL and LR/LALR parsers, with the aim of developing a shared library of intrinsically verified parsing utilities. We would also like to investigate how high-performance verified parsers can be implemented in Lambek^D . This might be done by developing an efficient compiler for Lambek^D directly or by developing a verified compiler to an imperative system within Lambek^D itself.

Semantic Actions. Our verification has mainly focused on the verification that a parser outputs a correct concrete syntax tree for a grammar. However, in practice, parsers are combined with a *semantic action* that emits an *abstract* syntax tree that omits superfluous syntactic details that are unneeded in later stages of the overall program. We can define a semantic action in Lambek^D for a linear type A with semantic outputs in a non-linear type X to be a function $\uparrow (A \multimap \bigoplus_{\cdot} X \top)$. That is, a semantic action is a function that produces a semantic element of X from the concrete parses of A . In future work, we aim to study the question of verifying efficient implementations of parsers with semantic actions, and integrating them into larger verified systems.

Implementation. Our Agda prototype implementation serves as a useful proof of concept for showing what can be implemented in Lambek^D , but it has downsides we aim to address in future work. Firstly, it would be preferable to work with the more intuitive type theoretic syntax we have used in this work, rather than the combinator-style our shallow embedding requires. Additionally, Agda itself does not have a high-performance implementation, and so the parsers we implement in Agda do not have competitive performance to industry parser generators. In future work we aim to study if we can embed a proof of the correctness of a parser generator that produces imperative programs, and if the correctness of those imperative programs can be proven within Lambek^D .

Type Checking and Semantic Analysis. Our focus in this work has been on the verification of parsers for grammars over strings, but because Lambek^D allows for the definition of arbitrarily powerful grammars, the system could also be used in principle for more sophisticated semantic analysis such as scope checking or type checking. Alternatively, we could more directly encode type type systems as linear types in a modified version of Lambek^D where linear types are not grammars over strings, but *type systems* over trees. This could analogously serve as a framework for verified type checking and static analysis.

Acknowledgments

This material is based upon work supported by the Air Force Office of Scientific Research under Grant No. FA9550-23-1-0760, the ERC Consolidator Grant BLAST and the ARIA programme on Safeguarded AI. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Defense, the European Research Council or the Advanced Research and Invention Agency.

Data-Availability Statement

The Agda implementation of Lambek^D and the formalization of our proofs are archived on Zenodo [Schaefer et al. 2025] and the source code is available on GitHub.

References

- Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (Jan. 2015). doi:10.1017/S095679681500009X
- P. N. Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (CSL 1994). doi:10.1007/BFb0022251
- W. Buszkowski. 2003. *Type Logics in Grammar*. Springer Netherlands, Dordrecht, 337–382. doi:10.1007/978-94-017-3598-8_12
- Noam Chomsky. 1963. Formal Properties of Grammars. *Handbook of Mathematical Psychology II* (1963), 323–418. Retrieved April 9, 2025 from <https://archive.org/details/handbookofmathem017893mbp/page/322/mode/2up>
- J. R. B. Cockett. 1993. Introduction to distributive categories. *Mathematical Structures in Computer Science* 3, 3 (1993), 277–307. doi:10.1017/S0960129500000232
- Thierry Coquand. 2013. Presheaf model of type theory. (2013). Retrieved April 8, 2025 from <https://www.cse.chalmers.se/~coquand/presheaf.pdf>
- Nils Anders Danielsson. 2010. Total Parser Combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) (ICFP '10). 285–296. doi:10.1145/1863543.1863585
- Brian John Day. 1970. *Construction of biclosed categories*. Ph.D. Dissertation. University of New South Wales PhD thesis. Retrieved April 8, 2025 from <https://web.science.mq.edu.au/~street/DayPhD.pdf>
- Romain Edelmann, Jad Hamza, and Viktor Kunčák. 2020. Zippy LL(1) parsing with derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Online) (PLDI 2020). doi:10.1145/3385412.3385992
- Conal Elliott. 2021. Symbolic and automatic differentiation of languages. *Proceedings of the ACM on Programming Languages* (PACMPL) 5, ICFP, Article 78 (Aug. 2021). doi:10.1145/3473583
- Alain Frisch and Luca Cardelli. 2004. Greedy Regular Expression Matching. In *Automata, Languages, and Programming* (Turku, Finland) (ICALP 2004). doi:10.1007/978-3-540-27836-8_53
- Nicola Gambino and Martin Hyland. 2003. Wellfounded Trees and Dependent Polynomial Functors. In *Types for Proofs and Programs* (Torino, Italy) (TYPES 2003). 210–225. doi:10.1007/978-3-540-24849-1_14
- Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. doi:10.1016/0304-3975(87)90045-4
- Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2021. Multimodal Dependent Type Theory. *Logical Methods in Computer Science* 17, 3 (July 2021). doi:10.46298/lmcs-17(3:11)2021
- Maxime Guillaume, Sylvain Pogodalla, and Vincent Tourneur. 2024. ACGtk: A Toolkit for Developing and Running Abstract Categorical Grammars. In *Functional and Logic Programming* (Kumamoto, Japan) (17th International Symposium, FLOPS 2024). 13–30. doi:10.1007/978-981-97-2300-3_2
- Fritz Henglein and Lasse Nielsen. 2011. Regular expression containment: coinductive axiomatization and computational interpretation. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) (POPL '11). 385–398. doi:10.1145/1926385.1926429
- Martin Hofmann. 1997. *Syntax and Semantics of Dependent Types*. Cambridge University Press, 79–130.
- Jacques-Henri Jourdan, Francois Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Programming Languages and Systems, 21st European Symposium on Programming* (Tallinn, Estonia) (ESOP 2012). 397–416. doi:10.1007/978-3-642-28869-2_20
- Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) (ICFP 2016). 256–269. doi:10.1145/2951913.2951943
- Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) (POPL '15). 17–30. doi:10.1145/2676726.2676969

- Joachim Lambek. 1958. The Mathematics of Sentence Structure. *The American Mathematical Monthly* 65, 3 (1958), 154–170. doi:10.1080/00029890.1958.11989160
- J. Lambek. 1988. *Categorical and Categorical Grammars*. 297–317. doi:10.1007/978-94-015-6878-4_11
- Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2021. CoStar: A Verified ALL(*) Parser. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (Online) (PLDI 2021)*. 420–434. doi:10.1145/3453483.3454053
- Haas Leiß. 1992. Towards Kleene Algebra with recursion (CSL 1991). doi:10.1007/BFb0023771
- Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:10.1145/1538788.1538814
- Zhaohui Luo. 2018. Substructural Calculi with Dependent Types. In *Linearity & TLLA Joint Workshop* (Oxford, UK). doi:10.29007/qrqp
- Georgi Nakov and Fredrik Nordvall Forsberg. 2022. Quantitative Polynomial Functors. In *27th International Conference on Types for Proofs and Programs (TYPES 2021)* (Leiden, The Netherlands) (*Leibniz International Proceedings in Informatics (LIPIcs)*, Vol. 239). 10:1–10:22. doi:10.4230/LIPIcs.TYPES.2021.10
- M. O. Rabin and D. Scott. 1959. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development* 3, 2 (April 1959), 114–125. doi:10.1147/rd.32.0114
- Aarne Ranta. 2011. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).
- J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science (Copenhagen, Denmark) (LICS 2002)*. 55–74. doi:10.1109/LICS.2002.1029817
- Steven Schaefer, Nathan Varner, Pedro Henrique Azevedo de Amorim, and Max S. New. 2025. *Agda Formalization of "Intrinsic Verification of Parsers and Formal Grammar Theory in Dependent Lambek Calculus"*. doi:10.5281/zenodo.15243560
- R. A. G. Seely. 1989. Linear logic, *-autonomous categories and cofree coalgebras. In *Categories in computer science and logic (Boulder, CO, 1987)*. Contemp. Math., Vol. 92. Amer. Math. Soc., Providence, RI, 371–382. doi:10.1090/conm/092/1003210
- The Agda Community. 2024. *Cubical Agda Library*. Retrieved April 8, 2025 from <https://github.com/agda/cubical>
- Ken Thompson. 1968. Programming Techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. doi:10.1145/363347.363387
- The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study. Retrieved April 8, 2025 from <https://homotopytypetheory.org/book>
- Matthijs Vákár. 2015. A Categorical Semantics for Linear Logical Frameworks. In *Foundations of Software Science and Computation Structures (London, UK) (FoSSaCS 2015)*. 102–116. doi:10.1007/978-3-662-46678-0_7
- Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proceedings of the ACM on Programming Languages (PACMPL)* 3, ICFP (July 2019). doi:10.1145/3341691
- Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (San Jose, California, USA) (PLDI '11)*. 283–294. doi:10.1145/1993498.1993532

$$\begin{aligned}
 \text{el}(\text{Var } M) B &= B M \\
 \text{el}(K A) B &= A \\
 \text{el}\left(\bigoplus A\right) B &= \bigoplus_{y:Y} \text{el}(A y) B \\
 \text{el}\left(\bigotimes A\right) B &= \bigotimes_{y:Y} \text{el}(A y) B \\
 \text{el}(A \otimes A') B &= \text{el}(A) B \otimes \text{el}(A') B \\
 \text{map}(\text{Var } M) f &= f M \\
 \text{map}(K A) f &= \lambda a. a \\
 \text{map}\left(\bigoplus A\right) f &= \lambda a. \text{let } \sigma y a_y = a \text{ in } \sigma y \text{map}(A y) f a_y \\
 \text{map}\left(\bigotimes A\right) f &= \lambda a. \lambda^{\&} y. \text{map}(A y) f (\pi y a) \\
 \text{map}(A \otimes A') f &= \lambda b. \text{let } (a, a') = b \text{ in } (\text{map}(A) f a, \text{map}(A') f a')
 \end{aligned}$$

Fig. 17. Strictly positive functors functorial actions

A Syntax

In this section we include the elided syntactic forms, as well as definitions and basic properties of linear and non-linear substitution.

- In Fig. 17, we provide the functorial actions of the el and map operations used to define the indexed inductive types in Fig. 10.
- In Figs. 18 and 19 we give the inference rules for the smallness judgments on nonlinear types and linear types, respectively.
- In Fig. 20, we include the full set of rules for non-linear types, and in Fig. 21 we provide their judgmental equalities.
- In Fig. 22, we give the judgmental equalities between linear terms in Lambek^D .

A.1 Non-linear Types

We define sum types, list types, and $\text{Fin } n$ from primitive non-linear types.

For $X, Y : U$, define the sum type $X + Y$ as,

$$X + Y = \sum_{b:\text{Bool}} \text{elim}_{\text{Bool}}(U, X, Y)(b)$$

For $n : \text{Nat}$, define $\text{Fin } n$ as,

$$\begin{aligned}
 \text{Fin } 0 &= \perp \\
 \text{Fin}(\text{suc } n) &= 1 + (\text{Fin } n)
 \end{aligned}$$

For $X : U$, define $\text{List } X$ as,

$$\text{List } X = \sum_{n:\text{Nat}} \prod_{k:\text{Fin } n} X$$

$$\begin{array}{c}
\frac{}{\Gamma \vdash 1 \text{ small}} \quad \frac{}{\Gamma \vdash \text{Bool small}} \quad \frac{}{\Gamma \vdash \perp \text{ small}} \quad \frac{}{\Gamma \vdash \text{Nat small}} \quad \frac{\Gamma \vdash X \text{ small} \quad \Gamma, x : X \vdash Y \text{ small}}{\Gamma \vdash \prod_{x:X} Y \text{ small}} \\
\frac{\Gamma \vdash X \text{ small} \quad \Gamma, x : X \vdash Y \text{ small}}{\Gamma \vdash \sum_{x:X} Y \text{ small}} \quad \frac{\Gamma \vdash M : U}{\Gamma \vdash [M] \text{ small}} \quad \frac{\Gamma \vdash A \text{ small lin.}}{\Gamma \vdash \uparrow A \text{ small}} \quad \frac{\Gamma \vdash X \text{ small} \quad \Gamma \vdash M : X \quad \Gamma \vdash N : X}{\Gamma \vdash M \approx_X N \text{ small}}
\end{array}$$

Fig. 18. Small Non-linear Types

$$\begin{array}{c}
\frac{}{\Gamma \vdash I \text{ small lin.}} \quad \frac{c \in \Sigma}{\Gamma \vdash 'c' \text{ small lin.}} \quad \frac{\Gamma \vdash A \text{ small lin.} \quad \Gamma \vdash B \text{ small lin.}}{\Gamma \vdash A \otimes B \text{ small lin.}} \quad \frac{\Gamma \vdash A \text{ small lin.} \quad \Gamma \vdash B \text{ small lin.}}{\Gamma \vdash A \multimap B \text{ small lin.}} \\
\frac{\Gamma \vdash A \text{ small lin.} \quad \Gamma \vdash B \text{ small lin.}}{\Gamma \vdash A \multimap B \text{ small lin.}} \quad \frac{\Gamma \vdash X \text{ small} \quad \Gamma, x : X \vdash A \text{ small lin.}}{\Gamma \vdash \bigoplus_{x:X} A \text{ small lin.}} \quad \frac{\Gamma \vdash X \text{ small} \quad \Gamma, x : X \vdash A \text{ small lin.}}{\Gamma \vdash \bigotimes_{x:X} A \text{ small lin.}} \\
\frac{\Gamma \vdash A \text{ small lin.} \quad \Gamma \vdash B \text{ small lin.} \quad \Gamma \vdash f : \uparrow (A \multimap B) \quad \Gamma \vdash g : \uparrow (A \multimap B)}{\Gamma \vdash \{a \mid f a = g a\} \text{ small lin.}} \quad \frac{\Gamma \vdash M : L}{\Gamma \vdash [M] \text{ small lin.}}
\end{array}$$

Fig. 19. Small Linear Types

$$\begin{array}{c}
\boxed{\frac{\Gamma \vdash X \text{ type}}{\Gamma \vdash M : X}} \quad \frac{}{\Gamma, x : X, \Gamma' \vdash x : X} \quad \frac{\Gamma \vdash M : Y \quad \Gamma \vdash X \equiv Y \text{ type}}{\Gamma \vdash M : X} \quad \frac{}{\Gamma \vdash \text{tt} : 1} \quad \frac{\Gamma \vdash M : \perp \quad \Gamma \vdash X \text{ type}}{\Gamma \vdash \text{elim}_{\perp}(X, M) : X} \\
\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \quad \frac{}{\Gamma \vdash \text{false} : \text{Bool}} \quad \frac{\Gamma, b : \text{Bool} \vdash X(b) \text{ type} \quad \Gamma \vdash M_0 : X(\text{false}) \quad \Gamma \vdash M_1 : X(\text{true})}{\Gamma, b : \text{Bool} \vdash \text{elim}_{\text{Bool}}(X, M_0, M_1)(b) : X(b)} \\
\frac{}{\Gamma \vdash \emptyset : \text{Nat}} \quad \frac{\Gamma \vdash n : \text{Nat}}{\Gamma \vdash \text{succ } n : \text{Nat}} \\
\frac{\Gamma, n : \text{Nat} \vdash X(n) \text{ type} \quad \Gamma \vdash M_0 : X(\emptyset) \quad \Gamma, n : \text{Nat}, x : X(n) \vdash M_{\text{succ}}(n, x) : X(\text{succ } n)}{\Gamma, n : \text{Nat} \vdash \text{elim}_{\text{Nat}}(X, M_0, M_{\text{succ}})(n) : X(n)} \\
\frac{\Gamma \vdash M : X \quad \Gamma \vdash N : Y\{M/x\}}{\Gamma \vdash (M, N) : \sum_{x:X} Y} \quad \frac{\Gamma \vdash M : \sum_{x:X} Y}{\Gamma \vdash M.\text{fst} : X} \quad \frac{\Gamma \vdash M : \sum_{x:X} Y}{\Gamma \vdash M.\text{snd} : Y\{M.\text{fst}/x\}} \quad \frac{\Gamma \vdash \prod_{x:X} Y \text{ type} \quad \Gamma, x : X \vdash M : Y}{\Gamma \vdash \lambda x. M : \prod_{x:X} Y} \\
\frac{\Gamma \vdash M : \prod_{x:X} Y \quad \Gamma \vdash N : X}{\Gamma \vdash MN : Y\{N/x\}} \quad \frac{\Gamma \vdash X \text{ type}}{\Gamma \vdash [X] : U} \quad \frac{\Gamma \vdash A \text{ lin. type}}{\Gamma \vdash [A] : L} \quad \frac{\Gamma; \cdot \vdash e : A}{\Gamma \vdash e : \uparrow A}
\end{array}$$

Fig. 20. Non-linear Typing

$$\begin{array}{c}
 \boxed{\frac{\Gamma \vdash M : X \quad \Gamma \vdash N : X}{\Gamma \vdash M \equiv N : X}} \qquad \frac{\Gamma \vdash M : 1 \quad \Gamma \vdash N : 1}{\Gamma \vdash M \equiv N : 1} \qquad \frac{\Gamma, x : X \vdash M : Y \quad \Gamma \vdash N : X}{\Gamma \vdash (\lambda x. M) N \equiv M\{N/x\} : X} \\
 \\
 \frac{\Gamma \vdash M : \prod_{x:X} Y}{\Gamma \vdash M \equiv \lambda x. M x : \prod_{x:X} Y} \qquad \frac{\Gamma \vdash M : X \quad \Gamma, x : X \vdash N : Y\{M/x\}}{\Gamma \vdash (M, N).fst \equiv M : X} \qquad \frac{\Gamma \vdash M : X \quad \Gamma, x : X \vdash N : Y\{M/x\}}{\Gamma \vdash (M, N).snd \equiv N : Y\{M/x\}} \\
 \\
 \frac{\Gamma \vdash M : \sum_{x:X} Y}{\Gamma \vdash (M.fst, M.snd) \equiv M : \sum_{x:X} Y} \qquad \frac{\Gamma \vdash M_0 : X(\text{false}) \quad \Gamma \vdash M_1 : X(\text{true})}{\Gamma \vdash \text{elim}_{\text{Bool}}(X, M_0, M_1)(\text{false}) \equiv M_0 : X(b)} \\
 \\
 \frac{\Gamma \vdash M_0 : X(\text{false}) \quad \Gamma \vdash M_1 : X(\text{true})}{\Gamma \vdash \text{elim}_{\text{Bool}}(X, M_0, M_1)(\text{true}) \equiv M_1 : X(b)} \qquad \frac{\Gamma \vdash M_0 : X(\emptyset) \quad \Gamma, n : \text{Nat}, x : X(n) \vdash M_{\text{suc}}(n, x) : X(\text{suc } n)}{\Gamma \vdash \text{elim}_{\text{Nat}}(X, M_0, M_{\text{suc}})(\emptyset) \equiv M_0 : X(\emptyset)} \\
 \\
 \frac{\Gamma \vdash M_0 : X(\emptyset) \quad \Gamma, n : \text{Nat}, x : X(n) \vdash M_{\text{suc}}(n, x) : X(\text{suc } n)}{\Gamma, n : \text{Nat} \vdash \text{elim}_{\text{Nat}}(X, M_0, M_{\text{suc}})(\text{suc } n) \equiv M_{\text{suc}}(n, \text{elim}_{\text{Nat}}(X, M_0, M_{\text{suc}})(n)) : X(\text{suc } n)} \qquad \frac{\Gamma \vdash M : U}{\Gamma \vdash \llbracket M \rrbracket \equiv M : U} \\
 \\
 \frac{\Gamma \vdash M : L}{\Gamma \vdash \llbracket M \rrbracket \equiv M : L}
 \end{array}$$

Fig. 21. Judgmental equality for non-linear terms

A.2 Substitutions

Definition A.1. The set of (non-linear) substitutions $\gamma \in \text{Subst}(\Gamma, \Gamma')$ where Γ ctx and Γ' ctx is defined by recursion on Γ :

$$\begin{aligned}
 \text{Subst}(\Gamma, \cdot) &= \{\cdot\} \\
 \text{Subst}(\Gamma, \Gamma', x : A) &= \{(\gamma, M/x) \mid \gamma \in \text{Subst}(\Gamma, \Gamma') \wedge \Gamma \vdash M : A[\gamma]\}
 \end{aligned}$$

simultaneously with an action of substitution on types, terms, etc. in the standard way.

It is straightforward, but laborious to establish that all forms in the type theory that are parameterized by a non-linear context Γ support the admissible actions of a substitution $\gamma \in \text{Subst}(\Gamma', \Gamma)$ given in Figs. 23 and 24.

Definition A.2. Let $\Gamma \vdash \Delta$ lin. ctx. and $\Gamma \vdash \Delta'$ lin. ctx.. The set of linear substitutions $\text{Subst}(\Delta', \Delta)$ is defined by recursion on Δ :

$$\begin{aligned}
 \text{Subst}(\Delta', \cdot) &= \{\cdot \mid \Delta' = \cdot\} \\
 \text{Subst}(\Delta', (\Delta, a : A)) &= \{(\delta, e/a) \mid \delta \in \text{Subst}(\Delta_1, \Delta), \Delta_2 \vdash e : A, \Delta' = (\Delta_1, \Delta_2)\}
 \end{aligned}$$

Given substitutions $\delta_1 \in \text{Subst}(\Delta'_1, \Delta_1)$ and $\delta_2 \in \text{Subst}(\Delta'_2, \Delta_2)$, we can define a substitution $\delta_1, \delta_2 \in \text{Subst}((\Delta'_1, \Delta'_2), (\Delta_1, \Delta_2))$. Furthermore, for any substitution $\delta \in \text{Subst}(\Delta, (\Delta_1, \Delta_2))$, we can deconstruct $\delta = \delta_1, \delta_2$ with $\delta_1 \in \text{Subst}(\Delta'_1, \Delta_1)$ and $\delta_2 \in \text{Subst}(\Delta'_2, \Delta_2)$.

Definition A.3. Given any $\Gamma; \Delta \vdash e : A$ and $\delta \in \text{Subst}(\Delta', \Delta)$, we define the action of the substitution on e in Fig. 24, frequently using the inversion principle to split the substitution into constituent components. By induction on linear term and equality judgments, we establish the following admissible rules for $\delta \in \text{Subst}(\Delta', \Delta)$:

$$\begin{array}{c}
 \frac{\Gamma; \Delta \vdash e : A}{\Gamma; \Delta' \vdash e[\delta] : A} \qquad \frac{\Gamma; \Delta \vdash e \equiv f : A}{\Gamma; \Delta' \vdash e[\delta] \equiv f[\delta'] : A}
 \end{array}$$

$$\begin{array}{c}
\boxed{\frac{\Gamma; \Delta \vdash e : A \quad \Gamma; \Delta \vdash e' : A}{\Gamma; \Delta \vdash e \equiv e' : A}} \quad \frac{\Gamma; \Delta, a : A \vdash e : C \quad \Gamma; \Delta' \vdash e' : A}{\Gamma; \Delta, \Delta' \vdash (\lambda^{\circ} a. e) e' \equiv e\{e'/a\} : C} \quad \frac{\Gamma; \Delta \vdash e : A \multimap B}{\Gamma; \Delta \vdash e \equiv \lambda^{\circ} a. e a : A \multimap B} \\
\\
\frac{\Gamma; a : A, \Delta \vdash e : C \quad \Gamma; \Delta' \vdash e' : A}{\Gamma; \Delta, \Delta' \vdash (\lambda^{\circ} a. e) e' \equiv e\{e'/a\} : C} \quad \frac{\Gamma; \Delta \vdash e : B \multimap A}{\Gamma; \Delta \vdash e \equiv \lambda^{\circ} a. e \multimap a : B \multimap A} \quad \frac{\Gamma, x : X \vdash e : A \quad \Gamma \vdash M : X}{\Gamma; \Delta \vdash (\lambda^{\otimes} x. e) M \equiv e\{M/x\} : C} \\
\\
\frac{\Gamma; \Delta \vdash e : \bigotimes_{x:X} A}{\Gamma; \Delta \vdash e \equiv \lambda^{\otimes} x. e x : \bigotimes_{x:X} A} \quad \frac{\Gamma; \Delta_1, \Delta_2 \vdash e : C}{\Gamma; \Delta_1, \Delta_2 \vdash \text{let } () = () \text{ in } e \equiv e : C} \\
\\
\frac{\Gamma; \Delta_2 \vdash e : I \quad \Gamma; \Delta_1, a : A, \Delta_3 \vdash e' : C}{\Gamma; \Delta_1, \Delta_3 \vdash \text{let } () = e \text{ in } e'\{()/a\} \equiv e'\{e/a\} : C} \\
\\
\frac{\Gamma; \Delta_2 \vdash e : A \quad \Gamma; \Delta_3 \vdash e' : B \quad \Gamma; \Delta_1, a : A, b : B, \Delta_4 \vdash e'' : C}{\Gamma; \Delta_1, \Delta_2, \Delta_3, \Delta_4 \vdash \text{let } (a, b) = (e, e') \text{ in } e'' \equiv e''\{e/a, e'/b\} : C} \\
\\
\frac{\Gamma; \Delta_2 \vdash e : A \otimes B \quad \Gamma; \Delta_1, c : A \otimes B, \Delta_3 \vdash e' : C}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{let } (a, b) = e \text{ in } e'\{(a, b)/c\} \equiv e'\{e/c\} : C} \\
\\
\frac{\Gamma \vdash M : X \quad \Gamma; \Delta_2 \vdash e : A \quad \Gamma, x : X \vdash \Delta_1, a : A, \Delta_3}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{let } \sigma x a = \sigma M \text{ in } e' \equiv e'\{M/x, e/a\} : C} \\
\\
\frac{\Gamma; \Delta_1, y : \bigoplus_{x:X} A, \Delta_2 \vdash e' : C \quad \Gamma; \Delta_2 \vdash e : \bigoplus_{x:X} A}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{let } \sigma x a = e \text{ in } e'\{\sigma x a/y\} \equiv e'\{e/y\} : C} \quad \frac{\Gamma; \Delta \vdash e : A \quad \Gamma; \Delta \vdash f e \equiv g e}{\Gamma; \Delta \vdash \langle e \rangle. \pi \equiv e : A} \\
\\
\frac{\Gamma; \Delta \vdash e : \{e \mid f e = g e\}}{\Gamma; \Delta \vdash \langle e \rangle. \pi \equiv e : \{e \mid f e = g e\}}
\end{array}$$

Fig. 22. Judgmental equality for linear terms

$$\begin{array}{c}
\frac{\Gamma \vdash X \text{ type}}{\Gamma' \vdash X[\gamma] \text{ type}} \quad \frac{\Gamma \vdash X \text{ small}}{\Gamma' \vdash X[\gamma] \text{ small}} \quad \frac{\Gamma \vdash X \equiv Y \text{ type}}{\Gamma' \vdash X[\gamma] \equiv Y[\gamma] \text{ type}} \quad \frac{\Gamma \vdash M : X}{\Gamma' \vdash M[\gamma] : X[\gamma]} \\
\\
\frac{\Gamma \vdash M \equiv N : X}{\Gamma' \vdash M[\gamma] \equiv N[\gamma] : X[\gamma]} \quad \frac{\Gamma \vdash \Delta \text{ lin. ctx.}}{\Gamma' \vdash \Delta[\gamma] \text{ lin. ctx.}} \quad \frac{\Gamma \vdash A \text{ lin. type}}{\Gamma' \vdash A[\gamma] \text{ lin. type}} \quad \frac{\Gamma \vdash A \equiv B \text{ lin. type}}{\Gamma' \vdash A[\gamma] \equiv B[\gamma] \text{ lin. type}} \\
\\
\frac{\Gamma; \Delta \vdash e : A}{\Gamma'; \Delta[\gamma] \vdash e[\gamma] : A[\gamma]} \quad \frac{\Gamma; \Delta \vdash e \equiv f : A}{\Gamma'; \Delta[\gamma] \vdash e[\gamma] \equiv f[\gamma] : A[\gamma]}
\end{array}$$

Fig. 23. Non-linear substitution

B Denotational Semantics

Here we extend the denotational semantics from Section 5 to cover all of Dependent Lambek Calculus syntax. Here we will freely use that the category of grammars is a complete, co-complete biclosed monoidal category, and use categorical notation for the constructions in the denotational semantics. For example, we will use the same notation I , \otimes , \multimap , \multimap for the biclosed monoidal structure of \mathbf{Gr} that we do for the corresponding syntactic notions.

$$\begin{aligned}
a[e/a] &= e \\
(e_1, e_2)[\delta_1, \delta_2] &= (e_1[\delta_1], e_2[\delta_2]) \\
(\text{let } (a, b) = e \text{ in } e')[\delta_1, \delta_2, \delta_3] &= \text{let } (a, b) = e[\delta_2] \text{ in } e'[\delta_1, a/a, b/b, \delta_2] \\
()[\cdot] &= () \\
\text{let } () = e \text{ in } e'[\delta_1, \delta_2, \delta_3] &= \text{let } () = e[\delta_2] \text{ in } e'[\delta_1, a/a, b/b, \delta_2] \\
(\lambda^\circ a. e)[\delta] &= \lambda^\circ a. e[\delta, a/a] \\
(e' e)[\delta_1, \delta_2] &= e'[\delta_1] e[\delta_2] \\
(\lambda^\circ a. e)[\delta] &= \lambda^\circ a. e[\delta, a/a] \\
(e' e)[\delta_1, \delta_2] &= e'[\delta_1] e[\delta_2] \\
(\lambda^\circ a. e)[\delta] &= \lambda^\circ a. e[a/a, \delta] \\
(e' \circ^\circ e)[\delta_1, \delta_2] &= e'[\delta_1] \circ^\circ e[\delta_2] \\
(\lambda^{\&x}. e)[\delta] &= \lambda^{\&x}. e[\delta] \\
(e . \pi M)[\delta] &= (e[\delta] . \pi M) \\
(\sigma M e)[\delta] &= \sigma M e[\delta] \\
(\text{let } \sigma x a = e \text{ in } e')[\delta_1, \delta_2, \delta_3] &= \text{let } \sigma x a = e[\delta_2] \text{ in } e'[\delta_1, \delta_3] \\
\langle e \rangle[\delta] &= \langle e[\delta] \rangle \\
(e . \pi)[\delta] &= e[\delta] . \pi
\end{aligned}$$

Fig. 24. Action of substitution on linear terms

Definition B.1 (Denotation of Linear Contexts). The semantics of linear contexts $\Gamma \vdash \Delta \text{ lin. ctx.}$ is defined as follows:

$$\begin{aligned}
\llbracket \cdot \rrbracket \gamma &= \mathbf{I} \\
\llbracket \Delta, x : A \rrbracket \gamma &= \llbracket \Delta \rrbracket \gamma \otimes \llbracket A \rrbracket \gamma
\end{aligned}$$

Definition B.2 (Denotation of Linear Substitutions). The semantics of a linear substitution $\delta : \text{Subst}(\Delta', \Delta)$ are given as maps $\llbracket \delta \rrbracket \gamma : \llbracket \Delta' \rrbracket \gamma \rightarrow \Delta$. Define $\llbracket \delta \rrbracket \gamma$ by recursion on δ :

$$\begin{aligned}
\llbracket \cdot \rrbracket \gamma &= \text{id}_{\mathbf{I}} \\
\llbracket \delta, e/a \rrbracket \gamma &= \llbracket \delta \rrbracket \gamma \otimes \llbracket e \rrbracket \gamma \circ m_{\Delta_1, \Delta_2}
\end{aligned}$$

where $\Delta_2 \vdash e : A$ and $\Delta' = \Delta_1, \Delta_2$.

THEOREM B.3. For any $\Gamma \vdash \Delta_1, \Delta_2 \text{ lin. ctx.}$ and $\gamma \in \llbracket \Gamma \rrbracket$ there is a natural isomorphism $m_{\Delta_1, \Delta_2} : \llbracket \Delta_1, \Delta_2 \rrbracket \gamma \cong \llbracket \Delta_1 \rrbracket \gamma \otimes \llbracket \Delta_2 \rrbracket \gamma$.

This can be extended to a sequence of contexts of any length.

PROOF. Construct m_{Δ_1, Δ_2} by recursion on Δ_2 .

$$\begin{aligned}
m_{\Delta_1, \cdot} &= \rho^{-1} \\
m_{\Delta_1, (\Delta_2, a:A)} &= \alpha \circ m_{\Delta_1, \Delta_2} \otimes \text{id}
\end{aligned}$$

□

LEMMA B.4. For each term $\Delta \vdash e : A$ and substitution $\delta : \text{Subst}(\Delta', \Delta)$, the semantics of δ acting on e splits into the composition $\llbracket e[\delta] \rrbracket \gamma = \llbracket e \rrbracket \gamma \circ \llbracket \delta \rrbracket \gamma$.

B.1 Grammar Semantics for Linear Terms

Here we define denotations of linear terms. Note that the denotations interpret typing derivations, not raw terms, as the data of how contexts are split is needed in order to construct the correct associator functions. Further, we demonstrate that the denotational semantics respects the equational theory of Lambek^D. The correctness of the equational theory heavily relies on the *coherence theorem* for monoidal categories. The coherence theorem says that any diagram in a monoidal category constructed using only associators $\alpha_{A,B,C} : (A \otimes B) \otimes C \cong A \otimes (B \otimes C)$, unitors $\rho_A : A \otimes I \cong A$ and $\lambda_A : I \otimes A \cong A$ and compositions and tensor products of these, commutes. We call a morphism built in this way a *generalized associator*.

B.1.1 Variables. Note that the denotation of a singleton context $a : A$ is given as

$$\llbracket a : A \rrbracket \gamma = \llbracket \cdot, a : A \rrbracket \gamma = I \otimes \llbracket A \rrbracket \gamma$$

So for $a : A \vdash a : A$, the denotation of a single variable term $\llbracket a \rrbracket \gamma : \llbracket a : A \rrbracket \gamma \rightarrow \llbracket A \rrbracket \gamma$ is given by the left unitor

$$\llbracket a \rrbracket \gamma = \lambda$$

B.1.2 Linear Unit.

I-Introduction. $\llbracket () \rrbracket \gamma : \llbracket \cdot \rrbracket \gamma \rightarrow \llbracket I \rrbracket \gamma$.

$$\llbracket () \rrbracket \gamma = \text{id}_I$$

I-Elimination. $\llbracket \text{let } () = e \text{ in } e' \rrbracket \gamma : \llbracket \Delta'_1, \Delta, \Delta'_2 \rrbracket \gamma \rightarrow \llbracket C \rrbracket \gamma$ defined in the following diagram,

$$\begin{array}{ccc} \llbracket \Delta'_1, \Delta, \Delta'_2 \rrbracket \gamma & \xrightarrow{m_{(\Delta'_1, \Delta), \Delta'_2}} & \llbracket \Delta'_1, \Delta \rrbracket \gamma \otimes \llbracket \Delta'_2 \rrbracket \gamma \xrightarrow{m_{\Delta'_1, \Delta} \otimes \text{id}} (\llbracket \Delta'_1 \rrbracket \gamma \otimes \llbracket \Delta \rrbracket \gamma) \otimes \llbracket \Delta'_2 \rrbracket \gamma \\ & & \downarrow (\text{id} \otimes \llbracket e \rrbracket \gamma) \otimes \text{id} \\ \llbracket \Delta'_1, \Delta'_2 \rrbracket \gamma & \xleftarrow{m_{\Delta'_1, \Delta'_2}^{-1}} & \llbracket \Delta'_1 \rrbracket \gamma \otimes \llbracket \Delta'_2 \rrbracket \gamma \xleftarrow{\rho \otimes \text{id}} (\llbracket \Delta'_1 \rrbracket \gamma \otimes I) \otimes \llbracket \Delta'_2 \rrbracket \gamma \\ & & \downarrow \llbracket e' \rrbracket \gamma \\ & & \llbracket C \rrbracket \gamma \end{array}$$

We demonstrate that the denotations of the introduction and elimination forms for I obey the β and η equalities for I.

I β . Given $\Delta'_1, \cdot, \Delta'_2 \vdash e' : C$, the desired β law is

$$\llbracket \text{let } () = () \text{ in } e' \rrbracket \gamma = \llbracket e \rrbracket \gamma$$

PROOF.

$$\begin{aligned} \llbracket \text{let } () = () \text{ in } e' \rrbracket \gamma &= \llbracket e' \rrbracket \gamma \circ m_{\Delta'_1, \Delta'_2}^{-1} \circ \rho \otimes \text{id} \circ (\text{id} \otimes \llbracket () \rrbracket \gamma) \otimes \text{id} \circ m_{\Delta_1, \cdot} \otimes \text{id} \circ m_{(\Delta'_1, \cdot), \Delta'_2} \\ &= \llbracket e' \rrbracket \gamma \circ m_{\Delta'_1, \Delta'_2}^{-1} \circ \rho \otimes \text{id} \circ (\text{id} \otimes \text{id}) \otimes \text{id} \circ m_{\Delta_1, \cdot} \otimes \text{id} \circ m_{\Delta'_1, \Delta'_2} \\ &= \llbracket e' \rrbracket \gamma \circ m_{\Delta'_1, \Delta'_2}^{-1} \circ \rho \otimes \text{id} \circ \rho^{-1} \otimes \text{id} \circ m_{\Delta'_1, \Delta'_2} \\ &= \llbracket e' \rrbracket \gamma \end{aligned} \quad (\text{coherence})$$

□

$I\eta$. Similarly, for $\Delta_1, a : A, \Delta_3 \vdash e' : C$ and $\Delta_2 \vdash e : I$ the desired η law is

$$\llbracket \text{let } () = e \text{ in } e' [()] / a \rrbracket \gamma = \llbracket e' [e/a] \rrbracket \gamma$$

However, through application of Lemma B.4 it suffices to handle the case where e is a variable a' . That is,

$$\begin{aligned} \text{let } () = e \text{ in } e' [()] / a &= (\text{let } () = a' \text{ in } e' [()] / a) [e/a'] \\ e' [e/a] &= e' [a'/a] [e/a] \end{aligned}$$

so without loss of generality we may take e to be variable a' . We will additionally use this style of argumentation when necessary throughout this section.

PROOF.

$$\begin{aligned} \llbracket \text{let } () = a' \text{ in } e' [()] / a \rrbracket \gamma &= \llbracket e' \rrbracket \gamma \circ m_{\Delta'_1, \Delta'_2}^{-1} \circ \rho \otimes \text{id} \circ (\text{id} \otimes \llbracket a' \rrbracket \gamma) \otimes \text{id} \circ m_{\Delta_1, \cdot} \otimes \text{id} \circ m_{\Delta'_1, \Delta'_2} \\ &= \llbracket e' \rrbracket \gamma \circ m_{\Delta'_1, \Delta'_2}^{-1} \circ \rho \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m_{\Delta_1, \cdot} \otimes \text{id} \circ m_{\Delta'_1, \Delta'_2} \\ &= \llbracket e' \rrbracket \gamma \quad (\text{coherence}) \end{aligned}$$

Because $m_{\Delta'_1, \Delta'_2}^{-1} \circ \rho \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m_{\Delta_1, \cdot} \otimes \text{id} \circ m_{\Delta'_1, \Delta'_2}$ is a composition of generalized associators from $\llbracket \Delta'_1, \Delta'_2 \rrbracket \gamma$ to itself, it is equal to the identity by the coherence theorem for monoidal categories.

Further by Lemma B.4,

$$\begin{aligned} \llbracket e' [a'/a] \rrbracket \gamma &= \llbracket e' \rrbracket \gamma \circ \llbracket a' / a \rrbracket \gamma \quad (\text{Lemma B.4}) \\ &= \llbracket e' \rrbracket \gamma \quad (\text{coherence}) \end{aligned}$$

Again by the coherence theorem, $\llbracket a' / a \rrbracket \gamma = \text{id}$. Thus, η law for I holds in the denotational semantics. \square

B.1.3 Tensor.

\otimes -Introduction. $\llbracket (e_1, e_2) \rrbracket \gamma : \llbracket \Delta, \Delta' \rrbracket \gamma \rightarrow \llbracket A \otimes B \rrbracket \gamma$ is given by the diagram

$$\llbracket \Delta, \Delta' \rrbracket \gamma \xrightarrow{m_{\Delta, \Delta'}} \llbracket \Delta \rrbracket \gamma \otimes \llbracket \Delta' \rrbracket \gamma \xrightarrow{\llbracket e_1 \rrbracket \gamma \otimes \llbracket e_2 \rrbracket \gamma} \llbracket A \rrbracket \gamma \otimes \llbracket B \rrbracket \gamma$$

\otimes -Elimination. $\llbracket \text{let } (a, b) = e \text{ in } f \rrbracket \gamma : \llbracket \Delta'_1, \Delta, \Delta'_2 \rrbracket \gamma \rightarrow \llbracket C \rrbracket \gamma$ defined via the diagram,

$$\begin{aligned} \llbracket \Delta'_1, \Delta, \Delta'_2 \rrbracket \gamma &\xrightarrow{m_{(\Delta'_1, \Delta), \Delta'_2}} \llbracket \Delta'_1, \Delta \rrbracket \gamma \otimes \llbracket \Delta'_2 \rrbracket \gamma \xrightarrow{m_{\Delta'_1, \Delta} \otimes \text{id}} ((\llbracket \Delta'_1 \rrbracket \gamma \otimes \llbracket \Delta \rrbracket \gamma) \otimes \llbracket \Delta'_2 \rrbracket \gamma) \\ &\quad \downarrow (\text{id} \otimes \llbracket e \rrbracket \gamma) \otimes \text{id} \\ \llbracket C \rrbracket \gamma &\xleftarrow{\llbracket f \rrbracket \gamma} ((\llbracket \Delta'_1 \rrbracket \gamma \otimes \llbracket A \rrbracket \gamma) \otimes \llbracket B \rrbracket \gamma) \otimes \llbracket \Delta'_2 \rrbracket \gamma \xleftarrow{\alpha \otimes \text{id}} (\llbracket \Delta'_1 \rrbracket \gamma \otimes (\llbracket A \rrbracket \gamma \otimes \llbracket B \rrbracket \gamma)) \otimes \llbracket \Delta'_2 \rrbracket \gamma \end{aligned}$$

$\otimes\beta$. The desired β equality for \otimes is,

$$\llbracket \text{let } (a, b) = (a', b') \text{ in } f \rrbracket \gamma = \llbracket f[a'/a, b'/b] \rrbracket \gamma$$

PROOF. The left hand side reduces as follows,

$$\begin{aligned} \llbracket \text{let } (a, b) = (a', b') \text{ in } f \rrbracket \gamma &= \llbracket f \rrbracket \gamma \circ \alpha \otimes \text{id} \circ (\text{id} \otimes \llbracket (a', b') \rrbracket \gamma) \otimes \text{id} \circ m_{\Delta, \Delta'} \\ &= \llbracket f \rrbracket \gamma \circ \alpha \otimes \text{id} \circ (\text{id} \otimes \lambda \otimes \lambda \otimes m_{a:A, b:B}) \otimes \text{id} \circ m_{\Delta, \Delta'} \\ &= \llbracket f \rrbracket \gamma \quad (\text{coherence}) \end{aligned}$$

Which is equal to the right hand side,

$$\begin{aligned} \llbracket f[a'/a, b'/b] \rrbracket \gamma &= \llbracket f \rrbracket \gamma \circ \llbracket a'/a, b'/b \rrbracket \gamma \\ &= \llbracket f \rrbracket \gamma \end{aligned} \quad (\text{coherence})$$

□

$\otimes \eta$. The desired η equality for \otimes is,

$$\llbracket \text{let } (a, b) = c' \text{ in } f[(a, b)/c] \rrbracket \gamma = \llbracket f[c'/c] \rrbracket \gamma$$

PROOF.

$$\begin{aligned} \llbracket \text{let } (a, b) = c' \text{ in } f[(a, b)/c] \rrbracket \gamma &= \llbracket f[(a, b)/c] \rrbracket \gamma \circ \alpha \otimes \text{id} \circ (\text{id} \otimes \llbracket c' \rrbracket \gamma) \otimes \text{id} \circ m_{\Delta, \Delta'} \\ &= \llbracket f \rrbracket \gamma \circ \llbracket (a, b)/c \rrbracket \gamma \circ \alpha \otimes \text{id} \circ (\text{id} \otimes \llbracket c' \rrbracket \gamma) \otimes \text{id} \circ m_{\Delta, \Delta'} \\ &= \llbracket f \rrbracket \gamma \end{aligned} \quad (\text{coherence})$$

$$\begin{aligned} \llbracket f[c'/c] \rrbracket \gamma &= \llbracket f \rrbracket \gamma \circ \llbracket c'/c \rrbracket \gamma \quad (\text{Lemma B.4}) \\ &= \llbracket f \rrbracket \gamma \quad (\text{coherence}) \end{aligned}$$

□

B.1.4 \multimap -Functions.

\multimap -Introduction. $\llbracket \lambda^{\multimap} a. e \rrbracket \gamma : \llbracket \Delta \rrbracket \gamma \rightarrow \llbracket A \multimap B \rrbracket \gamma$ is defined using the natural isomorphism $\phi : \text{Hom}(\llbracket \Delta \rrbracket \gamma \otimes \llbracket A \rrbracket \gamma, \llbracket B \rrbracket \gamma) \rightarrow \text{Hom}(\llbracket \Delta \rrbracket \gamma, \llbracket A \multimap B \rrbracket \gamma)$ that is provided by the adjunction between $\llbracket - \otimes A \rrbracket \gamma$ and $\llbracket A \multimap - \rrbracket \gamma$.

$$\llbracket \lambda^{\multimap} a. e \rrbracket \gamma = \phi(\llbracket e \rrbracket \gamma)$$

\multimap -Elimination. $\llbracket e' e \rrbracket \gamma : \llbracket \Delta, \Delta' \rrbracket \gamma \rightarrow \llbracket B \rrbracket \gamma$ is defined by the diagram,

$$\llbracket \Delta, \Delta' \rrbracket \gamma \xrightarrow{m_{\Delta, \Delta'}} \llbracket \Delta \rrbracket \gamma \otimes \llbracket \Delta' \rrbracket \gamma \xrightarrow{\text{id} \otimes \llbracket e \rrbracket \gamma} \llbracket \Delta \rrbracket \gamma \otimes \llbracket A \rrbracket \gamma \xrightarrow{\phi^{-1}(\llbracket e' \rrbracket \gamma)} \llbracket B \rrbracket \gamma$$

$\multimap \beta$. The β rule for \multimap is given by,

$$\llbracket (\lambda^{\multimap} a. e) a' \rrbracket \gamma = \llbracket e[a'/a] \rrbracket \gamma$$

PROOF.

$$\begin{aligned} \llbracket (\lambda^{\multimap} a. e) a' \rrbracket \gamma &= \phi^{-1}(\llbracket \lambda^{\multimap} a. e \rrbracket \gamma) \circ \text{id} \otimes \llbracket a' \rrbracket \gamma \circ m_{\Delta, \Delta'} \\ &= \phi^{-1}(\phi(\llbracket e \rrbracket \gamma)) \circ \text{id} \otimes \llbracket a' \rrbracket \gamma \circ m_{\Delta, \Delta'} \\ &= \llbracket e \rrbracket \gamma \circ \text{id} \otimes \llbracket a' \rrbracket \gamma \circ m_{\Delta, \Delta'} \\ &= \llbracket e \rrbracket \gamma \circ \text{id} \otimes \lambda \circ m_{\Delta, \Delta'} \\ &= \llbracket e \rrbracket \gamma \end{aligned} \quad (\text{coherence})$$

$$\begin{aligned} \llbracket e[a'/a] \rrbracket \gamma &= \llbracket e \rrbracket \gamma \circ \llbracket a'/a \rrbracket \gamma \quad (\text{Lemma B.4}) \\ &= \llbracket e \rrbracket \gamma \quad (\text{coherence}) \end{aligned}$$

□

$\multimap \eta$. The η rule for \multimap is given by,

$$\llbracket \lambda^{\multimap} a. e a \rrbracket \gamma = \llbracket e \rrbracket \gamma$$

PROOF.

$$\begin{aligned} \llbracket \lambda^{\multimap} a. e a \rrbracket \gamma &= \phi(\llbracket e a \rrbracket \gamma) \\ &= \phi(\phi^{-1}(\llbracket e \rrbracket \gamma) \circ \text{id} \otimes \llbracket a \rrbracket \gamma \circ m_{\Delta, a:A}) \\ &= \phi(\phi^{-1}(\llbracket e \rrbracket \gamma) \circ \text{id} \otimes \lambda \circ m_{\Delta, a:A}) \\ &= \phi(\phi^{-1}(\llbracket e \rrbracket \gamma)) && \text{(coherence)} \\ &= \llbracket e \rrbracket \gamma \end{aligned}$$

□

B.1.5 \multimap -Functions.

\multimap -Introduction. Just as with the other linear function type, we have an adjunction between $\llbracket A \otimes - \rrbracket \gamma$ and $\llbracket - \multimap A \rrbracket \gamma$. $\llbracket \lambda^{\multimap} a. e \rrbracket \gamma : \llbracket \Delta \rrbracket \gamma \rightarrow \llbracket B \multimap A \rrbracket \gamma$ is defined using the natural isomorphism $\psi : \text{Hom}(\llbracket A \rrbracket \gamma \otimes \llbracket \Delta \rrbracket \gamma, \llbracket B \rrbracket \gamma) \rightarrow \text{Hom}(\llbracket \Delta \rrbracket \gamma, \llbracket B \multimap A \rrbracket \gamma)$ induced by this adjunction. In particular, $\llbracket \lambda^{\multimap} a. e \rrbracket \gamma$ is given by ψ acting on the following diagram

$$\llbracket A \rrbracket \gamma \otimes \llbracket \Delta \rrbracket \gamma \xrightarrow{\lambda^{-1} \otimes \text{id}} \llbracket a : A \rrbracket \gamma \otimes \llbracket \Delta \rrbracket \gamma \xrightarrow{m_{a:A, \Delta}^{-1}} \llbracket a : A, \Delta \rrbracket \gamma \xrightarrow{\llbracket e \rrbracket \gamma} \llbracket B \rrbracket \gamma$$

\multimap -Elimination. The application of a \multimap -function, $\llbracket e^{\multimap} e' \rrbracket \gamma : \llbracket \Delta', \Delta \rrbracket \gamma \rightarrow \llbracket B \rrbracket \gamma$ defined by the diagram

$$\llbracket \Delta', \Delta \rrbracket \gamma \xrightarrow{m_{\Delta', \Delta}} \llbracket \Delta' \rrbracket \gamma \otimes \llbracket \Delta \rrbracket \gamma \xrightarrow{\llbracket e' \rrbracket \gamma \otimes \text{id}} \llbracket A \rrbracket \gamma \otimes \llbracket \Delta \rrbracket \gamma \xrightarrow{\psi^{-1}(\llbracket e \rrbracket \gamma)} \llbracket B \rrbracket \gamma$$

$\multimap \beta$. The β rule for \multimap is given by,

$$\llbracket (\lambda^{\multimap} a. e)^{\multimap} a' \rrbracket \gamma = \llbracket e[a'/a] \rrbracket \gamma$$

PROOF.

$$\begin{aligned} \llbracket (\lambda^{\multimap} a. e)^{\multimap} a' \rrbracket \gamma &= \psi^{-1}(\llbracket \lambda^{\multimap} a. e \rrbracket \gamma \circ \llbracket a' \rrbracket \gamma \otimes \text{id} \circ m_{a:A, \Delta}) \\ &= \psi^{-1}(\llbracket \lambda^{\multimap} a. e \rrbracket \gamma \circ \lambda \otimes \text{id} \circ m_{a:A, \Delta}) \\ &= \psi^{-1}(\psi(\llbracket e \rrbracket \gamma \circ m_{a:A, \Delta}^{-1} \circ \lambda^{-1} \otimes \text{id})) \circ \lambda \otimes \text{id} \circ m_{\Delta', \Delta} \\ &= \llbracket e \rrbracket \gamma \circ m_{a:A, \Delta}^{-1} \circ \lambda^{-1} \otimes \text{id} \circ \lambda \otimes \text{id} \circ m_{a:A, \Delta} \\ &= \llbracket e \rrbracket \gamma \end{aligned}$$

$$\begin{aligned} \llbracket e[a'/a] \rrbracket \gamma &= \llbracket e \rrbracket \gamma \circ \llbracket a'/a \rrbracket \gamma && \text{(Lemma B.4)} \\ &= \llbracket e \rrbracket \gamma && \text{(coherence)} \end{aligned}$$

□

$\multimap \eta$. The η rule for \multimap is given by,

$$\llbracket \lambda^{\multimap} a. e^{\multimap} a \rrbracket \gamma = \llbracket e \rrbracket \gamma$$

PROOF.

$$\begin{aligned} \llbracket \lambda^{\multimap} a. e^{\multimap} a \rrbracket \gamma &= \phi(\llbracket e^{\multimap} a \rrbracket \gamma \circ m_{a:A, \Delta}^{-1} \circ \lambda^{-1} \otimes \text{id}) \\ &= \phi(\phi^{-1}(\llbracket e \rrbracket \gamma) \circ \llbracket a \rrbracket \gamma \otimes \text{id} \circ m_{a:A, \Delta} \circ m_{a:A, \Delta}^{-1} \circ \lambda^{-1} \otimes \text{id}) \\ &= \phi(\phi^{-1}(\llbracket e \rrbracket \gamma) \circ \lambda \otimes \text{id} \circ m_{a:A, \Delta} \circ m_{a:A, \Delta}^{-1} \circ \lambda^{-1} \otimes \text{id}) \\ &= \phi(\phi^{-1}(\llbracket e \rrbracket \gamma)) \\ &= \llbracket e \rrbracket \gamma \end{aligned}$$

□

B.1.6 $\&$ -Products.

$\&$ -Introduction. $\llbracket \lambda^{\&} x. e \rrbracket \gamma : \llbracket \Delta \rrbracket \gamma \rightarrow \prod_{x:\llbracket X \rrbracket \gamma} \llbracket A \rrbracket (\gamma, x)$ is defined by the universal property of the product

$$\llbracket \lambda^{\&} x. e \rrbracket \gamma = (\llbracket e \rrbracket (\gamma, x))_{(x:\llbracket X \rrbracket \gamma)}$$

$\&$ -Elimination. $\llbracket e. \pi M \rrbracket \gamma : \llbracket \Delta \rrbracket \gamma \rightarrow \llbracket A \rrbracket (\gamma, M)$ is defined using the projection out of the product,

$$\llbracket \Delta \rrbracket \gamma \xrightarrow{\llbracket e \rrbracket \gamma} \prod_{x:\llbracket X \rrbracket \gamma} \llbracket A \rrbracket (\gamma, x) \xrightarrow{\pi_M} \llbracket A \rrbracket (\gamma, M)$$

$\&\beta$. The β law for $\&$ is given by,

$$\llbracket (\lambda^{\&} x. e) . \pi M \rrbracket (\gamma, x) = \llbracket e[M/x] \rrbracket (\gamma, x)$$

PROOF.

$$\begin{aligned} \llbracket (\lambda^{\&} x. e) . \pi M \rrbracket \gamma &= \pi_M \circ \llbracket \lambda^{\&} x. e \rrbracket \gamma \\ &= \pi_M \circ (\llbracket e \rrbracket (\gamma, x))_{(x:\llbracket X \rrbracket \gamma)} \\ &= \llbracket e \rrbracket (\gamma, M) \end{aligned}$$

by the universal property of the product.

$$\begin{aligned} \llbracket e[M/x] \rrbracket (\gamma, x) &= \llbracket e \rrbracket (\gamma, x) \circ \llbracket M/x \rrbracket (\gamma, x) \\ &= \llbracket e \rrbracket (\gamma, M) \end{aligned}$$

□

$\&\eta$. The η law for $\&$ is given by,

$$\llbracket (\lambda^{\&} x. e. \pi x) \rrbracket \gamma = \llbracket e \rrbracket \gamma$$

PROOF.

$$\begin{aligned} \llbracket (\lambda^{\&} x. e. \pi x) \rrbracket \gamma &= (\llbracket e. \pi x \rrbracket \gamma)_{x:\llbracket X \rrbracket \gamma} \\ &= (\pi_x \circ \llbracket e \rrbracket \gamma)_{x:\llbracket X \rrbracket \gamma} \\ &= \llbracket e \rrbracket \gamma \end{aligned}$$

□

by the universal property of the product.

B.1.7 \oplus -Sums.

\oplus -Introduction. $\llbracket \sigma M e \rrbracket \gamma : \llbracket \Delta \rrbracket \gamma \rightarrow \coprod_{x: \llbracket X \rrbracket \gamma} \llbracket A \rrbracket (\gamma, x)$

$$\llbracket \Delta \rrbracket \gamma \xrightarrow{\llbracket e \rrbracket \gamma} \llbracket A \rrbracket (\gamma, M) \xrightarrow{i_M} \coprod_{x: \llbracket X \rrbracket \gamma} \llbracket A \rrbracket (\gamma, x)$$

\oplus -Elimination. $\llbracket \text{let } \sigma x a = e \text{ in } e' \rrbracket \gamma : \llbracket \Delta'_1, \Delta, \Delta'_2 \rrbracket \gamma \rightarrow \llbracket C \rrbracket \gamma$ is defined in the diagram

$$\begin{array}{ccc} \llbracket \Delta'_1, \Delta, \Delta'_2 \rrbracket \gamma & \xrightarrow{m_{(\Delta'_1, \Delta), \Delta'_2}} & \llbracket \Delta'_1, \Delta \rrbracket \gamma \otimes \llbracket \Delta'_2 \rrbracket \gamma \\ & & \downarrow m_{\Delta'_1, \Delta} \otimes \text{id} \\ \left(\llbracket \Delta'_1 \rrbracket \gamma \otimes \coprod_{x: \llbracket X \rrbracket \gamma} \llbracket A \rrbracket (\gamma, x) \right) \otimes \llbracket \Delta'_2 \rrbracket \gamma & \xleftarrow{(\text{id} \otimes \llbracket e \rrbracket \gamma) \otimes \text{id}} & \left(\llbracket \Delta'_1 \rrbracket \gamma \otimes \llbracket \Delta \rrbracket \gamma \right) \otimes \llbracket \Delta'_2 \rrbracket \gamma \\ \downarrow d & & \downarrow \text{id} \otimes m_{\Delta'_1, \Delta}^{-1} \\ \coprod_{x: \llbracket X \rrbracket \gamma} \left(\llbracket \Delta'_1 \rrbracket (\gamma, x) \otimes \llbracket A \rrbracket (\gamma, x) \right) \otimes \llbracket \Delta'_2 \rrbracket (\gamma, x) & \xrightarrow{\coprod_{x: \llbracket X \rrbracket \gamma} m_{(\Delta'_1, a: A), \Delta'_2}^{-1}} & \coprod_{x: \llbracket X \rrbracket \gamma} \llbracket \Delta'_1, a : A, \Delta'_2 \rrbracket (\gamma, x) \\ & & \downarrow \llbracket [e'](\gamma, x) \rrbracket_{(x: \llbracket X \rrbracket \gamma)} \\ \llbracket C \rrbracket \gamma & \xleftarrow{\quad} & \llbracket C \rrbracket (\gamma, x) \end{array}$$

where d is the distributivity morphism, and the last morphism implicitly weakens $\llbracket C \rrbracket$.

$\oplus \beta$. The β rule for \oplus is given by,

$$\llbracket \text{let } \sigma x a = \sigma M a' \text{ in } e' \rrbracket \gamma = \llbracket e' [M/x, a'/a] \rrbracket \gamma$$

PROOF.

$$\begin{aligned} \llbracket \text{let } \sigma x a = \sigma M a' \text{ in } e' \rrbracket \gamma &= \llbracket [e'](\gamma, x) \rrbracket_{(x: \llbracket X \rrbracket \gamma)} \circ \coprod_{x: \llbracket X \rrbracket \gamma} (m^{-1}) \circ d \circ (\text{id} \otimes \llbracket \sigma M a' \rrbracket \gamma) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket [e'](\gamma, x) \rrbracket_{(x: \llbracket X \rrbracket \gamma)} \circ \coprod_{x: \llbracket X \rrbracket \gamma} m^{-1} \circ d \circ (\text{id} \otimes (i_M \circ \llbracket a' \rrbracket \gamma)) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket [e'](\gamma, x) \rrbracket_{(x: \llbracket X \rrbracket \gamma)} \circ \coprod_{x: \llbracket X \rrbracket \gamma} m^{-1} \circ d \circ (\text{id} \otimes i_M) \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket [e'](\gamma, x) \rrbracket_{(x: \llbracket X \rrbracket \gamma)} \circ \coprod_{x: \llbracket X \rrbracket \gamma} m^{-1} \circ i_M \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket [e'](\gamma, M) \rrbracket \circ m^{-1} (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket [e'](\gamma, M) \rrbracket \end{aligned} \tag{coherence}$$

$$\begin{aligned} \llbracket e' [M/x, a'/a] \rrbracket \gamma &= \llbracket e' \rrbracket \gamma \circ \llbracket M/x, a'/a \rrbracket \gamma \\ &= \llbracket e' \rrbracket (\gamma, x) \end{aligned} \tag{Lemma B.4}$$

□

$\oplus \eta$. It suffices to show

$$\llbracket \text{let } \sigma x a = c' \text{ in } f[(\sigma x a)/c] \rrbracket \gamma = \llbracket f[c'/c] \rrbracket \gamma = \llbracket f \rrbracket \gamma$$

First, expanding the left hand side, we have.

PROOF.

$$\begin{aligned}
& \llbracket \text{let } \sigma \times a = c' \text{ in } f[(\sigma \times a)/c] \rrbracket \gamma \\
&= \llbracket f[(\sigma \times a)/c] \rrbracket \gamma_{(x: \llbracket X \rrbracket \gamma)} \circ \coprod_{x: \llbracket X \rrbracket \gamma} (m^{-1}) \circ d \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\
&= \llbracket f \rrbracket \gamma \circ (\text{id} \otimes i_x) \otimes \text{id}_{(x: \llbracket X \rrbracket \gamma)} \circ \coprod_{x: \llbracket X \rrbracket \gamma} (m^{-1}) \circ d \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\
&= \llbracket f \rrbracket \gamma \circ [(\text{id} \otimes i_x) \otimes \text{id} \circ (m^{-1})]_{(x: \llbracket X \rrbracket \gamma)} \circ d \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m
\end{aligned}$$

Since the domain has the universal property of a coproduct (due to distributivity), to prove this is equal to $\llbracket f \rrbracket \gamma$, it is sufficient to prove they are equal when composed with the injections:

$$\begin{aligned}
& \llbracket f \rrbracket \gamma \circ [(\text{id} \otimes i_x) \otimes \text{id} \circ (m^{-1})]_{(x: \llbracket X \rrbracket \gamma)} \circ d \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \circ (\text{id} \otimes i_y) \otimes \text{id} \\
&= \llbracket f \rrbracket \gamma \circ [(\text{id} \otimes i_x) \otimes \text{id} \circ (m^{-1})]_{(x: \llbracket X \rrbracket \gamma)} \circ d \circ (\text{id} \otimes i_y) \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\
&\quad \text{(naturality)} \\
&= \llbracket f \rrbracket \gamma \circ [(\text{id} \otimes i_x) \otimes \text{id} \circ (m^{-1})]_{(x: \llbracket X \rrbracket \gamma)} \circ i_y \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\
&\quad \text{(naturality)} \\
&= \llbracket f \rrbracket \gamma \circ (\text{id} \otimes i_y) \otimes \text{id} \circ (m^{-1}) \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\
&= \llbracket f \rrbracket \gamma \circ (m^{-1}) \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \circ (\text{id} \otimes i_y) \otimes \text{id} \\
&= \llbracket f \rrbracket \gamma \circ (\text{id} \otimes i_y) \otimes \text{id} \quad \text{(coherence)}
\end{aligned}$$

□

B.1.8 Equalizer.

Equalizer Introduction. $\llbracket \langle e \rangle \rrbracket \gamma : \llbracket \Delta \rrbracket \gamma \rightarrow \llbracket \{e \mid f e = g e\} \rrbracket \gamma$ where $\llbracket e \rrbracket \gamma : \llbracket \Delta \rrbracket \gamma \rightarrow \llbracket A \rrbracket \gamma$ and $\llbracket f \rrbracket \gamma \circ \llbracket e \rrbracket \gamma = \llbracket g \rrbracket \gamma \circ \llbracket e \rrbracket \gamma$. By the universal property of the equalizer the preceding equality induces a unique morphism $\llbracket \Delta \rrbracket \gamma \rightarrow \text{Eq}(\llbracket f \rrbracket \gamma, \llbracket g \rrbracket \gamma) = \llbracket \{e \mid f e = g e\} \rrbracket \gamma$. Define $\llbracket \langle e \rangle \rrbracket \gamma$ to be this map.

Equalizer Elimination. $\llbracket e \cdot \pi \rrbracket \gamma : \llbracket \Delta \rrbracket \gamma \rightarrow \llbracket A \rrbracket \gamma$ is defined using the map π_{eq} from $\text{Eq}(\llbracket f \rrbracket \gamma, \llbracket g \rrbracket \gamma)$ to the domain of f and g .

$$\llbracket e \cdot \pi \rrbracket \gamma = \pi_{\text{eq}} \circ \llbracket e \rrbracket \gamma$$

Equalizer β . The β rule for $\{e \mid f e = g e\}$ is given as,

$$\llbracket \langle e \rangle \cdot \pi \rrbracket \gamma = \llbracket e \rrbracket \gamma$$

where $\llbracket f \rrbracket \gamma \circ \llbracket e \rrbracket \gamma = \llbracket g \rrbracket \gamma \circ \llbracket e \rrbracket \gamma$.

PROOF. In \mathbf{C} the universal property of $\text{Eq}(\llbracket f \rrbracket \gamma, \llbracket g \rrbracket \gamma)$ implies that the following diagram commutes, implying the β rule.

$$\begin{array}{ccccc}
& & \text{Eq}(\llbracket f \rrbracket \gamma, \llbracket g \rrbracket \gamma) & \xrightarrow{\pi_{\text{eq}}} & \llbracket A \rrbracket \gamma & \xrightarrow[\llbracket g \rrbracket \gamma]{\llbracket f \rrbracket \gamma} & \llbracket B \rrbracket \gamma \\
& \llbracket \langle e \rangle \rrbracket \gamma \uparrow & \nearrow & & \nearrow & & \\
& \llbracket \Delta \rrbracket \gamma & & & \llbracket e \rrbracket \gamma & &
\end{array}$$

□

Equalizer η . The η rule for $\{e \mid f e = g e\}$ is given as,

$$\llbracket \langle e . \pi \rangle \rrbracket \gamma = \llbracket e \rrbracket \gamma$$

PROOF. Likewise, the universal property of $\text{Eq}(\llbracket f \rrbracket \gamma, \llbracket g \rrbracket \gamma)$ implies η rule via this diagram.

$$\begin{array}{ccc} \text{Eq}(\llbracket f \rrbracket \gamma, \llbracket g \rrbracket \gamma) & \xrightarrow{\pi_{\text{eq}}} & \llbracket A \rrbracket \gamma \xrightarrow[\llbracket g \rrbracket \gamma]{\llbracket f \rrbracket \gamma} \llbracket B \rrbracket \gamma \\ \llbracket e \rrbracket \gamma \uparrow & \nearrow \llbracket e . \pi \rrbracket \gamma & \\ \llbracket \Delta \rrbracket \gamma & & \end{array}$$

□

B.2 Grammar Semantics Respects Additional Axioms

We verify that the denotational semantics validates each of the axioms we have assumed.

Axiom 3.1. Distributivity

THEOREM B.5 (☞). In **Gr**, $\llbracket \lambda^{-\circ} e . \text{let } \sigma f e' = e \text{ in } \lambda^{\&x} . e' . \pi x \rrbracket \gamma$ has an inverse.

PROOF. Distributivity is true in the denotational semantics, as the category **Gr** is a topos, which are well-known to be distributive. The following map forms the desired inverse,

$$\lambda \gamma . \lambda w . \lambda p . (\lambda (\lambda x . (p x) . \text{fst}) . , \lambda x . (p x) . \text{snd})$$

□

Axiom 3.3. σ -Disjointness

THEOREM B.6 (☞). In **Gr**, $\llbracket \sigma x \rrbracket \gamma$ and $\llbracket \sigma x' \rrbracket \gamma$ are disjoint for $x \neq x'$.

PROOF. This is trivially true, as the denotation of linear sum types is as Σ types in **Gr**. For any input, the first projections of $\llbracket \sigma x \rrbracket \gamma$ is x and the first projection of $\llbracket \sigma x' \rrbracket \gamma$ is x' . Because $x \neq x'$, the images of $\llbracket \sigma x \rrbracket \gamma$ and $\llbracket \sigma x' \rrbracket \gamma$ cannot agree. □

Axiom 3.4. String is strongly equivalent to \top .

THEOREM B.7 (☞). In **Gr**, $\llbracket ! \rrbracket \gamma$ has an inverse.

PROOF. Because $\llbracket \top \rrbracket \gamma w$ is a singleton set for all γ and w , it suffices to show that $\llbracket \text{String} \rrbracket \gamma w$ is likewise a singleton.

First, we prove by induction on w that $\llbracket \text{String} \rrbracket \gamma w$ is a retract of $\llbracket \bigoplus_{w:\text{String}} [w] \rrbracket \gamma w$, a sum over a nonlinear type of strings. Then, again by induction on w , we show that $\llbracket \bigoplus_{w:\text{String}} [w] \rrbracket \gamma w$ is isomorphic to $\llbracket \top \rrbracket \gamma w$.

Each $\llbracket [w] \rrbracket \gamma w'$ is inhabited if and only if w is equal to w' , and the parses of w for $\llbracket [w] \rrbracket \gamma$ are unique. That is, $\llbracket \bigoplus_{w:\text{String}} [w] \rrbracket \gamma w$ is a singleton set. So, $\llbracket \text{String} \rrbracket \gamma w$ is a retract of a singleton set, and is itself singleton. Thus, $\llbracket \text{String} \rrbracket \gamma w \cong \llbracket \top \rrbracket \gamma w$ in **Set**. □