

# Intrinsic Verification of Parsers and Formal Grammar Theory in Dependent Lambek Calculus

STEVEN SCHAEFER, University of Michigan, USA

NATHAN VARNER, University of Michigan, USA

PEDRO H. AZEVEDO DE AMORIM, University of Oxford, UK

MAX S. NEW, University of Michigan, USA

We present Dependent Lambek Calculus ( $\text{Lambek}^D$ ), a domain-specific dependent type theory for verified parsing and formal grammar theory. In  $\text{Lambek}^D$ , 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 types as a mathematical notion of formal grammars. Based on this denotational semantics, we have made a prototype implementation of  $\text{Lambek}^D$  using a shallow embedding in the Agda proof assistant. All of our examples parsers have been implemented in this prototype implementation.

## 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. For example, in an extensive experiment finding bugs in C compilers [35], an early version of the formally verified CompCert C compiler only contained bugs in the then unverified parsing component [23]. Bugs in parsers undermine the overall correctness theorem for a verified system: an incorrectly parsed C program will be compiled correctly but this is not very useful if it did not correctly correspond to the actual source program. Eventually, a correct parser was implemented using an automaton that is formally verified to implement an LR grammar [16].


It is entirely understandable from an engineering perspective *why* verified parsing was not part of the initial releases of CompCert: parsing algorithms and formal grammars 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 [5, 7, 21]. Each new grammar formalism is extended with its own independent verified implementation.

In this work, we present the design of Dependent Lambek Calculus ( $\text{Lambek}^D$ ), a domain-specific language for formal verification of parsers. A key property is that  $\text{Lambek}^D$  is an *extensible* framework for verification of parsers in that it supports the definition of grammar formalisms of unrestricted complexity. That is,  $\text{Lambek}^D$  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,  $\text{Lambek}^D$  is not a verified parser generator that compiles regular expressions to deterministic finite automata, but is instead is a domain specific language *for writing* such a verified parser generator.

The design of Lambek<sup>D</sup> is an extension of Joachim Lambek’s *syntactic calculus* [20]. 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 linear 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 [18].

The substructural nature of Lambek<sup>D</sup> is well-aligned with the requirements intrinsic to parsing and to the theory of formal languages, where strings constitute a very clear notion of resource that cannot be duplicated, reordered, or dropped. Moreover, the constructive aspect of Lambek<sup>D</sup> ensures that verification of parsers written in the calculus are *correct-by-construction*. The type system is powerful enough that derivations of a term type checking carry intrinsic proofs of correctness. Parsers written in Lambek<sup>D</sup> 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<sup>D</sup> as a shallowly embedded domain-specific language in the Cubical Agda proof assistant [34]. We have implemented many example grammars and parsers in our system including regular expressions, non-deterministic and deterministic automata, as well as some example context-free grammars and parsers based on LL(1) and LR(1) automata. Throughout this paper, we will use  to mark results that are mechanized in our Agda development.

Our Agda prototype is based on a *denotational semantics* of Lambek<sup>D</sup>. The core idea of the denotational semantics is [8], Elliott describes a formal grammar as type-level predicates on strings that prove language membership. 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<sup>D</sup> can be so interpreted as an abstract formal grammar in this sense, and that linear terms are 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<sup>D</sup>): 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
- Demonstration of how to encode many common grammar formalisms (regular expressions, (non-)deterministic automata, context-free grammars) and parser formalisms within our type theory.
- A prototype implementation of Lambek<sup>D</sup> in Agda with all examples mechanized.
- A denotational semantics for Lambek<sup>D</sup> that shows that the parsers are in fact verified to be correct and soundness of the equational theory.

This paper begins in Section 2 by studying small example programs from Lambek<sup>D</sup> 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<sup>D</sup> for relating familiar grammar and automata formalisms as well as building concrete parsers. Then in Section 5, we give

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

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

Fig. 1. "ac" matches  $('a' \oplus 'b') \otimes 'c'$ 

a denotational semantics that makes precise the connection between Lambek<sup>D</sup> 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<sup>D</sup>, 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  $w$ . Similarly,  $w$  matches  $A \otimes B$  if  $w$  can be split into two strings  $w_A$  and  $w_B$  that match  $A$  and  $B$ , respectively.

For a type  $A$ , a parse tree of a string  $w$  is represented as a term of type  $A$  in the context  $[w]$ , where  $[w]$  is a context with one variable for each character of  $w$ . 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, while contraction allows for the same variable to be used twice, but in a parse tree, every character must be accounted for exactly once. That is, we want to prevent the following erroneous derivations,

$$a : 'a', b : 'b' \not\vdash a : 'a' \quad a : 'a', b : 'b' \not\vdash (a, a) : 'a' \otimes 'a'$$

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,

$$a : 'a', b : 'b' \not\vdash (b, a) : 'b' \otimes 'a'$$

*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 *linear* type.  $A^*$  has two constructors:  $\text{nil}$ , which builds a parse of type  $A^*$  from nothing; and  $\text{cons}$ , 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

```

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

```

Fig. 2. Kleene Star as an inductive type

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

```

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

```

Fig. 3. "ab" matches  $('a'^* \otimes 'b') \oplus 'c'$ 

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 by themselves are not consumed upon usage, and so are *non-linear* values themselves. 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" matches 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 :  $\uparrow ('a'^*)$`  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 parse 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  $h$  is more compactly 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 a very operational perspective of how to parse it. For this reason, most parsers are based on compiling a grammar to a corresponding notion of automaton, which is readily implemented. To implement these algorithms

$$\begin{array}{c}
\frac{}{\cdot \vdash \text{nil} : A^*} \quad \frac{\frac{a_1 : A, a_2 : A, a_* : A^* \vdash \text{cons}(a_1, \text{cons}(a_2, a_*)) : A^*}{a' : A \otimes A, a_* : A^* \vdash g := \text{let } a_1 \otimes a_2 = a' \text{ in } \text{cons}(a_1, \text{cons}(a_2, a_*)) : A^*}}{\cdot \vdash f := \lambda^{-\circ} a. \lambda^{-\circ} a_* . g a a_* : (A \otimes A) \multimap A^* \multimap A^*} \\
\hline
a : (A \otimes A)^* \vdash h := \text{fold}(\text{nil}, f)(a) : A^*
\end{array}$$

$$\begin{array}{l}
h : \uparrow((A \otimes A)^* \multimap A^*) \\
h \text{ nil} = \text{nil} \\
h (\text{cons } (a_1 \otimes a_2) \text{ as}) = \text{cons } a_1 (\text{cons } a_2 (h \text{ as}))
\end{array}$$

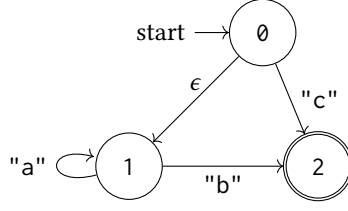
Fig. 4. A parse transformer for abstract grammars

in Lambek<sup>D</sup>, we need a way to represent automata as types in the same way we can represent 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 `Fin 3` is the finite type containing inhabitants  $\{0, 1, 2\}$ . We can think of this as defining three mutually recursive inductive types `Trace 0`, `Trace 1`, and `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  $\epsilon$ . The constructor `stop` terminates a trace in the accepting state 2. The constructors `1to1`, `1to2`, `0to2` 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 `0to1` behaves similarly, except its transition is labeled with the empty string  $\epsilon$ . Therefore, `0to1` 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 lambda 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 (`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<sup>D</sup> is based on the dependent linear-non-linear calculus (LNL<sub>D</sub>) and Lambek calculus, also known as non-commutative linear logic. [18, 20]. As in LNL<sub>D</sub>, Lambek<sup>D</sup> 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<sub>D</sub>'s design is that, as in Lambek calculus [20], the linear typing is *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<sub>D</sub> was enhanced with special connectives



```

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

$\frac{}{\Gamma \text{ ctx}}$	$\frac{\Gamma \text{ ctx}}{\Gamma \vdash X \text{ type}}$	$\frac{\Gamma \vdash X \text{ type}}{\Gamma \vdash X \text{ small}}$	$\frac{\Gamma \vdash X \text{ type} \quad \Gamma \vdash Y \text{ type}}{\Gamma \vdash X \equiv Y \text{ type}}$	$\frac{\Gamma \vdash X \text{ type}}{\Gamma \vdash M : X}$
$\frac{\Gamma \vdash M : X \quad \Gamma \vdash N : X}{\Gamma \vdash M \equiv N : X}$	$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \Delta \text{ lin. ctx.}}$	$\frac{\Gamma \text{ ctx}}{\Gamma \vdash A \text{ lin. type}}$	$\frac{\Gamma \vdash A \text{ lin. type} \quad \Gamma \vdash B \text{ lin. type}}{\Gamma \vdash A \equiv B}$	
	$\frac{\Gamma \vdash \Delta \text{ lin. ctx.} \quad \Gamma \vdash A \text{ lin. type}}{\Gamma; \Delta \vdash e : A}$		$\frac{\Gamma; \Delta \vdash e : A \quad \Gamma; \Delta \vdash f : A}{\Gamma; \Delta \vdash e \equiv f : A}$	

Fig. 6. Formation rules

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<sup>D</sup> are shown in Figure 6.  $\Gamma$  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;  $\Delta$  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  $\Gamma$ , but note that linear types  $A$  cannot depend on any *linear* variables in  $\Delta$ . We include definitional equality judgments for both kinds of type and term judgments as well. Additionally, we have a judgment  $\Gamma \vdash X \text{ small}$  which is used in the definition of universe types.

### 3.1 Non-linear Typing

The non-linear types of Lambek<sup>D</sup> include the standard dependent types for  $\Pi, \Sigma$ , extensional equality, natural numbers, booleans, unit and empty types, and we assume function extensionality[15]. We present the other non-linear type constructions in Figure 7. 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 a judgment saying when a non-linear type is small and define the universe to

$$\begin{array}{c}
\Gamma \vdash U \text{ type} \quad \frac{\Gamma \vdash M : U}{\Gamma \vdash [M] \text{ type}} \quad \frac{\Gamma \vdash X \text{ type}}{\Gamma \vdash [X] : U} \quad [ [X] ] \equiv X \quad \frac{\Gamma \vdash M : U}{\Gamma \vdash [ [M] ] \equiv M : U} \quad \Gamma \vdash L \text{ type} \quad \frac{\Gamma \vdash M : L}{\Gamma \vdash [M] \text{ lin. type}} \\
\\
\frac{\Gamma \vdash A \text{ lin. type}}{\Gamma \vdash [A] : L} \quad [ [A] ] \equiv A \quad \frac{\Gamma \vdash M : U}{\Gamma \vdash [ [M] ] \equiv M : U} \quad \frac{\Gamma \vdash A \text{ lin. type}}{\Gamma \vdash \uparrow A \text{ type}} \quad \frac{\Gamma; \cdot \vdash e : A}{\Gamma \vdash e : \uparrow A} \quad \frac{\Gamma \vdash M : \uparrow A}{\Gamma; \cdot \vdash M : A}
\end{array}$$

Fig. 7. Non-linear types (selection)

internalize precisely this judgment [4, 12]. These universe types are needed so that we can define types by recursion on natural numbers. Next, 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. 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 [11, 17].

### 3.2 Linear Typing

We give an overview of the linear types and terms in Figure 8. The equational theory for these types is straightforward  $\beta\eta$  equivalence and included in the appendix. First, the linear variable rule says that a linear variable can be used if it is the *only* variable in the context.

First, 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 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  $\Delta_2$  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  $\lambda$  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 (elided) 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  $\Sigma$  and  $\Pi$  connectives of ordinary dependent type theory, respectively. The indexed  $\&$  is defined by a  $\lambda$  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”  $\Sigma$  type: it has an injection introduction rule  $\sigma$ , 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

$$\begin{array}{c}
\frac{\Gamma \vdash I \text{ lin. type}}{\Gamma \vdash I \text{ lin. type}} \quad \frac{\Gamma \vdash A \text{ lin. type} \quad \Gamma \vdash B \text{ lin. type}}{\Gamma \vdash A \otimes B \text{ lin. type}} \quad \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}} \quad \frac{\Gamma, x : X \vdash A \text{ lin. type}}{\Gamma \vdash \bigoplus_{x:X} A \text{ lin. type}} \quad \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; a : A \vdash a : A} \quad \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} \quad \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} \quad \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} \quad \frac{\Gamma; \Delta' \vdash e' : A \quad \Gamma; \Delta \vdash e : A \multimap B}{\Gamma; \Delta, \Delta' \vdash e' e : B} \\
\\
\frac{\Gamma, x : X; \Delta \vdash e : A}{\Gamma; \Delta \vdash \lambda^{\otimes} x. e : \bigotimes_{x:X} (x : X). A} \quad \frac{\Gamma; \Delta \vdash e : \bigotimes_{x:X} (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} \quad \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\}} \quad \frac{\Gamma; \Delta \vdash e : \{a \mid f a = g a\}}{\Gamma; \Delta \vdash e. \pi : A}
\end{array}$$

Fig. 8. Linear types and terms (selection)

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 can't 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  $(A+B) \& C \cong (A \& C) + (B \& C)$ . In its most general form, the axiom says that the definable function  $\bigoplus_{f: \prod_{x:X} Y(x)} \bigotimes_{x:X} A \times (f x) \multimap \bigotimes_{x:X} \bigoplus_{y:Y(x)} A \times y$

has an inverse. Second, we need that the different constructors of  $\bigoplus$  are disjoint. We add this by adding the axiom that for any  $A : X \rightarrow \mathcal{L}$  and  $x \neq x' : X$  there is a function  $\uparrow (\{b \mid \sigma x \circ \pi_1 b = \sigma x' \circ \pi_2 b\} \multimap \emptyset)$  where  $b : A(x) \& A(x')$ , i.e., the grammar of pairs of an  $a : A(x)$  and an  $a' : A(x')$  such that  $\sigma x a = \sigma x' a'$  is empty.



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

Fig. 9. Strictly positive functors and indexed inductive linear types

### 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 [1, 25]. The syntax is given in Figure 9. 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  $\text{el}$  that interprets it as such a type constructor, as well as an operator  $\text{map}$  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 doesn't mention any type variables  $K$ , as well as tensor products and additive conjunction and disjunction of type expressions. We additionally add equations in the appendix that say that the  $\text{el}/\text{map}$  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  $\text{roll}$ , 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  $\mu F x \multimap Ax$  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  $\mu F$  into an *initial algebra* for the functor  $\text{el}(F)$ . That is the  $\beta$  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  $\eta$  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  $\eta$  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  $f, g : \uparrow (\mu F \times \multimap A \times)$  equal, it suffices to implement a function  $\text{ind} : \uparrow (\mu F \times \multimap \{a \mid f a = g a\})$  such that  $\text{ind}(a) \equiv a$ . Then an inductive-style proof can be implemented by constructing  $\text{ind}$  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. First for each character in our alphabet, 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  $\text{read} : \uparrow (\top \multimap \mathbf{String})$  that intuitively “reads” the input string from the input and makes it available. It is important that the input type of  $\text{read}$  is  $\top$ , which can control any amount of resources, and not  $I$  which controls no resources. Further, we add an axiom that  $\lambda s. \text{read}(!s) \equiv \lambda s. s$  where  $!$  is the unique function  $\uparrow (\mathbf{String} \multimap \top)$ , i.e., that if we have a string, but then throw it away and read it from the input, then in fact that is equivalent to the string we were given originally. 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<sup>D</sup> to conducting 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. By encoding well-established formal grammar concepts, we ensure that our framework remains grounded in the foundational principles of formal language theory while opening the door to compositional formal verification of parsers.

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 are isomorphic [3]. Using linear types as grammars, we can define both of these notions of equivalence in Lambek<sup>D</sup>.

*Definition 4.1.* Grammars  $A$  and  $B$  are *weakly equivalent* if there exists parse transformers  $f : \uparrow (A \multimap B)$  and  $g : \uparrow (B \multimap A)$ .  $A$  is a *retract* of  $B$  if  $A$  and  $B$  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  $\text{inl}$  and  $\text{inr}$ . 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<sup>D</sup> in a clever way:

**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 parse 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[33]. 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** ( $\mathcal{C}$ ). *If  $B$  is unambiguous and  $A$  is a retract of  $B$  then  $A$  is unambiguous. If a disjunction  $\bigoplus_{x:X} A(x)$  is unambiguous then each  $A(x)$  is unambiguous.*

From the first principle, 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 **String**  $\equiv I \oplus \text{Char} \oplus (\text{Char} \otimes \text{Char} \otimes \text{String})$ . Then by the second principle we have that  $I$  and **Char** and each literal ' $c$ ' are unambiguous as well.

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 doesn’t parse, but is far too weak as a specification: we can trivially implement a parser for any type by always returning  $\text{inr}$ . 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 4.4.** A parser for a linear type  $A$  is a function  $\uparrow (\text{String} \multimap A \oplus A_{\neg})$ , where  $A_{\neg}$  is a linear type that is *disjoint* from  $A$  in that we can implement a function  $\uparrow (A \& A_{\neg} \multimap \emptyset)$ .

Here we replace  $\top$  in our partial parser type with a type  $A_{\neg}$  that we can think of as a negation of  $A$ . The function  $\uparrow (A \& A_{\neg} \multimap \emptyset)$  ensures that it is impossible for  $A$  and  $A_{\neg}$  to match 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 our axiom that  $\oplus$  constructors are disjoint.

**LEMMA 4.5** ( $\mathcal{C}$ ). *If  $A \oplus A_{\neg}$  is unambiguous, then  $A$  and  $A_{\neg}$  are disjoint.*

Writing a parser as a linear term in this way is an intrinsic verification of the *soundness* of the parser completely for free from the typing: any  $\text{inl}$  parse that we return *must* correspond to a parse tree of the input string. Further if we verify the disjointness property we then also get the *completeness* of the parser as well, that it never fails to generate an  $A$  parse when it is possible.

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

```


data TraceN : (s : N.states) → L where
  nil : ↑(&[ s : N.states ] N.isAcc → TraceN s)
  cons : ↑(&[ t : N.transitions ]
    ('N.label t' → TraceN (N.dst t) → TraceN (N.src t)))
  εcons : ↑(&[ t : N.εtransitions ]
    (TraceN (N.εdst t) → TraceN (N.εsrc t)))

data TraceD : (s : D.states) (b : Bool) → L where
  nil : ↑(&[ s : D.states ] TraceD s D.isAcc s)
  cons : ↑(&[ c : Char ] &[ s : D.states ] &[ b : Bool ]
    ('c' → TraceD (D.δ c s) b → TraceD s b))

```

Fig. 10. Traces of an NFA  $N$  and a DFA  $D$ 

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 4.6 (). *If  $A$  is weakly equivalent to  $B$  then any parser for  $A$  can be extended to a parser for  $B$ .*

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

#### 4.1 Regular Expressions and Finite Automata

In this section, 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 combine this using Lemma 4.6 to get a verified regex parser.

A regular expressions in Lambek<sup>D</sup> 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 we define the linear type of traces as in Figure 10

In Fig. 10 we define a linear type of traces through an NFA  $N$ .  $\text{Trace}_N$  is an inductive type indexed by the starting state of the trace  $s : N.\text{states}$ . Traces in  $N$  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 linearly 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  $\epsilon$ -transition then we may use `εcons` to pull it back along the  $\epsilon$ -transition and construct a trace beginning at the source of the  $\epsilon$ -transition. As a shorthand, write  $\text{Parse}_N$  for the accepting traces out of  $N.\text{init}$ .

$\text{Trace}_D$ , 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.

```

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. 11. Parser/printer for DFA traces

Because DFAs are deterministic, we are able to prove that their type of traces are unambiguous and define a parser for them. In particular we show that for any start state  $s$ ,  $\bigoplus(b : \text{Bool}) . \text{Trace}_D s b$  is a retract of  $\text{String}$ . That is, first we construct a function  $\text{parse}_D$  that is a parser for  $\text{Trace}_D s \text{ true}$ , with  $\text{Trace}_D s \text{ false}$  being the disjoint type used. The fact that these types are disjoint follows by showing that this function is part of a retraction, i.e., that there is only one way to trace through a deterministic automaton.

The parser,  $\text{parse}_D$ , is defined by recursion on strings in Fig. 11. If this string is empty, then  $\text{parse}_D$  defines a linear function that terminates a trace at the input state  $s$ . If the string is nonempty, then  $\text{parse}_D$  walks forward in  $D$  from the input state  $s$  by the character at the head of the string. The inverse,  $\text{print}_D$  is defined by recursion on traces. If the trace is defined via  $\text{nil}$ , then  $\text{print}_D$  returns the empty string. Otherwise, if the trace is defined by  $\text{cons}$  then  $\text{parse}_D$  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 4.7** ( $\mathcal{U}$ ).  *$\text{parse}_D s$  is a parser for  $\text{Trace}_D s \text{ true}$ .*

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 [26]. 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.

**THEOREM 4.8** (DETERMINIZATION,  $\mathcal{U}$ ). *Given an NFA  $N$ , there exists a DFA  $D$  such that  $\text{Parse}_N$  is weakly equivalent to  $\text{Parse}_D$ .*

**PROOF.** Define the states of  $D$  to be the  $\mathbb{P}_\epsilon(N.\text{states})$  — the type of  $\epsilon$ -closed<sup>1</sup> subsets of  $N.\text{states}$ . A subset is accepting in  $D$  if it contains an accepting state from  $N$ . Construct the initial state of  $D$  as the  $\epsilon$ -closure of  $N.\text{init}$ . Lastly define the transition function of  $D$  to send the subset  $X$  under the character  $c$  to the  $\epsilon$ -closure of all the states reachable from  $X$  via a transition labeled with the character  $c$ .

We demonstrate the weak equivalence between  $\text{Parse}_N$  and  $\text{Parse}_D$  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 and build a term

$$\text{NtoD} : \uparrow \left( \text{Trace}_N s \text{ true} \multimap \big\&_{X : D.\text{states}} \big\&_{s \text{ in } X : X \ni s} \text{Trace}_D X \text{ true} \right)$$

<sup>1</sup>A subset of states  $X$  is  $\epsilon$ -closed if for every  $s \in X$  and  $\epsilon$ -transition  $s \xrightarrow{\epsilon} s'$  we have  $s' \in X$ .

```

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

```


that maps a trace in  $N$  from an 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 induction hypothesis and build a parse transformer

$$DtoN : \uparrow \left( \text{Trace}_D \ X \ \text{true} \rightarrow \bigoplus_{s:N.\text{states}} \bigoplus_{s \text{ in } X: X \ni s} \text{Trace}_N \ s \ \text{true} \right)$$

□

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<sup>D</sup>.

**THEOREM 4.9 (THOMPSON'S CONSTRUCTION, ).** *Given a regular expression  $R$ , there exists an NFA  $N$  such that  $R$  is strongly equivalent to  $\text{Trace}_N(N.init)$ .*

**PROOF.** We use a variant of Thompson's construction [32], showing that NFAs are, up to strong equivalence, closed under each type operation for regular expressions.


□

## 4.2 Context-free grammars

Next, we give two examples for parsing context-free grammars. Context-free grammars (CFG) can be encoded in our type theory in a similar way to regular expressions, as CFGs are equivalent to the formalism of  $\mu$ -regular expressions, where the Kleene star is replaced by an arbitrary fixed point operation[22].

A simple example of a CFG is the Dyck grammar of balanced parentheses, which we define in ?? Dyck is a grammar over the alphabet  $\{ "(", ")", " ", "NUM" \}$ . 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  $\text{Parse}_M$  is strongly equivalent to Dyck.

The Dyck language is an example of an  $LL(\emptyset)$  language, one that can be parsed top-down with no lookahead[29]. This means we can implement it simply as an *infinite* state deterministic automaton, in Figure 12. 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 the parser and printer for DFAs, and so  $\bigoplus(s : M.\text{states}). \bigoplus(b : \text{Bool}). \text{Trace}_M \ s \ b$  is likewise unambiguous.

**THEOREM 4.10 ().** *Dyck and  $\text{Parse}_M$  are strongly equivalent. And therefore we can construct a parser for Dyck.*

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 13 we define it 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 automaton has four different

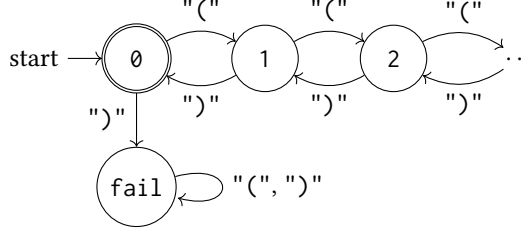


Fig. 12. Automaton M for the Dyck grammar

“states”, each with access to a natural number “stack”. The “opening” state 0 expects either an open 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 M. Here `NotStartsWithRP` is defined as  $I \oplus (((' \oplus '+' \oplus \text{NUM}) \otimes \top)$ . In the “closing” state C if we observe a close paren we decrement the count and continue to the D state. In the “multiplying” state M, we succeed if the string ends and the count is 0, and if we see a plus we continue to the 0 state. Additionally, since the automaton need also parse all of the incorrect strings, we add additional failing cases. It is straightforward to implement a parser for this lookahead automaton, generalizing the approach for deterministic automata.

**THEOREM 4.11** ( $\text{⌘}$ ). *We construct a parser for `Exp` by showing it is weakly equivalent to  $O \emptyset$  true.*

### 4.3 Unrestricted Grammars

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

$$\text{Reify } P = \bigoplus_{w : \text{String}' \times P \ w} \bigoplus [w]$$

where  $["] = I$  and  $[c : w] = 'c' \otimes [w]$ .

This reification operation on functions  $\text{String}' \rightarrow U$  is incredibly expressive, as it allows to sidestep our linear typing connectives and utilize the whole of nonlinear dependent type theory to define a grammar. For example, given a Turing machine `Tur` one may define a non-linear predicate `accepts : String' → U` such that `accept w` is equal to  $\top$  if `T` accepts `w` and equal to  $\emptyset$  otherwise. Then, `Reify accepts` is a linear type that captures precisely the strings accepted by `Tur`. That is,  $\text{Lang}(\text{Reify accepts})$  is recursively enumerable — the most general class of languages in the Chomsky hierarchy.

**THEOREM 4.12** ( $\text{⌘}$ ). *For any Turing machine, we can construct a grammar in  $\text{Lambek}^D$  that accepts the same language as the Turing machine.*

## 5 DENOTATIONAL SEMANTICS AND IMPLEMENTATION

To justify our assertion that  $\text{Lambek}^D$  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

```

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

```

Fig. 13. Associative arithmetic expressions and a corresponding lookahead automaton

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*[8]:

*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  $\mathbf{Gr}$  to be the category whose objects are formal grammars and morphisms are parse transformers.

This category is equivalent to the slice category  $\mathbf{Set}/\Sigma^*$  and as such 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<sup>D</sup>. These categorical properties are precisely what is required to interpret all of the linear type and term constructors.

## 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<sup>D</sup>:

- (1) For each non-linear context  $\Gamma \text{ ctx}$ , we define a set  $\llbracket \Gamma \rrbracket$ .
- (2) For each non-linear type  $\Gamma \vdash X \text{ type}$ , and element  $\gamma \in \llbracket \Gamma \rrbracket$ , we define a set  $\llbracket X \rrbracket_\gamma$ .
- (3) For each linear type  $\Gamma \Delta \text{ lin. type}$  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  $\Gamma \Delta \text{ lin. ctx.}$ .
- (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  $\Gamma \vdash X \text{ small}$ , then  $\llbracket X \rrbracket_\gamma$  is a small set.
- (2) If  $\Gamma \vdash X \equiv X'$  then for every  $\gamma$ ,  $\llbracket X \rrbracket_\gamma = \llbracket X' \rrbracket_\gamma$ .
- (3) If  $\Gamma \vdash A \equiv A'$  then for every  $\gamma$ ,  $\llbracket A \rrbracket_\gamma = \llbracket A' \rrbracket_\gamma$ .
- (4) If  $\Gamma \vdash M \equiv M' : X$  then for every  $\gamma$ ,  $\llbracket M \rrbracket_\gamma = \llbracket M' \rrbracket_\gamma$ .
- (5) If  $\Gamma; \Delta \vdash e \equiv e' : A$  then for every  $\gamma$ ,  $\llbracket e \rrbracket_\gamma = \llbracket e' \rrbracket_\gamma$ .

The interpretation of dependent types as sets is standard [15]. We present the concrete descriptions of the semantics of linear types, as well as our non-standard non-linear types in Figure 14. 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  $w_2$

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

Fig. 14. Grammar Semantics

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:

$$\llbracket \uparrow (A \multimap B) \rrbracket \gamma = \llbracket A \multimap B \rrbracket \gamma \varepsilon = \prod_{w'} \llbracket A \rrbracket \gamma w' \rightarrow \llbracket B \rrbracket \gamma w'$$

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* [1, 10].

*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  $f$ ;  $\Pi_g$  is the dependent product operation and  $\Sigma_h$  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  $K$ ,  $\text{Var}$ , etc. on functors in the obvious way that matches the definitional behavior of  $\text{el}$  and  $\text{map}$ . 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 polynomial 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  $\text{IW}$  trees which are already available in the cubical library of Agda [31]. Then an element  $F \in \llbracket X \rightarrow \text{SPF } X \rrbracket_Y$  is an  $\llbracket X \rrbracket_Y$ -indexed family of polynomial functors from  $\llbracket X \rrbracket_Y \times \Sigma^*$  to  $\Sigma^*$ , and taking the product of these constructs a polynomial functor from  $\llbracket X \rrbracket_Y \times \Sigma^*$  to itself. Then  $\llbracket \mu F \rrbracket_Y$  is defined to be the initial algebra of this functor, and the initial algebra structure is used to interpret  $\text{roll}$ ,  $\text{fold}$  and the corresponding axioms.

The remaining details of the interpretation of linear terms as parse transformers and verification of the equational axioms is a relatively straightforward extension of existing semantics of linear logic in monoidal categories, and is included in the appendix [30].

### 5.3 Agda Implementation

This denotational semantics in grammars and parse transformers serves as the basis for our prototype implementation of  $\text{Lambek}^D$  in cubical Agda. The implementation is a shallow embedding, meaning that rather than formalizing a syntax of  $\text{Lambek}^D$  types and terms, we work directly with Agda types and a definition of a formal grammar as a function  $\text{String} \rightarrow \text{Set}$ . Then we implement each of the type and term constructors of  $\text{Lambek}^D$  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 we use extensively. 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 that are possible in  $\text{Lambek}^D$ . The main difference between our shallow embedding and  $\text{Lambek}^D$  is that our linear terms are written in a combinator-style, without being able to use named variables in linear terms. 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, but with the goal of still making it integrate with an existing proof assistant.

## 6 DISCUSSION AND FUTURE WORK

*Grammars as (Linear) Types.* Lambek's original syntactic calculus [20] describes a logical system for linguistic derivations, and it can be given semantics inside of a non-commutative biclosed monoidal category [19]. This led to many uses of non-commutative linear logic and lambda calculi in linguistics [2] – including mechanized categorial grammar parsers [13, 27]. 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, and so cannot be used to express verified parsers.

The most similar prior work to our own is Luo's Lambek calculus with dependent types [24]. Their design differs from our own in allowing linear types to depend on other linear types and supporting directed  $\Pi$  and  $\Sigma$  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<sup>D</sup> is a practical system for describing grammar formalisms and parsers, and it is unclear if these could be implemented in their calculus.

The usage of a simple type system to reason about regular expressions up to weak equivalence was introduced by Frisch and Cardelli [9]. Henglein and Nielsen [14] later adapt this approach by providing a type system for regular expressions whose semantics were in one-to-one correspondence with the set of parse trees. Elliott [8] gives a similar semantic interpretation of regular grammars as type-level predicates on strings. The type system of Lambek<sup>D</sup> extends these semantic interpretations of to a broader class of grammars and also gives a formal syntax and equational theory for the parse transformers that these prior works lack.

*Relation to Separation Logic.* Lambek<sup>D</sup> is similar in spirit to separation logic [28]. Semantically, they are closely related: linear types in Lambek<sup>D</sup> 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 [17]. The monoidal structure in both cases is an instance of the category-theoretic notion of Day convolution monoidal structure [6]. 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<sup>D</sup>.

*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 aren't needed in later stages of the overall program. We can define a semantic action in Lambek<sup>D</sup> 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.

*Implementation.* Our Agda prototype implementation serves as a useful proof of concept for showing what can be implemented in Lambek<sup>D</sup>, 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<sup>D</sup>.

*Type Checking and Semantic Analysis.* Our focus in this work has been on the verification of parsers for grammars over strings, but because Lambek<sup>D</sup> 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 systems as linear types in a modified version of Lambek<sup>D</sup> 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.

## REFERENCES

- [1] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25:e5, January 2015.
- [2] W. Buszkowski. Type Logics in Grammar. In Ryszard Wójcicki, Daniele Mundici, Ewa Orłowska, Graham Priest, Krister Segerberg, Alasdair Urquhart, Heinrich Wansing, Vincent F. Hendricks, and Jacek Malinowski, editors, *Trends in Logic*, volume 21, pages 337–382. Springer Netherlands, Dordrecht, 2003. Series Title: Trends in Logic.
- [3] Noam Chomsky. Formal properties of grammars. *Handbook of Mathematical Psychology*, II:323–418, 1963.
- [4] Thierry Coquand. Presheaf model of type theory.
- [5] Nils Anders Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 285–296, Baltimore Maryland USA, September 2010. ACM.
- [6] Brian John Day. *Construction of biclosed categories*. PhD thesis, University of New South Wales PhD thesis, 1970.
- [7] Romain Edelmann, Jad Hamza, and Viktor Kunčák. Zippy ll(1) parsing with derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, page 1036–1051, New York, NY, USA, 2020. Association for Computing Machinery.
- [8] Conal Elliott. Symbolic and automatic differentiation of languages. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–18, August 2021.
- [9] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming*, pages 618–629, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [10] Nicola Gambino and Martin Hyland. Wellfounded Trees and Dependent Polynomial Functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 210–225, Berlin, Heidelberg, 2004. Springer.
- [11] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.
- [12] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal Dependent Type Theory. *Logical Methods in Computer Science*, Volume 17, Issue 3, July 2021.
- [13] Maxime Guillaume, Sylvain Pogodalla, and Vincent Tourneur. Acgtk: A toolkit for developing and running abstract categorial grammars. In *Functional and Logic Programming: 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15–17, 2024, Proceedings*, page 13–30, Berlin, Heidelberg, 2024. Springer-Verlag.
- [14] Fritz Henglein and Lasse Nielsen. Regular expression containment: coinductive axiomatization and computational interpretation. *ACM SIGPLAN Notices*, 46(1):385–398, January 2011.
- [15] Martin Hofmann. *Syntax and Semantics of Dependent Types*, page 79–130. Publications of the Newton Institute. Cambridge University Press, 1997.
- [16] Jacques-Henri Jourdan, Francois Pottier, and Xavier Leroy. Validating LR(1) Parsers. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and Helmut Seidl, editors, *Programming Languages and Systems*, volume 7211, pages 397–416. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.
- [17] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*, pages 256–269, New York, NY, USA, September 2016. Association for Computing Machinery.
- [18] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating Linear and Dependent Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 17–30, Mumbai India, January 2015. ACM.
- [19] J. Lambek. *Categorial and Categorical Grammars*, pages 297–317. Springer Netherlands, Dordrecht, 1988.
- [20] Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.
- [21] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. CoStar: A verified ALL(\*) parser. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 420–434, Virtual Canada, June 2021. ACM.
- [22] Haas Leiß. Towards Kleene Algebra with recursion. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, and Michael M. Richter, editors, *Computer Science Logic*, pages 242–256, Berlin, Heidelberg, 1992. Springer.
- [23] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.
- [24] Zhaohui Luo. Substructural calculi with dependent types. EasyChair Preprint 421, EasyChair, 2018.
- [25] Georgi Nakov and Fredrik Nordvall Forsberg. Quantitative Polynomial Functors. In *27th International Conference on Types for Proofs and Programs (TYPES 2021)*, pages 10:1–10:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
- [26] M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

- [27] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. Center for the Study of Language and Information/SRI, 2011.
- [28] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002. ISSN: 1043-6871.
- [29] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256, October 1970.
- [30] R. A. G. Seely. Linear logic, \*-autonomous categories and cofree coalgebras. In *Categories in computer science and logic (Boulder, CO, 1987)*, volume 92 of *Contemp. Math.*, pages 371–382. Amer. Math. Soc., Providence, RI, 1989.
- [31] The Agda Community. Cubical Agda Library, February 2024.
- [32] Ken Thompson. Programming Techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.
- [33] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- [34] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.
- [35] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers.

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

Fig. 15. Strictly positive functors functorial actions

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

Fig. 16. Context rules

$$\frac{\Gamma; a : A, \Delta \vdash e : B}{\Gamma; \Delta \vdash \lambda^{\circ-} a. e : B \multimap A} \quad \frac{\Gamma; \Delta \vdash e : B \multimap A \quad \Gamma; \Delta' \vdash e' : A}{\Gamma; \Delta', \Delta \vdash e \multimap^{\circ-} e' : B}$$

Fig. 17. Linear terms (extended)

## A SYNTAX

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

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

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

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

$$\boxed{\Gamma; \Delta \vdash e \equiv e' : A} \quad \frac{\Gamma; \Delta, a : A \vdash e : C \quad \Gamma; \Delta' \vdash e' : A}{\Gamma; \Delta, \Delta' \vdash (\lambda^{\circ-} a. e) e' \equiv e\{e'/a\} : C} \quad \frac{\Gamma; \Delta \vdash e : A \multimap B}{\Gamma; \Delta \vdash e \equiv \lambda^{\circ-} a. e a : A \multimap B}$$

$$\frac{\Gamma; a : A, \Delta \vdash e : C \quad \Gamma; \Delta' \vdash e' : A}{\Gamma; \Delta, \Delta' \vdash (\lambda^{\circ-} a. e) e' \equiv e\{e'/a\} : C} \quad \frac{\Gamma; \Delta \vdash e : B \multimap A}{\Gamma; \Delta \vdash e \equiv \lambda^{\circ-} a. e a : B \multimap A}$$

$$\frac{\Gamma, x : X \vdash e : A \quad \Gamma \vdash M : X}{\Gamma; \Delta \vdash (\lambda^{\&} x. e) M \equiv e\{M/x\} : C} \quad \frac{\Gamma; \Delta \vdash e : \&(x : X). A}{\Gamma; \Delta \vdash e \equiv \lambda^{\&} x. e x : \&(x : X). A}$$

$$\frac{\Gamma; \Delta_1, \Delta_2 \vdash e : C}{\Gamma; \Delta_1, \Delta_2 \vdash \text{let } () = () \text{ in } e \equiv e : C} \quad \frac{\Gamma; \Delta_2 \vdash e : I \quad \Gamma; \Delta_1, a : A, \Delta_3 \vdash e' : C}{\Gamma; \Delta_1, \Delta_3 \vdash \text{let } () = e \text{ in } e'\{()/a\} \equiv e'\{e/a\} : C}$$

$$\frac{\Gamma; \Delta_2 \vdash e : A \quad \Gamma; \Delta_3 \vdash e' : B \quad \Gamma; \Delta_1, a : A, b : B, \Delta_4 \vdash e'' : C}{\Gamma; \Delta_1, \Delta_2, \Delta_3, \Delta_4 \vdash \text{let } (a, b) = (e, e') \text{ in } e'' \equiv e''\{e/a, e'/b\} : C}$$

$$\frac{\Gamma; \Delta_2 \vdash e : A \otimes B \quad \Gamma; \Delta_1, c : A \otimes B, \Delta_3 \vdash e' : C}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{let } (a, b) = e \text{ in } e'\{(a, b)/c\} \equiv e'\{e/c\} : C}$$

$$\frac{\Gamma \vdash M : X \quad \Gamma; \Delta_2 \vdash e : A \quad \Gamma, x : X \vdash \Delta_1, a : A, \Delta_3}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{let } \sigma x a = \sigma M \text{ in } e' \equiv e'\{M/x, e/a\} : C}$$

$$\frac{\Gamma; \Delta_1, y : \bigoplus(x : X). A, \Delta_2 \vdash e' : C \quad \Gamma; \Delta_2 \vdash e : \bigoplus(x : X). A}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{let } \sigma x a = e \text{ in } e'\{\sigma x a/y\} \equiv e'\{e/y\} : C}$$

$$\frac{\Gamma; \Delta \vdash e : A \quad \Gamma; \Delta \vdash f e \equiv g e}{\Gamma; \Delta \vdash \langle e \rangle. \pi \equiv e : A} \quad \frac{\Gamma; \Delta \vdash e : \{e \mid f e = g e\}}{\Gamma; \Delta \vdash \langle e. \pi \rangle \equiv e : \{e \mid f e = g e\}}$$

Fig. 18. Judgmental equality for linear terms

It is straightforward, but laborious to establish that all forms in the type theory that are parameterized by a non-linear context  $\Gamma$  support the admissible actions of a substitution  $\gamma \in \text{Subst}(\Gamma', \Gamma)$  in Figure 19.

*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  $\Delta$ :

$$\text{Subst}(\Delta', \cdot) = \{\cdot \mid \Delta' = \cdot\}$$

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

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. 20, frequently using the inversion principle to split the substitution into



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

Fig. 19. Non-linear substitution

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

Fig. 20. Action of substitution on linear terms

constituent components. By induction on linear term and equality judgments, we establish the following admissible rules for  $\delta \in \text{Subst}(\Delta', \Delta)$ :

$$\frac{\Gamma; \Delta \vdash e : A}{\Gamma; \Delta' \vdash e[\delta] : A} \quad \frac{\Gamma; \Delta \vdash e \equiv f : A}{\Gamma; \Delta' \vdash e[\delta] \equiv f[\delta'] : A}$$

## 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.

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

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

*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  $\delta$ :

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

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

**THEOREM B.3.** *For any  $\Gamma \vdash \Delta_1, \Delta_2$  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  $m_{\Delta_1, \Delta_2}$  by recursion on  $\Delta_2$ .

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

□

**LEMMA B.4.** *For each term  $\Delta \vdash e : A$  and substitution  $\delta : \text{Subst}(\Delta', \Delta)$ , the semantics of  $\delta$  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<sup>D</sup>. 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.**  $\llbracket \text{let } () = e \text{ in } e' \rrbracket_Y : \llbracket \Delta'_1, \Delta, \Delta'_2 \rrbracket_Y \rightarrow \llbracket C \rrbracket_Y$  defined in the following diagram,

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

We demonstrate that the denotations of the introduction and elimination forms for I obey the  $\beta$  and  $\eta$  equalities for I.

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

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

**PROOF.**

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

□

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

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

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.**

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

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  $\llbracket \Delta'_1, \Delta'_2 \rrbracket_Y$  to itself, it is equal to the identity by the coherence theorem for monoidal categories.

Further by Lemma B.4,

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

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

### B.1.3 Tensor.

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

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

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

$$\begin{array}{c} \llbracket \Delta'_1, \Delta, \Delta'_2 \rrbracket_Y \xrightarrow{m_{(\Delta'_1, \Delta), \Delta'_2}} \llbracket \Delta'_1, \Delta \rrbracket_Y \otimes \llbracket \Delta'_2 \rrbracket_Y \xrightarrow{m_{\Delta'_1, \Delta} \otimes \text{id}} (\llbracket \Delta'_1 \rrbracket_Y \otimes \llbracket \Delta \rrbracket_Y) \otimes \llbracket \Delta'_2 \rrbracket_Y \\ \downarrow (\text{id} \otimes \llbracket e \rrbracket_Y) \otimes \text{id} \\ \llbracket C \rrbracket_Y \xleftarrow{\llbracket f \rrbracket_Y} ((\llbracket \Delta'_1 \rrbracket_Y \otimes \llbracket A \rrbracket_Y) \otimes \llbracket B \rrbracket_Y) \otimes \llbracket \Delta'_2 \rrbracket_Y \xleftarrow{\alpha \otimes \text{id}} (\llbracket \Delta'_1 \rrbracket_Y \otimes (\llbracket A \rrbracket_Y \otimes \llbracket B \rrbracket_Y)) \otimes \llbracket \Delta'_2 \rrbracket_Y \end{array}$$

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

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

PROOF. The left hand side reduces as follows,

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

Which is equal to the right hand side,

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

$\square$

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

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

PROOF.

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

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

$\square$

### 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  $\beta$  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  $\eta$  rule for  $\multimap$  is given by,

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

PROOF.

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

□

### 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  $\psi$  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  $\beta$  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  $\eta$  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*.  $\llbracket \lambda^{\&x}. e \rrbracket_\gamma : \llbracket \Delta \rrbracket_\gamma \rightarrow \prod_{x:X} \llbracket A \rrbracket_\gamma(\gamma, x)$  is defined by the universal property of the product

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

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

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

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

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

PROOF.

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

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  $\eta$  law for  $\&$  is given by,

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

PROOF.

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

□

by the universal property of the product.

#### B.1.7 $\oplus$ -Sums.

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

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

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

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

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

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

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

PROOF.

$$\begin{aligned} \llbracket \text{let } \sigma \times a = \sigma M a' \text{ in } e' \rrbracket_Y &= \llbracket e' \rrbracket_{(Y, X)} \circ \prod_{x:X} (m^{-1}) \circ d \circ (\text{id} \otimes \llbracket \sigma M a' \rrbracket_Y) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket e' \rrbracket_{(Y, X)} \circ \prod_{x:X} m^{-1} \circ d \circ (\text{id} \otimes (i_M \circ \llbracket a' \rrbracket_Y)) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket e' \rrbracket_{(Y, X)} \circ \prod_{x:X} m^{-1} \circ d \circ (\text{id} \otimes i_M) \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket e' \rrbracket_{(Y, X)} \circ \prod_{x:X} m^{-1} \circ i_M \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket e' \rrbracket_Y \circ m^{-1} (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket e' \rrbracket_Y (M) \quad (\text{coherence}) \\ \llbracket e' [M/x, a'/a] \rrbracket_Y &= \llbracket e' \rrbracket_Y \circ \llbracket M/x, a'/a \rrbracket_Y \quad (\text{Lemma B.4}) \\ &= \llbracket e' \rrbracket_Y (Y, X) \end{aligned}$$

□

$\oplus \eta$ . It suffices to show

$$\llbracket \text{let } \sigma \times a = c' \text{ in } f[\sigma \times a/c] \rrbracket_Y = \llbracket f[c'/c] \rrbracket_Y = \llbracket f \rrbracket_Y$$

First, expanding the left hand side, we have.

PROOF.

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

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

$$\begin{aligned} \llbracket f \rrbracket_Y \circ [(\text{id} \otimes i_X) \otimes \text{id} \circ (m^{-1})]_{(X, X)} \circ d \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \circ (\text{id} \otimes i_Y) \otimes \text{id} \\ &= \llbracket f \rrbracket_Y \circ [(\text{id} \otimes i_X) \otimes \text{id} \circ (m^{-1})]_{(X, X)} \circ d \circ (\text{id} \otimes i_Y) \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \quad (\text{naturality}) \\ &= \llbracket f \rrbracket_Y \circ [(\text{id} \otimes i_X) \otimes \text{id} \circ (m^{-1})]_{(X, X)} \circ i_Y \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \quad (\text{naturality}) \\ &= \llbracket f \rrbracket_Y \circ (\text{id} \otimes i_Y) \otimes \text{id} \circ (m^{-1}) \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \\ &= \llbracket f \rrbracket_Y \circ (m^{-1}) \otimes \text{id} \circ (\text{id} \otimes \lambda) \otimes \text{id} \circ m \otimes \text{id} \circ m \circ (\text{id} \otimes i_Y) \otimes \text{id} \\ &= \llbracket f \rrbracket_Y \circ (\text{id} \otimes i_Y) \otimes \text{id} \quad (\text{coherence}) \end{aligned}$$



□

### B.1.8 Equalizer.

*Equalizer Introduction.*  $\llbracket \langle e \rangle \rrbracket_\gamma : \llbracket \Delta \rrbracket_\gamma \rightarrow \{\{e \mid f e = g e\}\}_\gamma$  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  $\llbracket \Delta \rrbracket_\gamma \rightarrow \text{Eq}(\llbracket f \rrbracket_\gamma, \llbracket g \rrbracket_\gamma) = \{\{e \mid f e = g e\}\}_\gamma$ . 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  $\pi_{\text{eq}}$  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  $\beta$ .* The  $\beta$  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 **C** the universal property of  $\text{Eq}(\llbracket f \rrbracket_\gamma, \llbracket g \rrbracket_\gamma)$  implies that the following diagram commutes, implying the  $\beta$  rule.

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

□

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

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

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

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

□