Intrinsic Verification of Parsers in Dependent Lambek Calculus

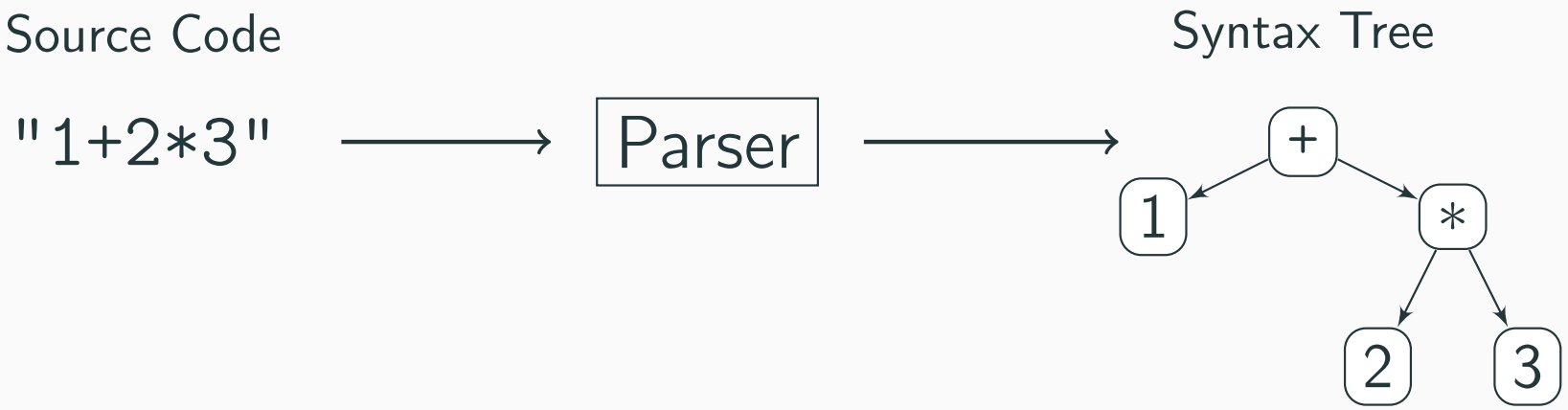
Steven Schaefer [1] Nathan Varner [1] Pedro H. Azevedo de Amorim [2]
Max S. New [1]

May 14, 2025

[1]University of Michigan

[2]University of Oxford

Parsing



- Parsing flat strings into structured data representations is ubiquitous problem in software.
- Incorrect/buggy parsers lead to junk data and unverified parser implementations are a common source of security vulnerabilities.
- Good target for formal verification: precise specifications, error-prone to implement.

How do we write (verified) parsers?

- Write the parser manually. Verify it against a one-off specification of the grammar.

How do we write (verified) parsers?

- Write the parser manually. Verify it against a one-off specification of the grammar.
- Implement a parser generator for some class of grammars.

How do we write (verified) parsers?

- Write the parser manually. Verify it against a one-off specification of the grammar.
- Implement a parser generator for some class of grammars.

In each development we formalize different notions of formal grammar, different automata formalisms. What is a reusable core to allow us to write new verified parsers and parser generators more easily?

How do we write (verified) parsers?

- Write the parser manually. Verify it against a one-off specification of the grammar.
- Implement a parser generator for some class of grammars.

In each development we formalize different notions of formal grammar, different automata formalisms. What is a reusable core to allow us to write new verified parsers and parser generators more easily?

What is the right language for implementing verified parsers?

A Language of Grammars and Parsers

Our proposal: Dependent Lambek Calculus (Lambek^D), a domain-specific type theory for defining formal grammars implementing verified parsers and formal grammar theory.

- An ordered linear typing foundation for formal grammar theory.
- Where soundness of parsers follows for free from the type discipline.
- Based on a simple denotational semantics.
- Prototype implementation using a shallow embedding of combinator syntax in cubical Agda.
- Case studies: verification of regex parsing via translation to NFAs and determinization, as well as hand-written LL(1) parsers.

Soundness and Completeness of Parsers

A formal grammar A defines for each string w a set of valid parse trees $[\![A]\!]w$.

A **parser** for grammar A is a partial function from strings to parse trees.

- Soundness: any parse tree produced by the parser is a valid parse of the input string
- Completeness: if any parse tree for the input string exists, the parser succeeds in producing one.

Our approach: soundness follows for free from type discipline.
Completeness requires proof.

1. Overview of Dependent Lambek Calculus

2. Formal Grammar Theory and Parsing in Lambek^D

3. Semantics and Implementation

4. Future Work

1. Overview of Dependent Lambek Calculus

2. Formal Grammar Theory and Parsing in Lambek^D

3. Semantics and Implementation

4. Future Work

Overview

Dependent Lambek Calculus

- Builds on Lambek's categorial grammars which define formal grammars using the structure of monoidal categories.
- We use a syntax based on non-commutative linear logic, extended with dependency on non-linear types and indexed inductive linear types.

Grammars	**Linear Types**
Grammar A	Linear type A
Parse of string w	$w \vdash M : A$
Parser	$\top \vdash M : A \oplus A_{\neg}$
Parse transformer	$\Delta \vdash M : A$

Finite Grammars

Fix an alphabet Σ

- For each character $c \in \Sigma$ a linear base type `'c'`
- Non-symmetric tensor product $A \otimes B$ and unit I. Analogues of sequencing and ε in regular expressions
- Nullary and Binary disjunction (0, $A \oplus B$).

Ordered Linear Typing

$$\frac{}{a : A \vdash a : A} \qquad \frac{\Delta \vdash e : A \qquad \Delta' \vdash e' : B}{\Delta, \Delta' \vdash (e, e') : A \otimes B}$$

$$\frac{\Delta_2 \vdash e : A \otimes B \qquad \Delta_1, a : A, b : B, \Delta_2 \vdash e' : C}{\Delta_1, \Delta_2, \Delta_3 \vdash \mathsf{let}\ (a, b) = e\ \mathsf{in}\ e' : C} \qquad \frac{}{\cdot \vdash () : I}$$

$$\frac{\Delta_2 \vdash e : I \qquad \Delta_1, \Delta_3 \vdash e' : C}{\Delta_1, \Delta_2, \Delta_3 \vdash \mathsf{let}\ () = e\ \mathsf{in}\ e' : C}$$

Additionally $\beta\eta$ rules

$$\mathsf{let}\ (a_1, a_2) = (e_1, e_2)\ \mathsf{in}\ e' = e'[e_1/a_1, e_2/a_2]$$

$$e[a : A \otimes B] = \mathsf{let}\ (a_1, a_2) = a\ \mathsf{in}\ e[(a_1, a_2)/a]$$

$$\mathsf{let}\ () = ()\ \mathsf{in}\ e' = e'$$

$$e[a : I] = \mathsf{let}\ () = a\ \mathsf{in}\ e[()/a]$$

Why Ordered Linear Typing?

Want to be able to represent A parses of a string, e.g., `"cat"` as a term

$$\boxed{x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash e : A}$$

Why Ordered Linear Typing?

Want to be able to represent A parses of a string, e.g., `"cat"` as a term

$$\boxed{x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash e : A}$$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'}$$

Why Ordered Linear Typing?

Want to be able to represent A parses of a string, e.g., `"cat"` as a term

$$\boxed{x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash e : A}$$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'}$$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \not\vdash (y, (x, z)) : \texttt{'a'} \otimes \texttt{'c'} \otimes \texttt{'t'}$$

Why Ordered Linear Typing?

Want to be able to represent A parses of a string, e.g., `"cat"` as a term

$$\boxed{x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash e : A}$$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'}$$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \not\vdash (y, (x, z)) : \texttt{'a'} \otimes \texttt{'c'} \otimes \texttt{'t'}$$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \not\vdash (x, (y, (z, z))) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \otimes \texttt{'t'}$$

Why Ordered Linear Typing?

Want to be able to represent A parses of a string, e.g., `"cat"` as a term

$$\boxed{x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash e : A}$$

$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'}$

$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \not\vdash (y, (x, z)) : \texttt{'a'} \otimes \texttt{'c'} \otimes \texttt{'t'}$

$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \not\vdash (x, (y, (z, z))) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \otimes \texttt{'t'}$

$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \not\vdash (y, z) : \texttt{'a'} \otimes \texttt{'t'}$

Why Ordered Linear Typing?

Want to be able to represent A parses of a string, e.g., `"cat"` as a term

$$\boxed{x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash e : A}$$

$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'}$

$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \not\vdash (y, (x, z)) : \texttt{'a'} \otimes \texttt{'c'} \otimes \texttt{'t'}$

$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \not\vdash (x, (y, (z, z))) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \otimes \texttt{'t'}$

$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \not\vdash (y, z) : \texttt{'a'} \otimes \texttt{'t'}$

Ordered linearity is what ensures that parsers are **sound-by-construction**

Recursive Grammars

Can encode regular and context-free grammars using inductive linear types:

```
data A* : L where
  nil : ↑(A*)
  cons : ↑(A ⊸ A* ⊸ A*)
```

Introduction rules are given by the constructors, elimination given by the corresponding fold.

$$\text{fold} : \uparrow A \to \uparrow (A \multimap B \multimap B) \to \uparrow (A^* \multimap B)$$

Lambek^D includes inductive types specified by any strictly positive (indexed) linear type expression.

Linear function Types

Because the tensor product is not symmetric, we get two different linear function types $A \multimap B$ and $B \mathrel{\circ\!\!-} A$, which add variables to the left or right side of the context:

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

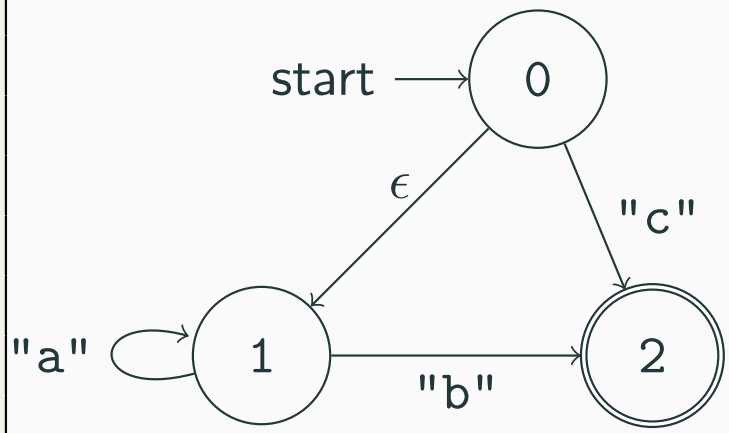
$$\frac{\Gamma; a : A, \Delta \vdash e : B}{\Gamma; \Delta \vdash \lambda^{\mathrel{-\!\!\circ}} a.\, e : B \mathrel{\circ\!\!-} A} \qquad \frac{\Gamma; \Delta \vdash e : A \qquad \Gamma; \Delta' \vdash e' : B \mathrel{\circ\!\!-} A}{\Gamma; \Delta, \Delta' \vdash e' \mathrel{{}^{\circ\!\!-}} e : B}$$

Automata

To encode automata, we use indexed inductive linear types, where the indices are non-linear data.

```
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))
```



Context-Free Grammars

Context-free grammars can be translated to (mutually) inductive linear data types. E.g., for balanced parentheses.

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

with elimination given by its corresponding fold

$$\text{fold} : \uparrow A \to \uparrow (\texttt{'('} \multimap A \multimap \texttt{')'} \multimap A \multimap A) \to \uparrow (\texttt{Dyck} \multimap A)$$

Dependent Linear Types

Linear types and terms can depend on non-linear data. Allows us to define two new linear type connectives, indexed versions of conjunction and disjunction:

Dependent Linear Types

Linear types and terms can depend on non-linear data. Allows us to define two new linear type connectives, indexed versions of conjunction and disjunction:

$$\bigoplus_{x:X} A$$

the linear version of a Σ types, rules are those of a weak Σ:

$$\frac{\Gamma \vdash M : X \quad \Gamma ; \Delta \vdash e : A[M/x]}{\Gamma ; \Delta \vdash \sigma\, M\, e : \bigoplus_{x:X} A}$$

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

Binary ($A \oplus B$) and nullary (0) versions are definable picking the index to be non-linear booleans.

Dependent Linear Types

Linear types and terms can depend on non-linear data. Allows us to define two new linear type connectives, indexed versions of conjunction and disjunction:

$$\mathop{\&}_{x:X} A$$

the linear version of a Π type

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

$$\frac{\Gamma; \Delta \vdash e : \&(x : X).A \qquad \Gamma \vdash M : X}{\Gamma; \Delta \vdash e.\pi M : A[M/x]}$$

Binary ($A \& B$) and nullary ($\top$) versions are definable picking the index to be non-linear booleans.

Dependent Linear Types

Final connective: for a linear type A, $\uparrow A$ is a non-linear type of "pure" elements of A. Plays a similar role to $!$ in linear logic or the persistence modality $\Box$ in separation logic.

We make the coercion invisible in the syntax for convenience:

$$\frac{\Gamma; \cdot \vdash e : A}{\Gamma \vdash e : \uparrow A} \qquad \frac{\Gamma \vdash e : \uparrow A}{\Gamma; \cdot \vdash e : A}$$

Unrestricted Complexity

Dependency on non-linear data and $\bigoplus$ are powerful enough to encode grammars of arbitrary complexity.

E.g. if $P : \texttt{String} \to \texttt{Type}$ is a non-linear type, we can define a grammar

$$\bigoplus_{w:\texttt{String}} \bigoplus_{x:P\ w} \lceil w \rceil$$

whose w parses are precisely Pw. where $\lceil w \rceil$ is a kind of singleton grammar for concrete strings:

$$\lceil \texttt{""} \rceil = I$$
$$\lceil c :: w \rceil = \texttt{'c'} \otimes \lceil w \rceil$$

Example: for any Turing machine T we can define
$\texttt{Accepts}\ T : \texttt{String} \to \texttt{Type}$ of accepting traces in ordinary dependent type theory and then lift it to a linear type.

1. Overview of Dependent Lambek Calculus

2. Formal Grammar Theory and Parsing in Lambek^D

3. Semantics and Implementation

4. Future Work

Defining Parsers

How to implement a parser for a grammar A?

- $s : \top \vdash e : A$

Defining Parsers

How to implement a parser for a grammar A?

- $s : \top \vdash e : A$ too strong, represents parsers only for total grammars that parse every input string

Defining Parsers

How to implement a parser for a grammar A?

- $s : \top \vdash e : A$ too strong, represents parsers only for total grammars that parse every input string
- $s : \top \vdash e : A \oplus \top$

Defining Parsers

How to implement a parser for a grammar A?

- $s : \top \vdash e : A$ too strong, represents parsers only for total grammars that parse every input string
- $s : \top \vdash e : A \oplus \top$ the type of a partial parser sound but allows for incompleteness

Defining Parsers

How to implement a parser for a grammar A?

- $s : \top \vdash e : A$ too strong, represents parsers only for total grammars that parse every input string
- $s : \top \vdash e : A \oplus \top$ the type of a partial parser sound but allows for incompleteness
- $s : \top \vdash e : A \oplus A_\neg$ where $A_\neg$ is a "complement grammar" for A, i.e., they are mutually exclusive $A \& A_\neg \vdash 0$

Soundness is by construction, completeness comes from showing $A \& A_\neg \vdash 0$

Grammar-specific axioms

So far: standard ordered linear logic + inductives + dependency on non-linear data. We also assume some axioms that don't follow from just linear type theory (distributivity of $\oplus$ over &, disjointness of constructors...)

To do formal grammar theory, we need one axiom to tell us that we are parsing finite strings.

- Define $\texttt{Char} = \bigoplus_{c \in \Sigma} \texttt{'c'}$
- And $\texttt{String} = \texttt{Char}*$

Axiom: `String` is isomorphic to $\top$.

Grammar-specific axioms

So far: standard ordered linear logic + inductives + dependency on non-linear data. We also assume some axioms that don't follow from just linear type theory (distributivity of $\oplus$ over &, disjointness of constructors...)

To do formal grammar theory, we need one axiom to tell us that we are parsing finite strings.

- Define $\texttt{Char} = \bigoplus_{c \in \Sigma} \texttt{'c'}$
- And $\texttt{String} = \texttt{Char}*$

Axiom: `String` is isomorphic to $\top$. Some consequences: `String` is unambiguous, `Char` is unambiguous, can access the underlying string of a parse tree.

Parser for the Dyck Grammar

```
data Dyck : LinTy where
  nil : ↑ Dyck
  bal : ↑('(' ⊸ Dyck ⊸ ')' ⊸ Dyck ⊸ Dyck)
```

$$\text{"()()"} \vdash \mathit{bal}\ \mathit{l1}\ \mathit{nil}\ \mathit{r1}\ (\mathit{bal}\ \mathit{l2}\ \mathit{nil}\ \mathit{r2}) : \mathit{Dyck}$$

Parser for the Dyck Grammar

```
data Dyck : LinTy where
  nil : ↑ Dyck
  bal : ↑('(' ⊸ Dyck ⊸ ')' ⊸ Dyck ⊸ Dyck)
```

$$\text{"()()"} \vdash \mathit{bal}\ \mathit{l1}\ \mathit{nil}\ \mathit{r1}\ (\mathit{bal}\ \mathit{l2}\ \mathit{nil}\ \mathit{r2}) : \mathit{Dyck}$$

Parser for the Dyck Grammar

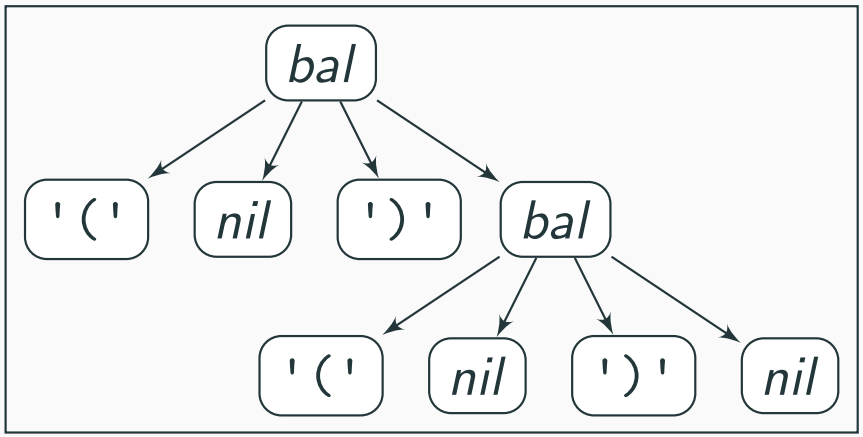
```
data Dyck : LinTy where
  nil : ↑ Dyck
  bal : ↑('(' ⊸ Dyck ⊸ ')' ⊸ Dyck ⊸ Dyck)
```

$$\text{"()()"} \vdash \mathit{bal}\ \mathit{l1}\ \mathit{nil}\ \mathit{r1}\ (\mathit{bal}\ \mathit{l2}\ \mathit{nil}\ \mathit{r2}) : \mathit{Dyck}$$

Parser for the Dyck Grammar

```
data Dyck : LinTy where
  nil : ↑ Dyck
  bal : ↑('(' ⊸ Dyck ⊸ ')' ⊸ Dyck ⊸ Dyck)
```

"()()" ⊢ *bal l*1 *nil r*1 (*bal l*2 *nil r*2) : *Dyck*



A Parser for the Dyck Grammar

Dyck Parser

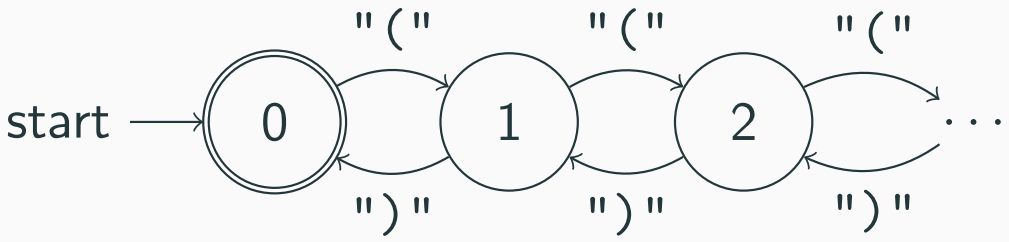
A parser for *Dyck* is a function

$$\uparrow (\mathit{String} \multimap \mathit{Dyck} \oplus \mathit{Dyck}_{\neg})$$

where $\mathit{Dyck} \,\&\, \mathit{Dyck}_{\neg} \cong 0$

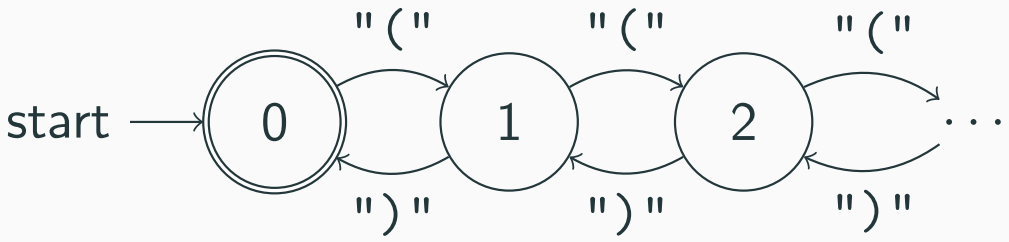
Strategy: use an intermediate automaton formalism

Dyck Traces



Instance of a general notion of deterministic automaton trace we define in the implementation.

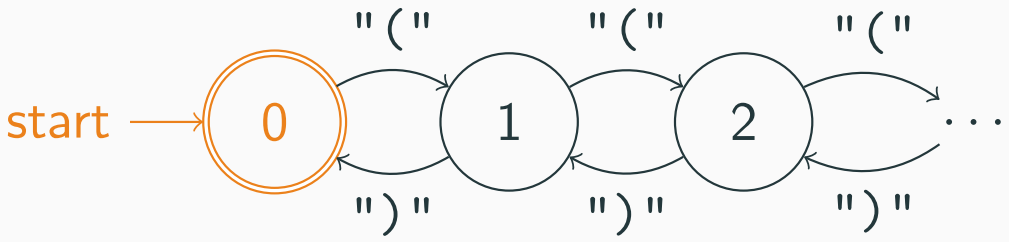
Dyck Traces




```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Instance of a general notion of deterministic automaton trace we define in the implementation.

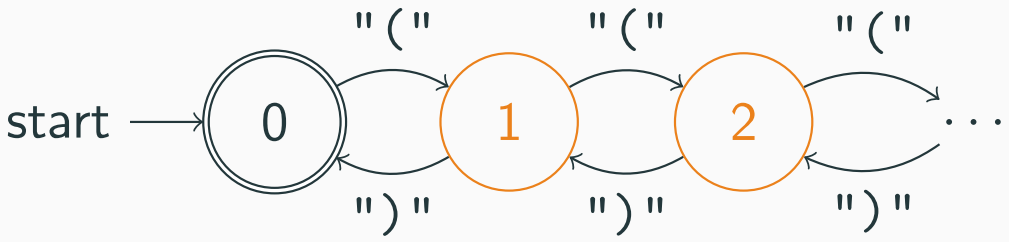
Dyck Traces




```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Instance of a general notion of deterministic automaton trace we define in the implementation.

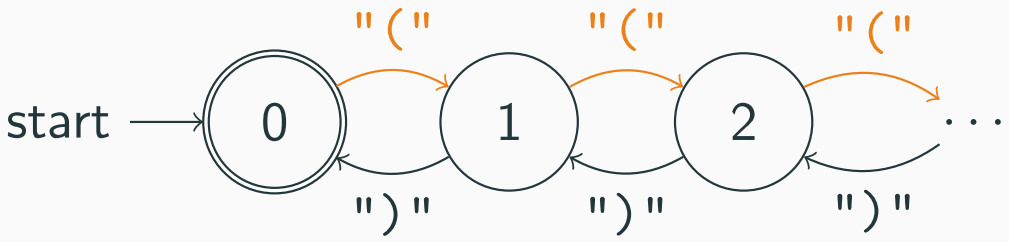
Dyck Traces




```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Instance of a general notion of deterministic automaton trace we define in the implementation.

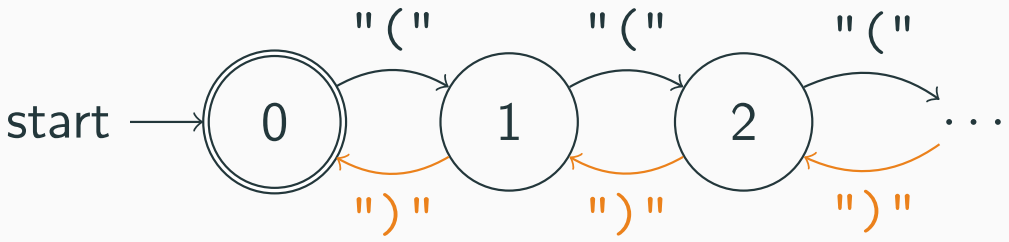
Dyck Traces




```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Instance of a general notion of deterministic automaton trace we define in the implementation.

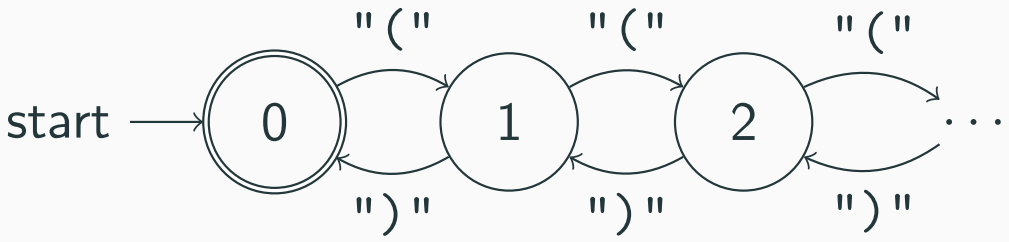
Dyck Traces




```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Instance of a general notion of deterministic automaton trace we define in the implementation.

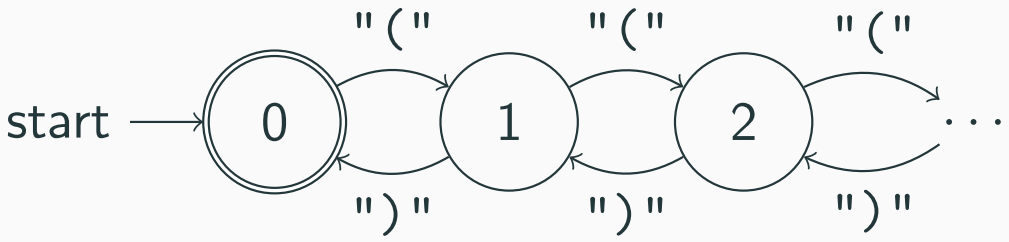
Dyck Traces




```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Instance of a general notion of deterministic automaton trace we define in the implementation.

Dyck Traces




```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Instance of a general notion of deterministic automaton trace we define in the implementation.

A Parser for the Dyck Grammar

Every String can be parsed into a unique trace

$$\top \cong \texttt{String} \cong \texttt{Trace true } 0 \oplus \texttt{Trace false } 0$$

Corollary: $\texttt{(Trace true } 0\texttt{)\&(Trace false } 0\texttt{)} \multimap 0$

A Parser for the Dyck Grammar

Every String can be parsed into a unique trace

$$\top \cong \texttt{String} \cong \texttt{Trace true } 0 \oplus \texttt{Trace false } 0$$

Corollary: $\texttt{(Trace true } 0\texttt{)\&(Trace false } 0\texttt{)} \multimap 0$

The Dyck Grammar is Strongly Equivalent to the Automaton

$$\mathit{Dyck} \cong \mathit{Trace}\ \mathit{true}\ 0$$

Strong equivalence is true here but not necessary for the parser. For a partial parser only need $\texttt{Trace true } 0 \multimap \texttt{Dyck}$. For completeness only need weak equivalence.

A Parser for the Dyck Grammar

Every String can be parsed into a unique trace

$$\top \cong \texttt{String} \cong \texttt{Trace true } 0 \oplus \texttt{Trace false } 0$$

Corollary: $\texttt{(Trace true } 0\texttt{)\&(Trace false } 0\texttt{)} \multimap 0$

The Dyck Grammar is Strongly Equivalent to the Automaton

$$\mathit{Dyck} \cong \mathit{Trace}\ \mathit{true}\ 0$$

Strong equivalence is true here but not necessary for the parser. For a partial parser only need $\texttt{Trace true } 0 \multimap \texttt{Dyck}$. For completeness only need weak equivalence.

$$\begin{aligned} \top \multimap \texttt{String} \multimap\ & \mathit{Trace}\ \mathit{true}\ 0 \oplus \mathit{Trace}\ \mathit{false}\ 0 \\ \multimap\ & \mathit{Dyck} \oplus \mathit{Trace}\ \mathit{false}\ 0 \end{aligned}$$

A Parser for the Dyck Grammar

Every String can be parsed into a unique trace

$$\top \cong \texttt{String} \cong \texttt{Trace true } 0 \oplus \texttt{Trace false } 0$$

Corollary: $\texttt{(Trace true } 0\texttt{)\&(Trace false } 0\texttt{)} \multimap 0$

The Dyck Grammar is Strongly Equivalent to the Automaton

$$\mathit{Dyck} \cong \mathit{Trace}\ \mathit{true}\ 0$$

Strong equivalence is true here but not necessary for the parser. For a partial parser only need $\texttt{Trace true } 0 \multimap \texttt{Dyck}$. For completeness only need weak equivalence.

$$\begin{aligned} \top &\multimap \texttt{String} \multimap \mathit{Trace}\ \mathit{true}\ 0 \oplus \mathit{Trace}\ \mathit{false}\ 0 \\ &\multimap \mathit{Dyck} \oplus \mathit{Trace}\ \mathit{false}\ 0 \end{aligned}$$

A Parser for Dyck Traces

Looks like an ordinary functional program, but syntactic discipline ensures soundness.

```
parse :
  ↑(String ⊸ &[ n ∈ ℕ ] ⊕[ b ∈ Bool ] Trace b n)
parse nil zero = σ true eof
parse nil (suc n) = σ false leftovers
parse (cons (σ '(' a) w) n =
  let σ b tr = parse w (suc n) in
  σ b (push a tr)
parse (cons (σ ')' a) w) zero =
  σ false (unexpected a _)
parse (cons (σ ')' a) w) (suc n) =
  let σ b tr = parse w n in
  σ b (pop a tr)
```

Functions from `Dyck` to `Trace` and vice-versa are similarly just functional programs between trees and lists but satisfying ordered linear discipline.

1. Overview of Dependent Lambek Calculus

2. Formal Grammar Theory and Parsing in Lambek^D

3. Semantics and Implementation

4. Future Work

Semantics: What is a formal grammar?

Fix a finite alphabet Σ.

- A formal language is a subset of strings over Σ. Equivalently, a function $\texttt{String} \to \texttt{Prop}$.

Formal languages provide specifications for recognizers, i.e., is the input string in a given language. Not sufficient for parsers, where we care about the reason that the string is in the language, i.e., the parse tree.

Various formalisms (Chomskyan generative grammars, Lambek's categorial grammars) define grammars and inductively generate their parse trees. Our approach (used by e.g., Conal Elliott):

- A formal grammar is a function $\texttt{String} \to \texttt{Set}$

A formal grammar is a "proof-relevant" formal language, it maps a string to the set of "proofs" that the string is in the language. A syntax-independent definition of grammar that isn't tied to a particular formalism.

Parse Transformers

Given formal grammars A, B, a parse transformer is a function $\prod_{w:\mathtt{String}} Aw \to Bw$, i.e., a function from A parses to B parses that respects the grammatical structure.

This is the category of families indexed by strings $\mathtt{Set}^{\mathtt{String}}$, which is very well-behaved: bi-complete, with a biclosed monoidal structure. Exactly what we need to interpret ordered linear type theory.

Denotational Semantics

Definition (Grammar Semantics)

We define the following interpretations by mutual recursion on the judgments of Lambek^D :

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

And this interpretation validates the equational theory.

Denotational Semantics

$$[\![c]\!]\gamma\, w = \{c \mid w = c\}$$

$$[\![I]\!]\gamma\, w = \{() \mid w = \varepsilon\}$$

$$[\![A \otimes B]\!]\gamma\, w = \{(w_1, w_2, a, b) \mid w_1 w_2 = w \wedge a \in [\![A]\!]\gamma\, w_1 \wedge b \in [\![B]\!]\gamma\, w_2\}$$

$$[\![A \multimap B]\!]\gamma\, w = \prod_{w'} [\![A]\!]\gamma\, w' \to [\![B]\!]\gamma\, ww'$$

$$[\![B \multimap A]\!]\gamma\, w = \prod_{w'} [\![A]\!]\gamma\, w' \to [\![B]\!]\gamma\, w'w$$

$$[\![\bigoplus_{x:X} A]\!]\gamma\, w = \{(x, a) \mid x \in [\![X]\!]\gamma \wedge a \in [\![A]\!](\gamma, x)\, w\}$$

$$[\![\mathop{\&}_{x:X} A]\!]\gamma\, w = \prod_{x \in [\![X]\!]\gamma} [\![A]\!](\gamma, x)\, w$$

$$[\![\uparrow A]\!]\gamma = [\![A]\!]\gamma\, \varepsilon$$

Implementation

We implement Lambek^D in cubical Agda by a shallow embedding based on the semantics:

- Non-linear types just implemented as Agda `Type`
- Linear types are implemented as `String` $\to$ `Type`

Benefit: don't need to reimplement dependent type theory(!), can re-use library functions about graphs, results about monoidal categories

Drawback: the programs aren't written as Lambek^D lambda terms, but instead hand-compiled to combinators analogous to the denotational semantics Example:

```
h : ↑((A ⊗ A)* ⊸ A*)
h nil = nil
h (cons (a1 , a2) as) = cons a1 (cons a2 (h as))
```

as combinators becomes h = `fold nil (cons` $\circ$ `id` $\otimes$ `cons` $\circ$ `assoc`$^{-1}$`)`

1. Overview of Dependent Lambek Calculus

2. Formal Grammar Theory and Parsing in Lambek^D

3. Semantics and Implementation

4. Future Work

Other Parsing Algorithms

- WIP on parser for arbitrary LL(1) grammars.

Other Parsing Algorithms

- WIP on parser for arbitrary LL(1) grammars.
- Derivative-based techniques: The left and right Brzozowski derivatives of grammar A by a character c are definable in Lambek^D :

$$\texttt{'c'} \multimap A \qquad A \circ\!\!- \texttt{'c'}$$

  Unclear if the usual rules of derivatives are derivable without additional axioms

Other Parsing Algorithms

- WIP on parser for arbitrary LL(1) grammars.
- Derivative-based techniques: The left and right Brzozowski derivatives of grammar A by a character c are definable in Lambek^D :

  $$\texttt{'c'} \multimap A \qquad A \multimap \texttt{'c'}$$

  Unclear if the usual rules of derivatives are derivable without additional axioms
- Context-sensitivity using dependency. For variable binding can define an indexed inductive grammar $\mathrm{Term}\, E$ where E : **List Ident** and type the λ constructor as follows:

  $$\&_{x:\mathbf{Ident}}\ \texttt{'fun'} \multimap \mathbf{Single}(x) \multimap \texttt{'.'} \multimap \mathrm{Term}\,(x :: E) \multimap \mathrm{Term}\, E$$

Type Systems as Types?

Type systems can be viewed as a formal grammar over abstract syntax trees. Type checking/inference is the analogue of the parser.

- Can a tree version of Lambek^D help us to write verified type checkers/inference/elaborators?
- What does a tree version of Lambek^D even look like? Instead of a single $\otimes$ representing concatenation, we have a tensor-like operation for each untyped term constructor.
- Can we do this over trees that incorporate binding structure (ABTs)?

Dependent Lambek Calculus

Grammars	**Linear Types**
Grammar A	Linear type A
Parse of string w	$w \vdash M : A$
Parser	$\top \vdash M : A \oplus A_{\neg}$
Parse transformer	$\Delta \vdash M : A$

Sound-by-construction parsers using dependent ordered linear typing.

Language and examples implemented in Cubical Agda

- github.com/maxsnew/grammars-and-semantic-actions
- Dockerized version: https://zenodo.org/records/15049780

Upcoming paper at PLDI 2025

- ArXiv preprint: arxiv.org/abs/2504.03995