Intrinsic Verification of Parsers in Dependent Lambek Calculus

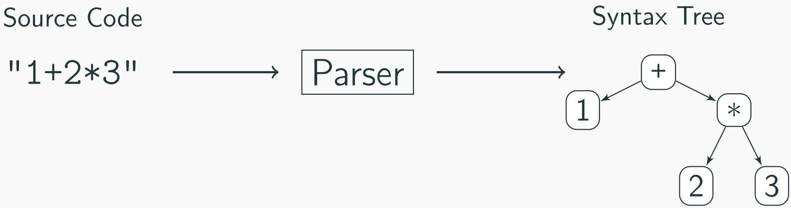
Steven Schaefer [1]   Nathan Varner [1]   Pedro H. Azevedo de Amorim [2]
Max S. New [1]

November 22, 2024

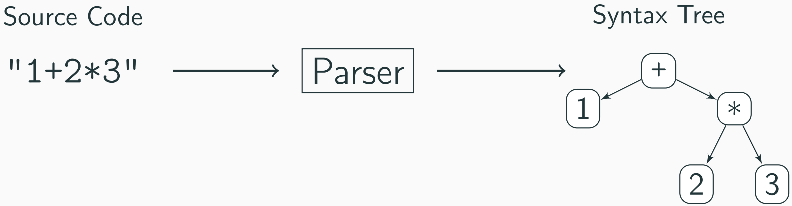
[1]University of Michigan

[2]University of Oxford

Parsing is Everywhere!

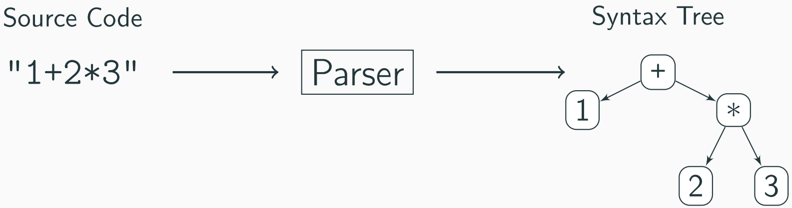


Parsing is Everywhere!



- Incorrect parsers jeopardize verified developments

Parsing is Everywhere!



- Incorrect parsers jeopardize verified developments
- Early versions of CompCert
    - 0 middle or backend bugs, 6 parser bugs (Yang et al., 2011)

Parser Correctness

A **parser** for grammar `A` is a partial function `f` : `String` $\rightharpoonup$ `Parse(A)`

Parser Correctness

A **parser** for grammar $\mathtt{A}$ is a partial function $\mathtt{f} : \mathtt{String} \rightharpoonup \mathtt{Parse(A)}$

$\mathtt{f}$ is correct if it is **sound** and **complete**

Parser Correctness

A **parser** for grammar `A` is a partial function $\texttt{f} : \texttt{String} \rightharpoonup \texttt{Parse(A)}$

`f` is correct if it is **sound** and **complete**

Soundness of `f`

`f(w)`, if it exists, is a parse tree of `w` for the grammar `A`

Parser Correctness

A **parser** for grammar `A` is a partial function `f : String` $\rightharpoonup$ `Parse(A)`

`f` is correct if it is **sound** and **complete**

Soundness of `f`

`f(w)`, if it exists, is a parse tree of `w` for the grammar `A`

Completeness of `f`

If a parse tree of `w` for `A` exists, then `f(w)` is defined

Parser Correctness

A **parser** for grammar `A` is a partial function `f : String ⇀ Parse(A)`

`f` is correct if it is **sound** and **complete**

Soundness of `f`

`f(w)`, if it exists, is a parse tree of `w` for the grammar `A`

Completeness of `f`

If a parse tree of `w` for `A` exists, then `f(w)` is defined

Our Approach

Define a domain-specific language in which all parsers are **sound-by-construction**

1. A DSL for Verified Parsing

2. An Example Parser

3. Implementation

4. Future Work

Domain-Specific Language for Verified Parsers

Dependent Lambek Calculus

- Non-commutative linear logic

Domain-Specific Language for Verified Parsers

Dependent Lambek Calculus

- Non-commutative linear logic

Grammars	**Linear Types**
Grammar `A`	Linear type `A`

Domain-Specific Language for Verified Parsers

Dependent Lambek Calculus

- Non-commutative linear logic

Grammars	**Linear Types**
Grammar `A`	Linear type `A`
Parse of string `w`	$\mathtt{w} \vdash \mathtt{A}$

Domain-Specific Language for Verified Parsers

Dependent Lambek Calculus

- Non-commutative linear logic

Grammars	**Linear Types**
Grammar $\mathtt{A}$	Linear type $\mathtt{A}$
Parse of string $\mathtt{w}$	$\mathtt{w} \vdash \mathtt{A}$
Parser	$\mathtt{String} \vdash \mathtt{A} \oplus \mathtt{A}_{\neg}$

Domain-Specific Language for Verified Parsers

Dependent Lambek Calculus

- Non-commutative linear logic

Grammars	**Linear Types**
Grammar $\mathtt{A}$	Linear type $\mathtt{A}$
Parse of string $\mathtt{w}$	$\mathtt{w} \vdash \mathtt{A}$
Parser	$\mathtt{String} \vdash \mathtt{A} \oplus \mathtt{A}_{\neg}$
Parse transformer	$\Delta \vdash \mathtt{A}$

Domain-Specific Language for Verified Parsers

Dependent Lambek Calculus

- Non-commutative linear logic

Grammars	**Linear Types**
Grammar $\mathtt{A}$	Linear type $\mathtt{A}$
Parse of string $\mathtt{w}$	$\mathtt{w} \vdash \mathtt{A}$
Parser	$\mathtt{String} \vdash \mathtt{A} \oplus \mathtt{A}_{\neg}$
Parse transformer	$\Delta \vdash \mathtt{A}$

Implemented in Agda and instantiated with verified parsers for regular expressions and selected context-free grammars

Why Linearity?

An ordered and linear type system

Why Linearity?

An ordered and linear type system

$$\boxed{\texttt{"cat"} \vdash \texttt{A}}$$

Why Linearity?

An ordered and linear type system

$$\boxed{\texttt{"cat"} \vdash \texttt{A}}$$

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

Why Linearity?

An <u>ordered</u> and <u>linear</u> type system

$$\boxed{\texttt{"cat"} \vdash \texttt{A}}$$

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

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

Why Linearity?

An ordered and linear type system

$$\boxed{\texttt{"cat"} \vdash \texttt{A}}$$

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

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

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

Why Linearity?

An <u>ordered</u> and <u>linear</u> type system

$$\boxed{\texttt{"cat"} \vdash \texttt{A}}$$

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

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

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

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

Why Linearity?

An <u>ordered</u> and <u>linear</u> type system

$$\boxed{\texttt{"cat"} \vdash \texttt{A}}$$

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

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

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

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

These restrictions ensure parsers are **sound-by-construction**

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string `I`

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string `I`
- Concatenation $\otimes$

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string `I`
- Concatenation $\otimes$
- Linear function types $\multimap$, $\circ\!\!-$

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string `I`
- Concatenation $\otimes$
- Linear function types $\multimap$, $\circ\!-$
- Disjunction ($\oplus$, 0), conjunction (&, $\top$)

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string $\texttt{I}$
- Concatenation $\otimes$
- Linear function types $\multimap$, $\multimapinv$
- Disjunction $(\oplus, 0)$, conjunction $(\&, \top)$
- Indexed disjunction $\bigoplus_{\texttt{x:X}} \texttt{A x}$, indexed conjunction $\&_{\texttt{x:X}} \texttt{A x}$ over a non-linear type $\texttt{X}$

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string `I`
- Concatenation $\otimes$
- Linear function types $\multimap$, $\multimapinv$
- Disjunction $(\oplus, 0)$, conjunction $(\&, \top)$
- Indexed disjunction $\bigoplus_{x:X} A\ x$, indexed conjunction $\bigwith_{x:X} A\ x$ over a non-linear type `X`
- Indexed inductive linear types

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string `I`
- Concatenation $\otimes$
- Linear function types $\multimap$, $\circ\!-$
- Disjunction $(\oplus, 0)$, conjunction $(\&, \top)$
- Indexed disjunction $\bigoplus_{x:X} A\ x$, indexed conjunction $\&_{x:X} A\ x$ over a non-linear type X
- Indexed inductive linear types

Non-linear Types

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string $\mathtt{I}$
- Concatenation $\otimes$
- Linear function types $\multimap$, $\circ\!\!-$
- Disjunction $(\oplus, 0)$, conjunction $(\&, \top)$
- Indexed disjunction $\bigoplus_{\mathtt{x:X}} \mathtt{A\ x}$, indexed conjunction $\mathop{\&}_{\mathtt{x:X}} \mathtt{A\ x}$ over a non-linear type $\mathtt{X}$
- Indexed inductive linear types

Non-linear Types

- Ordinary dependent types

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string `I`
- Concatenation $\otimes$
- Linear function types $\multimap$, $\circ\!-$
- Disjunction ($\oplus$, 0), conjunction ($\&$, $\top$)
- Indexed disjunction $\bigoplus_{\texttt{x:X}} \texttt{A x}$, indexed conjunction $\&_{\texttt{x:X}} \texttt{A x}$ over a non-linear type `X`
- Indexed inductive linear types

Non-linear Types

- Ordinary dependent types
- $\uparrow$: $\texttt{LinTy} \to \texttt{NonLinTy}$

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string `I`
- Concatenation $\otimes$
- Linear function types $\multimap$, $\circ\!-$
- Disjunction $(\oplus, 0)$, conjunction $(\&, \top)$
- Indexed disjunction $\bigoplus_{\mathtt{x:X}} \mathtt{A\ x}$, indexed conjunction $\bigwith_{\mathtt{x:X}} \mathtt{A\ x}$ over a non-linear type `X`
- Indexed inductive linear types

Non-linear Types

- Ordinary dependent types
- $\uparrow$: `LinTy` $\to$ `NonLinTy`
  - $\uparrow$ `A` is the type of `A`-parses for the empty string

Types in Dependent Lambek Calculus

Linear Types

- Characters in the alphabet — i.e. `'a'`
- Empty string `I`
- Concatenation $\otimes$
- Linear function types $\multimap$, $\multimapinv$
- Disjunction $(\oplus, 0)$, conjunction $(\&, \top)$
- Indexed disjunction $\bigoplus_{x:X} A\ x$, indexed conjunction $\&_{x:X} A\ x$ over a non-linear type `X`
- Indexed inductive linear types

Non-linear Types

- Ordinary dependent types
- $\uparrow$: `LinTy` $\rightarrow$ `NonLinTy`
  - $\uparrow$ `A` is the type of `A`-parses for the empty string
  - `!` in linear logic, $\square$ in separation logic

1. A DSL for Verified Parsing

2. An Example Parser

3. Implementation

4. Future Work

Dyck Grammar

- Context-free grammar of balanced parentheses

Dyck Grammar

- Context-free grammar of balanced parentheses
- Define as an inductive linear type

Dyck Grammar

- Context-free grammar of balanced parentheses
- Define as an inductive linear type

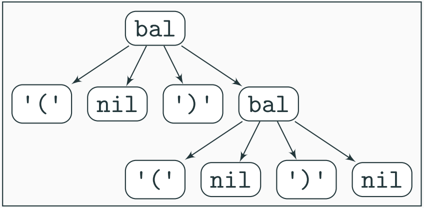
```
data Dyck : LinTy where
  nil : ↑ Dyck
  bal : ↑('(' ⊸ Dyck ⊸ ')' ⊸ Dyck ⊸ Dyck)
```

Dyck Grammar

- Context-free grammar of balanced parentheses
- Define as an inductive linear type

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

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



A Parser for the Dyck Grammar

Dyck Parser

A parser for $\mathtt{Dyck}$ is a function

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

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

A Parser for the Dyck Grammar

Dyck Parser

A parser for $\mathtt{Dyck}$ is a function

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

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

Soundness: guaranteed by the linear type system

A Parser for the Dyck Grammar

Dyck Parser

A parser for $\mathtt{Dyck}$ is a function

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

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

Soundness: guaranteed by the linear type system

Completeness: guaranteed by the disjointness of $\mathtt{Dyck}$ and $\mathtt{Dyck}_{\neg}$

A Parser for the Dyck Grammar

Dyck Parser

A parser for $\mathtt{Dyck}$ is a function

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

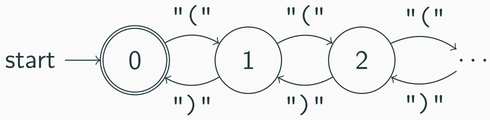
where $\mathtt{Dyck} \,\&\, \mathtt{Dyck}_{\neg} \cong 0$

Soundness: guaranteed by the linear type system

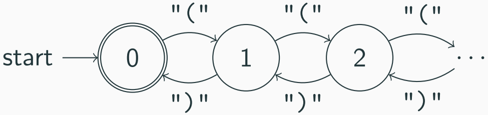
Completeness: guaranteed by the disjointness of $\mathtt{Dyck}$ and $\mathtt{Dyck}_{\neg}$

Need an automaton to define the parser!

Dyck Traces

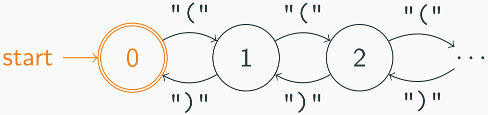


Dyck Traces



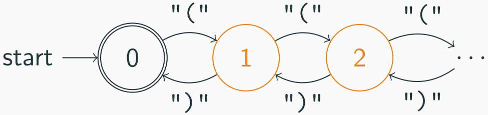

```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Dyck Traces



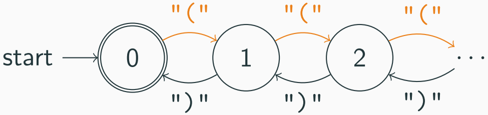

```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Dyck Traces



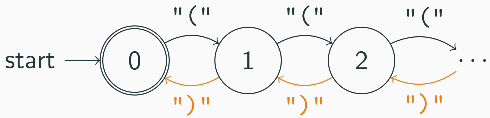

```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Dyck Traces



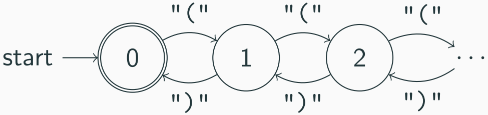

```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

Dyck Traces



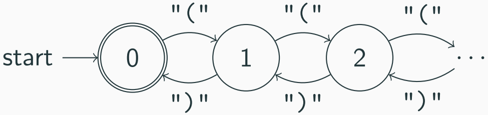

```
data Trace : Bool → ℕ → LinTy where
 eof : ↑ (Trace true 0)
 leftovers : ∀ n → ↑ (Trace false (suc n))
 push : ∀ b n → ↑('(' ⊸ Trace b (suc n) ⊸ Trace b n)
 pop : ∀ b n → ↑(')' ⊸ Trace b n ⊸ Trace b (suc n))
 unexpected : ↑(')' ⊸ ⊤ ⊸ Trace false 0)
```

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)
```

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)
```

A Parser for the Dyck Grammar

Deterministic Automaton

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

A Parser for the Dyck Grammar

Deterministic Automaton

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

The Dyck Grammar is Recognized by the Automaton

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

A Parser for the Dyck Grammar

Deterministic Automaton

$$\mathtt{String} \cong \mathtt{Trace}\ \mathtt{true}\ 0 \oplus \mathtt{Trace}\ \mathtt{false}\ 0$$

The Dyck Grammar is Recognized by the Automaton

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

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

A Parser for the Dyck Grammar

Deterministic Automaton

$$\mathtt{String} \cong \mathtt{Trace}\ \mathtt{true}\ 0 \oplus \mathtt{Trace}\ \mathtt{false}\ 0$$

The Dyck Grammar is Recognized by the Automaton

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

$$\begin{aligned} \mathtt{String} &\multimap \mathtt{Trace}\ \mathtt{true}\ 0 \oplus \mathtt{Trace}\ \mathtt{false}\ 0 \\ &\multimap \mathtt{Dyck} \oplus \mathtt{Trace}\ \mathtt{false}\ 0 \qquad \text{A parser!} \end{aligned}$$

A Parser for Dyck Traces

Looks like an ordinary functional program

```
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)
```

Semantics and Implementation

Denotational semantics

$$[\![\texttt{A}]\!] : \texttt{String} \rightarrow \texttt{Set} \qquad [\![\texttt{A}]\!]\ \texttt{w} = \{\text{parse trees of } \texttt{w} \text{ for } \texttt{A}\}$$

Semantics and Implementation

Denotational semantics

$$[\![\mathtt{A}]\!] : \mathtt{String} \to \mathtt{Set} \qquad [\![\mathtt{A}]\!]\ \mathtt{w} = \{\text{parse trees of } \mathtt{w} \text{ for } \mathtt{A}\}$$

Implementation of Dependent Lambek Calculus shallowly embedded in Cubical Agda

- $\mathtt{LinTy} := \mathtt{String} \to \mathtt{Set}$
- **Pros:** reuses the Agda typechecker
- **Cons:** poor performance and enforced combinatory style

Parsers Implemented in Agda

Dyck grammar (LL(0))

Parsers Implemented in Agda

Dyck grammar (LL(0))

Arithmetic expressions with a binary operation (LL(1))

Parsers Implemented in Agda

Dyck grammar (LL(0))

Arithmetic expressions with a binary operation (LL(1))

Regular expressions

- Equivalences between NFAs, DFAs, and regexes
- Thompson's construction, powerset construction

1. A DSL for Verified Parsing

2. An Example Parser

3. Implementation

4. Future Work

Future Work

Parsing

- LL/LR parser generator

Future Work

Parsing

- LL/LR parser generator
- Semantic actions

Future Work

Parsing

- LL/LR parser generator
- Semantic actions

Implementation

Future Work

Parsing

- LL/LR parser generator
- Semantic actions

Implementation

- Performance, ergonomics

Future Work

Parsing

- LL/LR parser generator
- Semantic actions

Implementation

- Performance, ergonomics

Typechecking?

Dependent Lambek Calculus

Grammars	**Linear Types**
Grammar $\mathtt{A}$	Linear type $\mathtt{A}$
Parse of string $\mathtt{w}$	$\mathtt{w} \vdash \mathtt{A}$
Parser	$\mathtt{String} \vdash \mathtt{A} \oplus \mathtt{A}_{\neg}$
Parse transformer	$\Delta \vdash \mathtt{A}$

Grammars are types! Automata are types!

Language and examples implemented in Cubical Agda

- github.com/maxsnew/grammars-and-semantic-actions

Preprint of this work

- maxsnew.com/docs/lambek.pdf