Context-free Languages, Type Theoretically

Jaro Reinders[1][0000−0002−6837−3757] and Casper Bach[2][0000−0003−0622−7639]

[1] Delft University of Technology, Delft, The Netherlands
[2] University of Southern Denmark, Odense, Denmark

Abstract. Parsing is the process of recovering structure from strings, an essential part of implementing programming languages. Previous work has shown that formalizing languages and parsers using an idiomatic type theoretic approach can be simple and enlightening. Unfortunately, this approach has only been applied to regular languages, which are not expressive enough for many practical applications. We are working on extending the type theoretic formalization to context-free languages which substantially more expressive. We hope our formalization can serve as a foundation for reasoning about new disambiguation techniques and even more expressive formalisms such as data-dependent grammars.

Keywords: Language · Parsing · Type Theory

1 Introduction

Parsing is the conversion of flat, human-readable text into a tree structure that is easier for computers to manipulate. As one of the central pillars of compiler tooling since the 1960s, today almost every automated transformation of computer programs requires a form of parsing. Though it is a mature research subject, it is still actively studied, for example the question of how to resolve ambiguities in context-free grammars [1].

Most parsing works mix the essence of the parsing technique with operational details . Our understanding and ability to improve upon these parsing techniques is hindered by the additional complexity of these inessential practical concerns. To address this issue, we are developing natural denotational semantics for traditional parsing techniques.

[JR] such as... state machines, continuations, memoization?

[JR] Elliot has kicked off this effort...

Recent work by Elliot uses interactive theorem provers to state simple specifications of languages and that proofs of desirable properties of these language specifications transfer easily to their parsers [3]. Unfortunately, this work only considers regular languages which are not powerful enough to describe practical programming languages.

[JR] Make the problem clear through an example: if we have a left-recursive grammar then naively unfolding it gets us into an infinite loop.

In this paper, we formalize context-free languages and show how to parse them, extending Elliot's type theoretic approach to language specfication. One of the main challenges is that the recursive nature of context-free languages does not map directly onto interactive theorem provers as they do not support general recursion (for good reasons). We encode context-free languages as fixed points of functors (initial algebras).

[JR] Say something about the limitation that we only study acyclic grammars: there must be a total order on nonterminals and a nonterminal is not allowed to refer to nonterminals that come before it. We wanted to start by limiting ourselves to grammars with only one nonterminal, but those are not closed under derivatives.

We make the following concrete contributions:

– We extend Elliot's type theoretic formalization of regular languages to context-free languages.

For this paper we have chosen Agda as our type theory and interactive theorem prover. We believe our definitions should transfer easily to other theories and tools. This paper itself is a literate Agda file; all highlighted Agda code has been accepted by Agda's type checker, giving us a high confidence of correctness.

2 Languages and Parsers

In this section, we introduce background information, namely how we define languages, basic language combinators, and parsers. Our exposition follows Elliot [3]. In Section 3, we extend these concepts to context free languages.

2.1 Languages

We define languages as being functions from strings to types.[3]

```
Lang = String → Type
```

The result type can be thought of as the type of proofs that the string is in the language.

Remark 1. Note that a language may admit multiple different proofs for the same string. That is an important difference between the type theoretic approach and the more common set theoretic approach, which models languages as sets of strings.

This is a broad definition of what a language is; it includes languages that are outside the class of context-free languages.

Example 1. The language $a^n b^n c^n$ can be specified as follows:

```
abc : Lang
abc w = Σ[ n ∈ ℕ ] w ≡ repeat n 'a' ++ repeat n 'b' ++ repeat n 'c'
```

We can show that the string *aabbcc* is in this language by choosing n to be 2, from which the required equality follows by reflexivity after normalization:

```
aabbcc : abc "aabbcc"
aabbcc = 2 , refl
```

Example 1 shows that it is possible to specify languages and prove that certain strings are in those languages, but for practical applications we do not want to be burdened with writing such proofs ourselves. The compiler should be able to decide whether or not your program is valid by itself.

[JR] do I need to give an example?

- Agda is too powerful: it can specify undecidable languages
- So, we need to define a simpler language which still supports all the features we need.

2.2 Basic Language Combinators

Let's start with a simple example: POSIX file system permissions. These are usually summarized using the characters 'r', 'w', and 'x' if the permissions are granted, or '-' in place of the corresponding character if the permission is denied. For example the string "r-x" indicates that read and execute permissions are granted, but the write permission is denied. The full language can be expressed using the following BNF grammar:

[JR] cite: BNF

$$\langle permissions \rangle ::= \langle read \rangle\ \langle write \rangle\ \langle execute \rangle$$

$$\langle read \rangle ::= \text{‘-’} \mid \text{‘r’}$$

$$\langle write \rangle ::= \text{‘-’} \mid \text{‘w’}$$

$$\langle execute \rangle ::= \text{‘-’} \mid \text{‘x’}$$

This grammar uses three important features: sequencing, choice, and matching character literals. We can define these features are combinators in Agda as shown in Figure 1 and use them to write our permissions grammar as follows:

```
permissions = read * write * execute
read        = ‘ '-' ∪ ‘ 'r'
write       = ‘ '-' ∪ ‘ 'w'
execute     = ‘ '-' ∪ ‘ 'x'
```

[3] We use Type as a synonym for Agda's Set to avoid confusion.

```
‘_ : Char → Lang                                  ∅ : Lang
(‘ c) w = w ≡ c ∷ []                              ∅ _ = ⊥

_∪_ : Lang → Lang → Lang                          ϵ : Lang
(P ∪ Q) w = P w ⊎ Q w                             ϵ w = w ≡ []

_∗_ : Lang → Lang → Lang                          _·_ : Type → Lang → Lang
(P ∗ Q) w = ∃[ u ] ∃[ v ] w ≡ u ++ v × P u × Q v  (A · P) w = A × P w
```

Fig. 1. Basic language combinators.

2.3 Parsers

We want to write a program which can prove for us that a given string is in the language. What should this program return for strings that are not in the language? We want to make sure our program does find a proof if it exists, so if it does not exist then we want a proof that the string is not in the language. We can capture this using a type called Dec from the Agda standard library. It can be defined as follows:

```
data Dec (A : Type) : Type where
  yes : A → Dec A
  no : ¬ A → Dec A
```

A parser for a language, then, is a program which can tell us whether any given string is in the language or not.

```
Parser : Lang → Set
Parser P = (w : String) → Dec (P w)
```

Remark 2. Readers familiar with Haskell might see similarity between this type and the type `String -> Maybe a`, which is one way to implement parser combinators (although usually the return type is `Maybe (a, String)` giving parsers the freedom to consume only a prefix of the input string and return the rest). The differences are that the result of our Parser type depends on the language specification and input string, and that a failure carries with it a proof that the string cannot be part of the language. This allows us to separate the specification of our language from the implementation while ensuring correctness.

Remark 3. Note that the Dec type only requires our parsers to produce a single result; it does not have to exhaustively list all possible ways to parse the input string. In Haskell, one might write `String -> [(a, String)]`, which allows a parser to return multiple results but still does not enforce exhaustiveness. Instead, we could use:

[JR] cite: monadic parser combinators

[JR] This should be explained in more detail

– completely unique account of enumeration.
– bijection with Fin n for some n or Nat.

In this paper, however, we use Dec to keep the presentation simple.

To construct a parser for our permissions language, we start by defining parsers for each of the language combinators. Let us start by considering the character combinator. If the given string is empty or has more than one character, it can never be in a language formed by one character. If the string does consist of only one character, then it is in the language if that character is the same as from the language specification. In Agda, we can write such a parser for characters as follows:

```
‘-parse_ : (x : Char) → Parser (‘ x)
(‘-parse _) [] = no λ ()
(‘-parse x) (c ∷ []) = Dec.map (mk⇔ (λ { refl → refl }) (λ { refl → refl })) (c ≟ x)
(‘-parse _) (_ ∷ _ ∷ _) = no λ ()
```

This is a correct implementation of a parser for languages that consist of a single character, but the implementation is hard to read and does not give much insight. Instead, we can factor this parser into two cases: the empty string case and the case where the string has at least one character. We call the former nullability and use the greek character ν to signify it, and we call the latter derivative and use the greek character δ to signify it. Figure 2 shows how these cases can be defined and how they relate to the basic combinators. These properties motivate the introduction of three new basic combinators: guards _ · _, the language consisting of only the empty string ϵ, and the empty language $\emptyset$.

[JR] This does not motivate the split into ν and δ well enough. Also, the new combinators can be motivated more clearly.

$$\nu\ P = P\ [] \qquad\qquad (\delta\ c\ P)\ w = P\ (c :: w)$$

$$A \Leftrightarrow B = (A \to B) \times (B \to A) \qquad\qquad P \Longleftrightarrow Q = \forall\ \{w\} \to P\ w \Leftrightarrow Q\ w$$

$$\begin{array}{llll}
\nu\emptyset & : \bot & \Leftrightarrow \nu\ \emptyset \\
\nu\epsilon & : \top & \Leftrightarrow \nu\ \epsilon \\
\nu\cdot & : (A \times \nu\ P) & \Leftrightarrow \nu\ (A\ \cdot\ P) \\
\nu‘ & : \bot & \Leftrightarrow \nu\ (‘\ c’) \\
\nu\cup & : (\nu\ P \uplus \nu\ Q) & \Leftrightarrow \nu\ (P \cup Q) \\
\nu* & : (\nu\ P \times \nu\ Q) & \Leftrightarrow \nu\ (P * Q)
\end{array}
\qquad
\begin{array}{lll}
\delta\emptyset & : \emptyset & \Longleftrightarrow \delta\ c\ \emptyset \\
\delta\epsilon & : \emptyset & \Longleftrightarrow \delta\ c\ \epsilon \\
\delta\cdot & : (A\ \cdot\ \delta\ c\ P) & \Longleftrightarrow \delta\ c\ (A\ \cdot\ P) \\
\delta‘ & : ((c \equiv c’)\ \cdot\ \epsilon) & \Longleftrightarrow \delta\ c\ (‘\ c’) \\
\delta\cup & : (\delta\ c\ P \cup \delta\ c\ Q) & \Longleftrightarrow \delta\ c\ (P \cup Q) \\
\delta* & : (\nu\ P\ \cdot\ \delta\ c\ Q \cup \delta\ c\ P * Q) & \\
 & & \Longleftrightarrow \delta\ c\ (P * Q)
\end{array}$$

Fig. 2. Nullability, derivatives, and how they relate to the basic combinators.

Now the implementation of parsers for languages consisting of a single character follows completely from the decomposition into nullability and derivatives.

```
‘-parse_ : (c’ : Char) → Parser (‘ c’)
(‘-parse _) []        = Dec.map ν‘ ⊥-dec
(‘-parse c’) (c ∷ w) = Dec.map δ‘ (((c ≟ c’) ·-parse ϵ-parse) w)
```

The implementation of ·-parse, ϵ-parse, and $\emptyset$-parse are straightforward and can be found in our source code artifact.

[JR] todo: reference this nicely

```
_∪-parse_ : Parser P → Parser Q → Parser (P ∪ Q)
(ϕ ∪-parse ψ) []        = Dec.map ν∪ (ν ϕ ⊎-dec ν ψ)
(ϕ ∪-parse ψ) (c ∷ w) = Dec.map δ∪ ((δ c ϕ ∪-parse δ c ψ) w)

_∗-parse_ : Parser P → Parser Q → Parser (P ∗ Q)
(ϕ ∗-parse ψ) []        = Dec.map ν∗ (ν ϕ ×-dec ν ψ)
(ϕ ∗-parse ψ) (c ∷ w) = Dec.map δ∗ ((ν ϕ ·-parse δ c ψ ∪-parse δ c ϕ ∗-parse ψ) w)
```

Using these combinators we can define a parser for the permissions language by simply mapping each of the language combinators onto their respective parser combinators.

```
permissions-parse = read-parse ∗-parse (write-parse ∗-parse execute-parse)
read-parse        = (‘-parse '-') ∪-parse (‘-parse 'r')
write-parse       = (‘-parse '-') ∪-parse (‘-parse 'w')
execute-parse     = (‘-parse '-') ∪-parse (‘-parse 'x')
```

2.4 Infinite Languages

This permissions language is very simple. In particular, it is finite. In practice, many languages are inifinite, for which the basic combinators will not suffice. For example, file paths can be arbitrarily long on most systems. Elliot [3] defines a Kleene star combinator which enables him to specify regular languages such as file paths.

[JR] does this need citation?

However, we want to go one step further, speficying and parsing context-free languages. Most practical programming languages are at least context-free, if not more complicated. One essential feature of many languages is the ability to recognize balanced brackets. A minimal example language with balanced brackets is the following:

$$\langle \mathit{brackets} \rangle \quad ::= \epsilon \mid \text{‘}\texttt{[}\text{’} \ \langle \mathit{brackets} \rangle \ \text{‘}\texttt{]}\text{’} \mid \langle \mathit{brackets} \rangle \ \langle \mathit{brackets} \rangle$$

This is the language of all strings which consist of balanced square brackets. Many practical programming languages include some form of balanced brackets. Furthermore, this language is well known to be context-free and not regular. Thus, we need more powerful combinators.

We could try to naively transcribe the brackets grammar using our basic combinators, but Agda will justifiably complain that it is not terminating (here I've added a NON_TERMINATING pragma to make Agda to accept it any way).

```
{-# NON_TERMINATING #-}
brackets = ϵ ∪ ‘ '[' ∗ brackets ∗ ‘ ']' ∪ brackets ∗ brackets
```

We need to find a different way to encode this recursive relation.

```
postulate μ : (Lang → Lang) → Lang
bracketsμ = μ (λ P → ϵ ∪ ‘ '[' ∗ P ∗ ‘ ']' ∪ P ∗ P)
```

- μ cannot be implemented just like that
- we need to restrict the Lang → Lang function that we take a fixed point over
- polynomial functors would work, but our grammars are slightly different.
- Luckily, our basic combinators with variables added also works
- We can make this obvious to agda by defining a data type of descriptions a la gentle art of levitation.

3 Context-free Languages

3.1 Syntax

```
data Exp : Type₁ where
  ∅ : Exp
  ϵ : Exp
```

```
    ‘_ : (c : Char) → Exp
    _·_ : {a : Type} → Dec a → Exp → Exp
    _∪_ : Exp → Exp → Exp
    _*_ : Exp → Exp → Exp
    i : Exp
    μ : Exp → Exp -- explain later
```

Mapping syntax onto semantics:

```
⟦_⟧₁ : Exp → Lang → Lang
```

```
data ⟦_⟧ (e : Exp) : Lang where
  ∞ : ⟦ e ⟧₁ ⟦ e ⟧ w → ⟦ e ⟧ w
! : ⟦ e ⟧ w → ⟦ e ⟧₁ ⟦ e ⟧ w
! (∞ x) = x
```

```
⟦ ∅ ⟧₁ _ = ◇.∅
⟦ ε ⟧₁ _ = ◇.ε
⟦ ‘ c ⟧₁ _ = ◇.‘ c
⟦ x · e ⟧₁ l = x ◇.· ⟦ e ⟧₁ l
⟦ e ∪ e₁ ⟧₁ l = ⟦ e ⟧₁ l ◇.∪ ⟦ e₁ ⟧₁ l
⟦ e * e₁ ⟧₁ l = ⟦ e ⟧₁ l ◇.* ⟦ e₁ ⟧₁ l
⟦ i ⟧₁ l = l
⟦ μ e ⟧₁ _ = ⟦ e ⟧ -- explain this later
```

3.2 Goal

Our goal is to define:

```
parse : (e : Exp) (w : String) → Dec (⟦ e ⟧ w)
```

[JR] Now we should explain the $\diamond\nu$ and $\diamond\delta$

Our approach uses the decomposition of languages into ν and δ.

```
ν : (e : Exp) → Dec (.◇ν ⟦ e ⟧)
δ : Char → Exp → Exp
```

The ν function can easily be written to be correct by construction, however δ must be proven correct separately as follows:

```
δ-sound : ⟦ δ c e ⟧ w → .◇δ c ⟦ e ⟧ w
δ-complete : .◇δ c ⟦ e ⟧ w → ⟦ δ c e ⟧ w
```

The actual parsing follows the ν∘foldlδ decomposition.

```
parse e [] = ν e
parse e (c ∷ w) = map' δ-sound δ-complete (parse (δ c e) w)
```

That is the main result of this paper. The remainder of the paper concerns the implementation of ν, $af\delta$, δ-sound, and δ-commplete.

3.3 Nullability correctness

Lemma 1. *nullability of e substituted in e is the same as the nullability of e by itself*

```
νe∅→νee : (e : Exp) → .◇ν (⟦ e ⟧₁ ◇.∅) → .◇ν (⟦ e ⟧₁ ⟦ e₀ ⟧) -- more general than we need, but easy
νee→νe∅ : (e : Exp) → .◇ν (⟦ e ⟧₁ ⟦ e ⟧) → .◇ν (⟦ e ⟧₁ ◇.∅)
```

Syntactic nullability (correct by construction):

```
ν₁ : (e : Exp) → Dec (.◇ν (⟦ e ⟧₁ ◇.∅))
ν₁ ∅ = no λ ()
ν₁ ϵ = yes refl
ν₁ (‘ c) = no λ ()
ν₁ (x · e) = x ×-dec ν₁ e
ν₁ (e ∪ e₁) = ν₁ e ⊎-dec ν₁ e₁
ν₁ (e * e₁) = map’ (λ x → [] , [] , refl , x) (λ { ([] , [] , refl , x) → x }) (ν₁ e ×-dec ν₁ e₁)
ν₁ i = no λ ()
ν₁ (μ e) = map’ (∞ ∘ νe∅→νee e) (νee→νe∅ e ∘ !) (ν₁ e)
```

Using Lemma 1 we can define ν in terms of ν_1:

```
ν e = map’ (∞ ∘ νe∅→νee e) (νee→νe∅ e ∘ !) (ν₁ e)
```

[JR] TODO: show how ν works through examples

The forward direction is proven using straightforward induction.

```
νe∅→νee ϵ x = x
νe∅→νee (x₁ · e) (x , y) = x , νe∅→νee e y
νe∅→νee (e ∪ e₁) (inj₁ x) = inj₁ (νe∅→νee e x)
νe∅→νee (e ∪ e₁) (inj₂ y) = inj₂ (νe∅→νee e₁ y)
νe∅→νee (e * e₁) ([] , [] , refl , x , y) = [] , [] , refl , νe∅→νee e x , νe∅→νee e₁ y
νe∅→νee i ()
νe∅→νee (μ e) x = x
```

The backwards direction requires a bit more work. We use the following lemma:

Lemma 2. *If substituting e_0 into e is nullable then that must mean:*

1. *e by itself was already nullable, or*
2. *e_0 by itself is nullable*

Proof:

```
ν-split : (e : Exp) → .◇ν (⟦ e ⟧₁ ⟦ e₀ ⟧) → .◇ν (⟦ e ⟧₁ ◇.∅) ⊎ .◇ν (⟦ e₀ ⟧₁ ◇.∅)
ν-split ϵ x = inj₁ x
ν-split (_ · e) (x , y) = Sum.map₁ (x ,_) (ν-split e y)
ν-split (e ∪ e₁) (inj₁ x) = Sum.map₁ inj₁ (ν-split e x)
ν-split (e ∪ e₁) (inj₂ y) = Sum.map₁ inj₂ (ν-split e₁ y)
ν-split (e * e₁) ([] , [] , refl , x , y) = lift⊎₂ (λ x y → [] , [] , refl , x , y) (ν-split e x) (ν-split e₁ y)
ν-split {e₀ = e} i (∞ x) = inj₂ (reduce (ν-split e x))
ν-split (μ e) x = inj₁ x
```

The backwards direction of Lemma 1 is now simply a result of Lemma 2 where both sides of the disjoint union are equal and thus we can reduce it to a single value.

```
νee→νe∅ e x = reduce (ν-split {e₀ = e} e x)
```

3.4 Derivative correctness

[JR] At thi… need to int…

Internal/syntactic substitution:

$\mathsf{sub} : \mathsf{Exp} \to \mathsf{Exp} \to \mathsf{Exp}$
$\mathsf{sub}\ \_\ \emptyset = \emptyset$
$\mathsf{sub}\ \_\ \epsilon = \epsilon$
$\mathsf{sub}\ \_\ (`\ c) = `\ c$
$\mathsf{sub}\ e_0\ (x\ \cdot\ e) = x\ \cdot\ \mathsf{sub}\ e_0\ e$
$\mathsf{sub}\ e_0\ (e \cup e_1) = \mathsf{sub}\ e_0\ e \cup \mathsf{sub}\ e_0\ e_1$
$\mathsf{sub}\ e_0\ (e * e_1) = \mathsf{sub}\ e_0\ e * \mathsf{sub}\ e_0\ e_1$
$\mathsf{sub}\ e_0\ \mathsf{i} = e_0$
$\mathsf{sub}\ \_\ (\mu\ e) = \mu\ e$

We would like to be able to say $[\![\ \texttt{sub e}_0\ \texttt{e}\ ]\!] \equiv [\![\ \texttt{e}\ ]\!]_1\ [\![\ \texttt{e}_0\ ]\!]$\verb, but we can't because e_0's free variable would get (implicitly) captured. μ closes off an expression and thus prevents capture.

Lemma 3. *(Internal) syntactic substitution is the same as (external) semantic substitution. This is the raison d'être of μ.*

Proof:

$\textit{sub-sem}' : (e : \textit{Exp}) \to [\![\ \textit{sub}\ (\mu\ e_0)\ e\ ]\!]_1\ l \equiv [\![\ e\ ]\!]_1\ [\![\ e_0\ ]\!]$
$\textit{sub-sem}'\ \emptyset = \textit{refl}$
$\textit{sub-sem}'\ \epsilon = \textit{refl}$
$\textit{sub-sem}'\ (`\ \_) = \textit{refl}$
$\textit{sub-sem}'\ (x\ \cdot\ e) = \textit{cong}\ (x \diamond.\cdot\_)\ (\textit{sub-sem}'\ e)$
$\textit{sub-sem}'\ (e \cup e_1) = \textit{cong}_2\ \diamond.\_\cup\_\ (\textit{sub-sem}'\ e)\ (\textit{sub-sem}'\ e_1)$
$\textit{sub-sem}'\ (e * e_1) = \textit{cong}_2\ \diamond.\_*\_\ (\textit{sub-sem}'\ e)\ (\textit{sub-sem}'\ e_1)$
$\textit{sub-sem}'\ i = \textit{refl}$
$\textit{sub-sem}'\ (\mu\ \_) = \textit{refl}$

We only need to use this proof in its expanded form:

$\textit{sub-sem} : (e : \textit{Exp}) \to [\![\ \textit{sub}\ (\mu\ e_0)\ e\ ]\!]_1\ l\ w \equiv [\![\ e\ ]\!]_1\ [\![\ e_0\ ]\!]\ w$
$\textit{sub-sem}\ e = \textit{cong}\ (\lambda\ l \to l\ \_)\ (\textit{sub-sem}'\ e)$

This is the syntactic derivative (the e_0 argument stands for the whole expression):

$\delta_1 : (c : \mathsf{Char}) \to \mathsf{Exp} \to \mathsf{Exp} \to \mathsf{Exp}$
$\delta_1\ c\ \_\ \emptyset = \emptyset$
$\delta_1\ c\ \_\ \epsilon = \emptyset$
$\delta_1\ c\ \_\ (`\ c_1) = (c \stackrel{?}{=} c_1)\ \cdot\ \epsilon$
$\delta_1\ c\ e_0\ (x\ \cdot\ e) = x\ \cdot\ \delta_1\ c\ e_0\ e$
$\delta_1\ c\ e_0\ (e \cup e_1) = \delta_1\ c\ e_0\ e \cup \delta_1\ c\ e_0\ e_1$
$\delta_1\ c\ e_0\ (e * e_1) = (\delta_1\ c\ e_0\ e * \mathsf{sub}\ (\mu\ e_0)\ e_1) \cup (\mathsf{Dec.map}\ (\Leftrightarrow.\mathsf{trans}\ (\mathsf{mk}\Leftrightarrow\ !\ \infty)\ (\equiv\to\Leftrightarrow\ (\mathsf{sub\text{-}se}$
$\delta_1\ c\ e_0\ \mathsf{i} = \mathsf{i}$
$\delta_1\ c\ \_\ (\mu\ e) = \mu\ (\delta_1\ c\ e\ e)$

For a top-level expression the derivative is just the open δ_1 where e_0 is e itself:

$\delta\ c\ e = \delta_1\ c\ e\ e$

[JR] todo: show how δ works through examples.

The proofs are by induction and the Lemma 3:

```
δ-sound' : (e : Exp) → ⟦ δ1 c e0 e ⟧1 ⟦ δ c e0 ⟧ w → .◇δ c (⟦ e ⟧1 ⟦ e0 ⟧) w
δ-sound' (‘ _) (refl , refl) = refl
δ-sound' (x1 · e) (x , y) = x , δ-sound' e y
δ-sound' (e ∪ e1) (inj1 x) = inj1 (δ-sound' e x)
δ-sound' (e ∪ e1) (inj2 y) = inj2 (δ-sound' e1 y)
δ-sound' {c = c} (e * e1) (inj1 (u , v , refl , x , y)) = c ∷ u , v , refl , δ-sound' e x , transport (sub-sem e1) y
δ-sound' {c = c} {w = w} (e * e1) (inj2 (x , y)) = [] , c ∷ w , refl , x , δ-sound' e1 y
δ-sound' {e0 = e} i (∞ x) = ∞ (δ-sound' e x)
δ-sound' (μ e) (∞ x) = ∞ (δ-sound' e x)

δ-sound {e = e} (∞ x) = ∞ (δ-sound' e x)

δ-complete' : (e : Exp) → .◇δ c (⟦ e ⟧1 ⟦ e0 ⟧) w → ⟦ δ1 c e0 e ⟧1 ⟦ δ c e0 ⟧ w
δ-complete' (‘ _) refl = refl , refl
δ-complete' (_ · e) (x , y) = x , δ-complete' e y
δ-complete' (e ∪ e1) (inj1 x) = inj1 (δ-complete' e x)
δ-complete' (e ∪ e1) (inj2 y) = inj2 (δ-complete' e1 y)
δ-complete' (e * e1) (c ∷ u , v , refl , x , y) = inj1 (u , v , refl , δ-complete' e x , transport (sym (sub-sem e1)) y)
δ-complete' (e * e1) ([] , c ∷ w , refl , x , y) = inj2 (x , δ-complete' e1 y)
δ-complete' {e0 = e} i (∞ x) = ∞ (δ-complete' e x)
δ-complete' (μ e) (∞ x) = ∞ (δ-complete' e x)

δ-complete {e = e} (∞ x) = ∞ (δ-complete' e x)
```

That's the end of the proof.

4 Discussion

Finally, we want to discuss three aspects of our work: expressiveness, performance, and simplicity.

[JR] TODO: μ-regular expressions have been studied before, cite

Expressiveness We conjecture that our grammars which include variables and fixed points can describe any context-free language. We have shown the example of balanced the bracket language which is known to be context-free. Furthermore, Grenrus shows that any context-free grammar can be converted to his grammars [4], which are similar to our grammars. The main problem is showing that mutually recursive nonterminals can be expressed using our simple fixed points, which requires Bekić's bisection lemma [2]. Formalizing this in our framework is future work.

Going beyond context-free languages, many practical programming languages cannot be adequately described as context-free languages. For example, features such as associativity, precedence, and indentation sensitivity cannot be expressed directly using context-free grammars. Recent work by Afroozeh and Izmaylova [1] shows that all these advanced features can be supported if we extend our grammars with data-dependencies. Our framework can form a foundation for such extensions and we consider formalizing it as future work.

Performance For a parser to practically useful, it must at least have linear asymptotic complexity for practical grammars. Might et al. [5] show that naively parsing using derivatives does not achieve that bound, but optimizations might make it possible. In particular, they argue that we could achieve $O(n|G|)$ time complexity (where $|G|$ is the grammar size) if the grammar size stays approximately constant after every derivative. By compacting the grammar, they conjecture it is possible to achieve this bound for any unambiguous grammar. We want to investigate if similar optimizations could be applied to our parser and if we can prove that we achieve this bound.

[JR] cite Jeremy Yallop's work

[JR] TODO: finish this paragraph

Simplicity One of the main contributions of Elliot's type theoretic formalization of languages [3] is its simplicity of implementation and proof. To be able to extend his approach to context-free languages we have had to introduce some complications.

In conclusion, we have formalized (acyclic) context-free grammars using a type theoretic approach to provide fertile ground for further formalizations of disambiguation strategies and parsers that are both correct and performant.

References

1. Afroozeh, A., Izmaylova, A.: One parser to rule them all. In: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). p. 151–170. Onward! 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2814228.2814242
2. Bekić, H.: Definable operations in general algebras, and the theory of automata and flowcharts, pp. 30–55. Springer Berlin Heidelberg, Berlin, Heidelberg (1984). https://doi.org/10.1007/BFb0048939, https://doi.org/10.1007/BFb0048939
3. Elliott, C.: Symbolic and automatic differentiation of languages. Proc. ACM Program. Lang. **5**(ICFP) (Aug 2021). https://doi.org/10.1145/3473583
4. Grenrus, O.: Fix-ing regular expressions (2020), https://well-typed.com/blog/2020/06/fix-ing-regular-expressions/, accessed: 2024-12-12
5. Might, M., Darais, D., Spiewak, D.: Parsing with derivatives: a functional pearl. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. p. 189–195. ICFP '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2034773.2034801