Context-free Languages, Type Theoretically

Anonymous

No Institute Given

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 such a mature research subject, it is still actively studied, for example the question of how to resolve ambiguities in context-free grammars [1].

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.

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 automated theorem provers as they do not support general recursion. We use a fuel-based approach to solve this problem.

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. Unfortunately, we are still working out the proof of three postulates in **??**. These are the only postulates that we have yet to prove.

1.1 Languages

We define languages as being functions from strings to types.[1]

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

```
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.

```
∅ : Lang                          _*_ : Lang → Lang → Lang
∅ _ = ⊥                           (P * Q) w = ∃[ u ] ∃[ v ] w ≡ u ++ v × P u × Q v

ϵ : Lang                          ‘_ : Char → Lang
ϵ w = w ≡ []                      (‘ c) w = w ≡ c ∷ []

_∪_ : Lang → Lang → Lang          _·_ : {A : Type} → Dec A → Lang → Lang
(P ∪ Q) w = P w ⊎ Q w             _·_ {A} _ P w = A × P w
```

Fig. 1. Basic language combinators.

For starters, we define some structure on this definition of language in Figure 1. First, Languages form a semiring, with union `_∪_`, concatenation `_*_`, the empty language `∅` which is the unit of union, and the language which only includes the empty string `ϵ` which is the unit of concatenation. Furthermore the `‘_` combinator defines a language which contains exactly the string consisting of a single given character. Finally, the scalar multiplication `_·_` combinator injects an Agda type into a language. The purpose of this combinator will become clearer in later sections.

[JR] mention specific sections

2 Grammars

We have seen in Example 1 that our definition of language is very general, comprising even context-sensitive languages. Parsing such languages automatically poses a significant challenge. Hence, we side-step this problem by restricting the scope of our parsers to a smaller well-defined subset of languages. In this subsection, we consider a subset of regular languages without Kleene star (i.e., closure under concatenation). In Section 3, we extend this class of languages to include fixed points which subsume the Kleene star.

```
data Exp : Type₁ where
  ∅ : Exp
  ϵ : Exp
  `_ : (c : Char) → Exp
  _·_ : {A : Type} → Dec A → Exp → Exp
  _∪_ : Exp → Exp → Exp
  _*_ : Exp → Exp → Exp
```

This syntax maps directly onto the semantics we defined in Figure 1.

```
⟦_⟧ : Exp → Lang
⟦ ∅ ⟧ = ◇.∅
⟦ ϵ ⟧ = ◇.ϵ
⟦ ` c ⟧ = ◇.` c
⟦ x · e ⟧ = x ◇.· ⟦ e ⟧
⟦ e ∪ e₁ ⟧ = ⟦ e ⟧ ◇.∪ ⟦ e₁ ⟧
⟦ e * e₁ ⟧ = ⟦ e ⟧ ◇.* ⟦ e₁ ⟧
```

2.1 Parsing

To facilitate proving the inclusion of strings in a language, we start by decomposing the problem. A string is either empty or a character followed by the tail of the string. We can decompose the problem of string inclusion along the same dimensions. First, we define nullability ν as the inclusion of the empty string in a language as follows:

```
◇ν : Lang → Type
◇ν ℒ = ℒ []
```

Second, we define the derivative δ of a language $\mathcal{L}$ with respect to the character c to be all the suffixes of the words in $\mathcal{L}$ which start with the c.

```
◇δ : Char → Lang → Lang
◇δ c ℒ = λ w → ℒ (c ∷ w)
```

The relevance of these definitions is shown by Theorem 1.

Theorem 1. *Nullability after repeated derivatives fully captures what a language is. Formally, we state this as follows:*

```
◇ν ∘ foldl ◇δ ℒ ≡ ℒ
```

```
ν : (e : Exp) → Dec (◇ν ⟦ e ⟧)
δ : Char → Exp → Exp
δ-sound : ∀ e → ⟦ δ c e ⟧ w → ◇δ c ⟦ e ⟧ w
δ-complete : ∀ e → ◇δ c ⟦ e ⟧ w → ⟦ δ c e ⟧ w
```

```
parse : (e : Exp) (w : String) → Dec (⟦ e ⟧ w)
parse e [] = ν e
parse e (c ∷ w) = map' (δ-sound e) (δ-complete e) (parse (δ c e) w)
```

2.2 Nullability

Lemma 1. *Two languages, $\mathcal{L}_1$ and $\mathcal{L}_2$, are nullable if and only if their concatenation, $\mathcal{L}_1 \diamond.* \mathcal{L}_2$, is nullable.*

```
ν* : (◇ν ℒ₁ × ◇ν ℒ₂) ⇔ ◇ν (ℒ₁ ◇.* ℒ₂)
```

```
ν ∅ = no λ ()
ν ϵ = yes refl
ν (‘ c) = no λ ()
ν (x · e) = x ×-dec ν e
ν (e ∪ e₁) = ν e ⊎-dec ν e₁
ν (e * e₁) = Dec.map ν* (ν e ×-dec ν e₁)
```

2.3 Derivation

```
δ c ∅ = ∅
δ c ϵ = ∅
δ c (‘ c₁) = (c ≟ c₁) · ϵ -- a bit interesting
δ c (x · e) = x · δ c e
δ c (e ∪ e₁) = δ c e ∪ δ c e₁
δ c (e * e₁) = (δ c e * e₁) ∪ (ν e · δ c e₁) -- interesting
```

The proofs are very straightforward:

```
δ-sound (‘ c) (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 (e * e₁) (inj₁ (u , v , refl , x , y)) = _ ∷ u , v , refl , δ-sound e x , y
δ-sound (e * e₁) (inj₂ (x , y)) = [] , _ , refl , x , δ-sound e₁ y
```

```
δ-complete (‘ c) refl = refl , refl
δ-complete (x₁ · 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) (_ ∷ _ , _ , refl , x , y) = inj₁ (_ , _ , refl , δ-complete e x , y)
δ-complete (e * e₁) ([] , _ , refl , x , y) = inj₂ (x , δ-complete e₁ y)
```

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)
```

Our approach uses the decomposition of languages into ν and δ. [JR] Now we should explain the $\diamond\nu$ and $\diamond\delta$

```
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 2. *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):

$$
\begin{array}{l}
\nu_1 : (e : \mathsf{Exp}) \to \mathsf{Dec}\ (.\diamond\nu\ (\llbracket\ e\ \rrbracket_1\ \diamond.\emptyset)) \\
\nu_1\ \emptyset = \mathsf{no}\ \lambda\ () \\
\nu_1\ \epsilon = \mathsf{yes\ refl} \\
\nu_1\ (\text{‘}\ c) = \mathsf{no}\ \lambda\ () \\
\nu_1\ (x\ \cdot\ e) = x \times\text{-}\mathsf{dec}\ \nu_1\ e \\
\nu_1\ (e \cup e_1) = \nu_1\ e \uplus\text{-}\mathsf{dec}\ \nu_1\ e_1 \\
\nu_1\ (e\ *\ e_1) = \mathsf{map'}\ (\lambda\ x \to [\,]\ ,\ [\,]\ ,\ \mathsf{refl}\ ,\ x)\ (\lambda\ \{\ ([\,]\ ,\ [\,]\ ,\ \mathsf{refl}\ ,\ x) \to x\ \})\ (\nu_1\ e \times\text{-}\mathsf{dec}\ \nu_1\ e_1) \\
\nu_1\ \mathsf{i} = \mathsf{no}\ \lambda\ () \\
\nu_1\ (\mu\ e) = \mathsf{map'}\ (\infty \circ \nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ e)\ (\nu\mathsf{ee}{\to}\nu\mathsf{e}\emptyset\ e \circ\ !)\ (\nu_1\ e)
\end{array}
$$

Using Lemma 2 we can define ν in terms of ν_1:

$$\nu\ e = \mathsf{map'}\ (\infty \circ \nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ e)\ (\nu\mathsf{ee}{\to}\nu\mathsf{e}\emptyset\ e \circ\ !)\ (\nu_1\ e)$$

[JR] TODO: show how ν works through examples

The forward direction is proven using straightforward induction.

$$
\begin{array}{l}
\nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ \epsilon\ x = x \\
\nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ (x_1\ \cdot\ e)\ (x\ ,\ y) = x\ ,\ \nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ e\ y \\
\nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ (e \cup e_1)\ (\mathsf{inj}_1\ x) = \mathsf{inj}_1\ (\nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ e\ x) \\
\nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ (e \cup e_1)\ (\mathsf{inj}_2\ y) = \mathsf{inj}_2\ (\nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ e_1\ y) \\
\nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ (e\ *\ e_1)\ ([\,]\ ,\ [\,]\ ,\ \mathsf{refl}\ ,\ x\ ,\ y) = [\,]\ ,\ [\,]\ ,\ \mathsf{refl}\ ,\ \nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ e\ x\ ,\ \nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ e_1\ y \\
\nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ \mathsf{i}\ () \\
\nu\mathsf{e}\emptyset{\to}\nu\mathsf{ee}\ (\mu\ e)\ x = x
\end{array}
$$

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

Lemma 3. *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:

$$
\begin{array}{l}
\nu\text{-}split : (e : Exp) \to .\diamond\nu\ (\llbracket\ e\ \rrbracket_1\ \llbracket\ e_0\ \rrbracket) \to .\diamond\nu\ (\llbracket\ e\ \rrbracket_1\ \diamond.\emptyset) \uplus .\diamond\nu\ (\llbracket\ e_0\ \rrbracket_1\ \diamond.\emptyset) \\
\nu\text{-}split\ \epsilon\ x = inj_1\ x \\
\nu\text{-}split\ (\_\ \cdot\ e)\ (x\ ,\ y) = Sum.map_1\ (x\ ,\_)\ (\nu\text{-}split\ e\ y) \\
\nu\text{-}split\ (e \cup e_1)\ (inj_1\ x) = Sum.map_1\ inj_1\ (\nu\text{-}split\ e\ x) \\
\nu\text{-}split\ (e \cup e_1)\ (inj_2\ y) = Sum.map_1\ inj_2\ (\nu\text{-}split\ e_1\ y) \\
\nu\text{-}split\ (e\ *\ e_1)\ ([\,]\ ,\ [\,]\ ,\ refl\ ,\ x\ ,\ y) = lift{\uplus}_2\ (\lambda\ x\ y \to [\,]\ ,\ [\,]\ ,\ refl\ ,\ x\ ,\ y)\ (\nu\text{-}split\ e\ x)\ (\nu\text{-}split\ e_1 \\
\nu\text{-}split\ \{e_0 = e\}\ i\ (\infty\ x) = inj_2\ (reduce\ (\nu\text{-}split\ e\ x)) \\
\nu\text{-}split\ (\mu\ e)\ x = inj_1\ x
\end{array}
$$

The backwards direction of Lemma 2 is now simply a result of Lemma 3 where both sides of the disjoint union are equal and thus we can reduce it to a single value.

$$\nu\mathsf{ee}{\to}\nu\mathsf{e}\emptyset\ e\ x = \mathsf{reduce}\ (\nu\text{-}\mathsf{split}\ \{e_0 = e\}\ e\ x)$$

3.4 Derivative correctness

[JR] At this point (specifically the _*_ case of δ_1) we need to introduce μ

Internal/syntactic substitution:

$$
\begin{array}{l}
\mathsf{sub} : \mathsf{Exp} \to \mathsf{Exp} \to \mathsf{Exp} \\
\mathsf{sub}\ \_\ \emptyset = \emptyset
\end{array}
$$

```
sub _ ε = ε
sub _ (` c) = ` c
sub e₀ (x · e) = x · sub e₀ e
sub e₀ (e ∪ e₁) = sub e₀ e ∪ sub e₀ e₁
sub e₀ (e * e₁) = sub e₀ e * sub e₀ e₁
sub e₀ i = e₀
sub _ (μ e) = μ e
```

We would like to be able to say ⟦ `sub e0 e` ⟧ ≡ ⟦ `e` ⟧$_1$ ⟦ `e0` ⟧\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 4. *(Internal) syntactic substitution is the same as (external) semantic substitution. This is the raison d'être of μ.*

Proof:

```
sub-sem' : (e : Exp) → ⟦ sub (μ e₀) e ⟧₁ l ≡ ⟦ e ⟧₁ ⟦ e₀ ⟧
sub-sem' ∅ = refl
sub-sem' ε = refl
sub-sem' (` _) = refl
sub-sem' (x · e) = cong (x ⋄.·_) (sub-sem' e)
sub-sem' (e ∪ e₁) = cong₂ ⋄._∪_ (sub-sem' e) (sub-sem' e₁)
sub-sem' (e * e₁) = cong₂ ⋄._*_ (sub-sem' e) (sub-sem' e₁)
sub-sem' i = refl
sub-sem' (μ _) = refl
```

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

```
sub-sem : (e : Exp) → ⟦ sub (μ e₀) e ⟧₁ l w ≡ ⟦ e ⟧₁ ⟦ e₀ ⟧ w
sub-sem e = cong (λ l → l _) (sub-sem' e)
```

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

```
δ₁ : (c : Char) → Exp → Exp → Exp
δ₁ c _ ∅ = ∅
δ₁ c _ ε = ∅
δ₁ c _ (` c₁) = (c ≟ c₁) · ε
δ₁ c e₀ (x · e) = x · δ₁ c e₀ e
δ₁ c e₀ (e ∪ e₁) = δ₁ c e₀ e ∪ δ₁ c e₀ e₁
δ₁ c e₀ (e * e₁) = (δ₁ c e₀ e * sub (μ e₀) e₁) ∪ (Dec.map (⇔.trans (mk⇔ ! ∞) (≡→⇔ (sub-sem e))) (ν (sub (μ e₀) e)) · δ₁ c
δ₁ c e₀ i = i
δ₁ c _ (μ e) = μ (δ₁ c e e)
```

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

```
δ c e = δ₁ c e e
```

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

The proofs are by induction and the Lemma 4:

```
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 (s
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
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.

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.

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. Most notably, we use fuel to define the semantics of our grammars. We have explored other approaches such as using guarded type theory, but we did not manage to significantly simplify our formalization. Furtheremore, we expect that many proofs remain simple despite our fuel-based approach.

In conclusion, we have (almost) formalized 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`