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 have extended the type theoretic formalization to context-free languages (without mutual recursion) which are 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: Parsing · Type Theory · Formal Languages

1 Introduction

Parsing—i.e., the process of recovering structure from strings—is an essential building block for many practical programming applications. While parsing has been extensively studied, it remains a relevant subject of research where new research questions continuously emerge. For example, how to compose grammars and parsers (e.g., [15]), dealing with ambiguous parse trees (e.g., [4,3,1]), and parsing grammar formalisms beyond context-free grammars (e.g., [1]). Research questions such as these serve a practical purpose, but answering them often requires a deep theoretical understanding of the semantics of parsing.

This theoretical understanding can be approached in different ways. Parsing is often studied using automata theory [10]. However, there is value in studying more *denotational* approaches to parsing. A main purpose of denotational semantics is to abstract away operational concerns, as such concerns tends to be a hindrance for equational reasoning. Such equational reasoning could be used to study and answer some of the open research questions in the parsers of today and tomorrow.

This paper studies the denotational semantics of parsing for context-free grammars. While the study is theoretical in nature, the motivation is to provide a foundation for practical future studies on proving the correctness of, e.g., parser optimizations and disambiguation techniques, as well as potentially providing a foundation for building and reasoning about parsers for more expressive grammar formalisms, such as data-dependent grammars [1].

We approach the question of giving a denotational semantics of parsing by building on existing work by Elliott [9]. In his recent work, Elliott demonstrated that regular grammars have a simple and direct denotational semantics. And that we can obtain parsers for such languages that are correct by construction, using *derivatives*. While it was well-known that we can parse regular grammars using Brzozowski derivatives [6], Elliott's work provides a simple and direct mechanization in Agda's type theory of the denotational semantics of these derivatives. This mechanization provides an implementation of parsing that is correct by construction, and that we can reason about without

relying on (bi-)simulation arguments. While the parsers obtained in this manner are not exactly performant, the denotational approach opens up the door to exploiting grammar structure to obtain optimized parsers.

Elliott leaves open the question of how the approach scales to more expressive grammar formalisms, such as context-free languages and beyond. However, the question of using derivatives to parse context-free grammars has been considered by others. Might et al. [13] demonstrate how to build parsers from context-free grammars using derivatives and optimizations applied to them, to obtain reasonable performance. Thiemann's work [16] uses lattice theory and powerset semantics to formalize a notion of partial derivative for a variant of context-free grammars. In this work, we build on the approach of Elliott and study how to build a simple and direct mechanization in Agda's type theory of the denotational semantics of derivatives for context-free grammars.

The main challenge for our mechanization is the question of how to deal with the recursive nature of context-free languages.

1.1 The Challenge with Parsing Context-Free Grammars

We give an overview of what it means to take the derivative of a grammar, how this provides an approach to parsing, and which challenges it poses.

To illustrate, consider the following context-free grammar of palindromic bit strings:

$$\langle pal \rangle ::= \epsilon \mid 0 \mid 1 \mid 0 \, \langle pal \rangle \, 0 \mid 1 \, \langle pal \rangle \, 1$$

Say we want to use this grammar to parse the string 0110. The idea of parsing with derivatives is to divide the problem into separate steps for each input symbol, each narrowing down the grammar. After all input symbols have been processed, we can simply check if the final grammar accepts the empty string to see if the whole input is accepted by the original grammar.

We call each of these transformation steps derivatives. Taking the derivative of a grammar with respect to an input symbol can be done naively by keeping only the productions which start with that symbol and then removing that symbol from the start of each of the productions. Empty productions (ϵ) are always removed when taking a derivative.

Using this naive derivation procedure on the $\langle pal \rangle$ grammar with respect to the bit 0 yields the following derived grammar:

$$\langle pal_0 \rangle ::= \epsilon \mid \langle pal \rangle \, 0$$

This grammar essentially represents the residual parsing obligations after parsing a 0 bit. The derived grammar contains fewer productions than the original grammar because we have pruned those productions that started with the terminal symbol 1.

For the next derivative step, we encounter a new special case: one of the productions starts with a nonterminal. A naive solution is to recursively unfold nonterminals until the production starts with a terminal symbol. For the $\langle pal \rangle$ grammar, we only have to unfold $\langle pal \rangle$ once which yields the following grammar:

$$\langle pal'_0 \rangle ::= \epsilon \mid 0 \, 0 \mid 1 \, 0 \mid 0 \, \langle pal \rangle \, 0 \, 0 \mid 1 \, \langle pal \rangle \, 1 \, 0$$

By continuing this procedure, with additional recursive unfolding where needed, we eventually yield a grammar that contains the the empty production ϵ, whereby we can conclude that 0110 is, in fact, a palindromic bit string.

However, naive recursive unfolding does not work for all grammars. Consider, for example, the infinitely recursive grammar:

$$\langle rec \rangle ::= \langle rec \rangle$$

Unfolding this grammar never reveals a starting nonterminal. While the $\langle rec \rangle$ grammar is contrived, the same issue arise for any *left-recursive* grammar, such as the following grammar of arithmetic expressions (left-recursive because of the $\langle expr \rangle$ non-terminal in the left-most position in the first production):

$$\langle expr \rangle ::= \langle expr \rangle \ + \ \langle expr \rangle \mid 0 \mid 1$$

Parsing context-free grammars, thus, requires more sophisticated techniques.

1.2 Contributions

This paper tackles the challenges discussed in the previous section by providing a mechanization in Agda of parsing a subset of context-free grammars with derivatives. The subset of grammars that we consider corresponds to context-free grammars without mutually recursive nonterminals. In other words, the grammars can consist of multiple nonterminals which may refer to themselves and others, but there may not be a cycle which involves more than one nonterminal. For example, the following mutually recursive grammar does not fit into the subset of grammars we consider:

$$\langle expr \rangle ::= \langle expr \rangle \ + \ \langle expr \rangle \mid 0 \mid 1 \mid \langle stmt \rangle$$
$$\langle stmt \rangle ::= \langle expr \rangle \mid \langle stmt \rangle ; \langle stmt \rangle$$

The $\langle pal \rangle$, $\langle rec \rangle$, and $\langle expr \rangle$ grammars from the previous section are all examples of grammars that are in the subset. We conjecture that the approach we present later can be extended to deal with all context-free grammars, at the cost of some additional bookkeeping. We leave verifying this conjecture as a challenge for future work.

We make the following technical contributions:

- We provide a semantics in Agda of context-free grammars without mutual recursion.
- We provide a derivative-based parser for this class of grammars, along with its simple and direct correctness proof.

The paper assumes basic familiarity with Agda. The rest of this paper is structured as follows. Section 2 recalls the essential definition from Elliott's work which we subsequently extend in Section 3 to context-free grammars. Section 4 discusses expressiveness, performance, and simplicity of our approach, whereas Section 5 discusses related work, and Section 6 concludes.

2 Finite Languages

In this section, we introduce background information, namely how we define languages, basic language combinators, and parsers. Our exposition follows Elliott [9]. 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. In other words, we want a parser which can determine by itself whether a string is in the language or not.

Unfortunately, we cannot hope to write a parser for arbitrary languages defined in this way. A language could be defined, for example, such that the inclusion of a particular string is predicated on whether or not the Collatz conjecture holds. Therefore, we need to restrict ourselves to a subset of languages. Next, we explore basic language combinators for this purpose.

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 grammar:

⟨*permissions*⟩ ::= ⟨*read*⟩ ⟨*write*⟩ ⟨*execute*⟩

⟨*read*⟩ ::= - | r

⟨*write*⟩ ::= - | w

⟨*execute*⟩ ::= - | x

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

∅ : Lang
∅ _ = ⊥

_∪_ : Lang → Lang → Lang
(P ∪ Q) w = P w ⊎ Q w

ϵ : Lang
ϵ w = w ≡ []

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

_·_ : Type → Lang → Lang
(A · P) w = A × P w
```

Fig. 1. Basic language combinators.

This grammar uses three important features: sequencing, choice, and matching character literals. We can define these features as combinators on languages in Agda as shown in the left column of Figure 1. Using these combinators we can define our permissions language as follows:

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

The right column of Figure 1 lists combinators whose purpose will become clear when we discuss how to write parsers for this simple language in the next section.

2.3 Parsers

We want to write a program which can automatically prove for us whether or not a given string is in a 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 → Type
Parser P = (w : String) → Dec (P w)
```

[3] We use Type as a synonym for Agda's Set to avoid confusion with set-theoretic sets.

Remark 2. Readers familiar with Haskell might notice the similarity between the type $(w : \mathsf{String}) \to \mathsf{Dec}\ (P\ w)$ and the type `String -> Maybe a`, which is a common 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)]` following Hutton and Meijer [11], which allows a parser to return multiple results but does nothing to ensure that it correctly produces all possible results. We could imagine replacing `Dec` by a requirement that the result type is in bijection with a possibly infinite set. However, that would introduce too many complications in our proofs. In practice, furthermore, we want our parsers to only give us a single result. Hence, our effort would be better spent in proving that our languages are unambiguous, meaning there is at most one valid way to parse each input string. Thus, in this paper, we use Dec.

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 seems ad hoc and it is hard to read, especially considering this is one of the simpler combinators.

Following the approach of parsing with derivatives, we can factor this parser into two cases: the empty string case and the case with at least one character. We call the former nullability and denote it with the greek character ν, and we call the latter derivative and denote it with the greek character δ.

Crucially, nullability deals only with (decidable) types, and derivatives deal only with languages. This clearly separates the level of abstraction between both cases.

Returning to our character parser, a single character language is not nullable. On the level of types we express this as $\bot$, the uninhabited type, which is trivially decidable as no λ ().

The derivative of a single character language depends on whether the character of the derivative is the same as the character of the language. We might be tempted to define this condition externally in Agda, but that would break the abstraction of derivatives only dealing with languages. Instead, we are pushed toward defining a combinator, $\_ \cdot \_$, which allows us to express this conditional on the level of languages. If the condition holds then there is still a second condition which is that the remainder of the string needs to be empty. We use the epsilon language, ϵ, for that purpose. To

$$\nu\ P = P\ []$$

$$(\delta\ c\ P)\ w = P\ (c :: w)$$

$$A \Leftrightarrow B = (A \to B) \times (B \to A)$$

$$P \Longleftrightarrow Q = \forall\ \{w\} \to P\ w \Leftrightarrow Q\ w$$

$$\begin{array}{lll}
\nu\emptyset & : \bot & \Leftrightarrow \nu\ \emptyset \\
\nu\epsilon & : \top & \Leftrightarrow \nu\ \epsilon \\
\nu\cdot & : (A \times \nu\ P) & \Leftrightarrow \nu\ (A\ \cdot\ P) \\
\nu\text{`} & : \bot & \Leftrightarrow \nu\ (\text{`}\ c\text{'}) \\
\nu\cup & : (\nu\ P \uplus \nu\ Q) & \Leftrightarrow \nu\ (P \cup Q) \\
\nu* & : (\nu\ P \times \nu\ Q) & \Leftrightarrow \nu\ (P * Q)
\end{array}$$

$$\begin{array}{lll}
\delta\emptyset & : \emptyset & \Longleftrightarrow \delta\ c\ \emptyset \\
\delta\epsilon & : \emptyset & \Longleftrightarrow \delta\ c\ \epsilon \\
\delta\cdot & : (A\ \cdot\ \delta\ c\ P) & \Longleftrightarrow \delta\ c\ (A\ \cdot\ P) \\
\delta\text{`} & : ((c \equiv c')\ \cdot\ \epsilon) & \Longleftrightarrow \delta\ c\ (\text{`}\ c\text{'}) \\
\delta\cup & : (\delta\ c\ P \cup \delta\ c\ Q) & \Longleftrightarrow \delta\ c\ (P \cup Q) \\
\delta* & : (\nu\ P\ \cdot\ \delta\ c\ Q \cup \delta\ c\ P * Q) & \\
 & & \Longleftrightarrow \delta\ c\ (P * Q)
\end{array}$$

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

conclude, the derivative of the character language ` c' with respect to the character c is $(c \stackrel{?}{=} c')\ \cdot\ \epsilon$ as shown in Figure 2.

Furthermore, Figure 2 shows the nullability and derivatives of all basic combinators using simple and self-contained equivalances. The implementation of parsers for our basic combinators follow completely from the decomposition into nullability and derivatives and these equivalances. For example, we can rewrite our character parser as follows:

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

Parsers for the other basic combinators are equally straightforward and can be found in our source code artifact.

The parser for our full permissions language can now be implemented 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')
```

This allows us to generate a parser for any language that is defined using the basic combinators from Figure 1. We mechanize this result later in Section 3.3, but we first consider extending the expressivity of our combinators.

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. Elliott [9] defines a Kleene star combinator which enables him to specify regular languages such as file paths.

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

$$\langle brackets \rangle \quad ::= \epsilon \mid [\ \langle brackets \rangle\ ] \mid \langle brackets \rangle\ \langle brackets \rangle$$

This is the language of all strings which consist of balanced square brackets. It is common for programming languages to 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 we have added a NON_TERMINATING pragma to make Agda to accept it anyway, but this is obviously not the proper way to define our brackets language.

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

We need to find a different way to encode this recursive relation. A fixed point combinator could resolve this issue as follows:

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

Unfortunately, such a fixed point combinator does not exists for arbitrary functions on languages. Luckily, we will see in the next section that we can define such a fixed point combinator if we restrict the class of functions of which we take the fixed point.

3 Context-free Languages

3.1 Fixed Points

To be able to specify the recursive structure of context-free languages, we need a fixed point. From type theory we know that a fixed point of a functor $F : \mathsf{Type} \to \mathsf{Type}$ is well-defined if it is strictly positive. So we could restrict the argument of our fixed point combinator to only accept strictly positive functors. We are dealing with languages and not types directly, but luckily our definition of language is based on types and our basic combinators are strictly positive. One catch is that Agda currently has no way of directly expressing that a functor is strictly positive.[4] We can still make this evident to Agda by defining a data type of descriptions as shown by Chapman et al. [7].

```
data Desc : Type₁ where
  ∅    : Desc
  ϵ    : Desc
  ‘_   : Char → Desc
```

[4] There is work on implementing positivity annotations.[14]

```
_∪_ : Desc → Desc → Desc
_*_ : Desc → Desc → Desc
_ · _ : {A : Type} → Dec A → Desc → Desc
var   : Desc
```

We can give semantics to our descriptions in terms of languages that we defined in Section 2. We use the $\diamond$ prefix to refer to the language combinators defined in Section 2.

```
⟦_⟧o : Desc → Lang → Lang
⟦ ∅ ⟧o              _ = ◇∅
⟦ ϵ ⟧o              _ = ◇ϵ
⟦ ' c ⟧o            _ = ◇' c
⟦ D₁ ∪ D₂ ⟧o        P = ⟦ D₁ ⟧o P ◇∪ ⟦ D₂ ⟧o P
⟦ D₁ * D₂ ⟧o        P = ⟦ D₁ ⟧o P ◇* ⟦ D₂ ⟧o P
⟦ _ · _ {A} _ D ⟧o  P = A ◇· ⟦ D ⟧o P
⟦ var ⟧o            P = P
```

Using these descriptions, we can define a fixed point as follows:

```
data ⟦_⟧ (D : Desc) : Lang where
  roll : ⟦ D ⟧o ⟦ D ⟧ w → ⟦ D ⟧ w


unroll : ⟦ D ⟧ w → ⟦ D ⟧o ⟦ D ⟧ w
unroll (roll x) = x
```

With this fixed point, we can finally define the brackets language.[5]

```
bracketsD = ϵ ∪ ' '[' * var * ' ']' ∪ var * var
brackets = ⟦ bracketsD ⟧
```

This representation is not modular, however. We cannot nest fixed points in descriptions. For example, we could not create a new language which contains the brackets language as a subexpression, because the fixed point is only taken over the whole descriptor. This problem comes up naturally when considering derivatives, which we discuss next.

3.2 Derivatives by Example

As we have seen with finite languages in Section 2, when writing parsers it is useful to consider how a language changes after one character has been parsed. We will call this the *derivative* of the original language. For example, we could consider what happens to our brackets languages after one opening brackets has been parsed: δ `'['` brackets. In this section, we search for a description of this reduced language.

We can systematically deduce the derivative language from the brackets definition by going over each of the disjuncts. The first disjunct, ϵ, does not play a role because we know the string contains at least the opening bracket. The second disjunct, brackets surrounding a self-reference, is trickier.

[5] We split this definition into two because we want to separately reuse the description later.

The opening bracket clearly matches, but it would be a mistake to say the new disjunct should be a self-reference followed by a closing bracket: var ∗ ' ']'.

Note that the self-reference in the new language would refer to the derivative of the old language, not to the old language itself. We would like to refer to the original bracket language: brackets ∗ ' ']', but we cannot nest the brackets language into another description.

There are cases where we do want to use self-reference in the new language. Consider the third disjunct, var ∗ var. It is a sequence so we expect from the finite case of Section 2 that matching one character results in two new disjuncts: one where the first sequent matches the empty string and then we take the derivative of the second, and one where we take the derivative of the first and the second is unchanged. In this case both sequents are self-references, so we need to know three things:

1. Does the original language match the empty string?
2. What is the derivative of the language?
3. What does it mean for the language to remain the same?

At first glance, the last point seems obvious, but remember that we are taking the derivative of the language, so self-references will change meaning even if they remain unchanged. Similarly to the previous disjunct, we want to refer to the original brackets in this case. To resolve this issue of referring to the original brackets expression, we introduce a new combinator μ, which has the meaning of locally taking a fixed point of a subexpression.

```
data Desc : Type₁ where
  – ...
  μ : Desc → Desc

⟦_⟧ₒ : Desc → Lang → Lang
– ...
⟦ μ D ⟧ₒ _ = ⟦ D ⟧
```

The first question is easy to answer: yes, the first disjunct of brackets is ϵ which matches the empty string.

```
νbrackets : Dec (ν brackets)
νbrackets = yes (roll (inj₁ refl))
```

The second question is where having a self-reference in the new language is useful. We can refer to the derivative of brackets by using self-reference.

Using these answers, we can write the derivative of brackets with respect to the opening bracket.

```
bracketsD' = μ bracketsD ∗ ' ']' ∪ νbrackets · var ∪ var ∗ μ bracketsD
brackets'  = ⟦ bracketsD' ⟧
```

This example has illustrated three important points which we should keep in mind:

– We can reuse many of the results of finite languages (Section 2).
– We need a new μ combinator to nest fixed points in descriptions. This is necessary to refer back to the original language before derivation.
– Taking the derivative of a self-reference simply results in a self-reference again, because self-references in the derivative refer to the derivative.

Again, we do not want to have to manually construct these derivatives. Instead, we show how to do it in general for any description in the next section.

3.3 Parsing in General

Our goal is to define a parse function for every description fixed point.

```
parse : ∀ D → Parser ⟦ D ⟧
```

We approach this by decomposing parsing into computing nullability and derivatives separately as follows:

```
νD : ∀ D → Dec (ν ⟦ D ⟧)
δD : Char → Desc → Desc
```

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

```
δD-correct : ⟦ δD c D ⟧ ⟺ δ c ⟦ D ⟧
```

The actual parsing can now be done character by character:

```
parse D [] = νD D
parse D (c :: w) = Dec.map δD-correct (parse (δD c D) w)
```

That is the main result of this paper. The remainder of the paper concerns the implementation of νD, δD, δD-correct.

3.4 Nullability

If we know the nullability of a language, P, then the nullability of a description functor applied to P is the same as the empty string parsers for our finite languages, but with the nullability of the variables given by the nullability of P. For the μ case we use the nullability of the fixed point, which we will implement shortly. The cases for the basic combinators are the same as in Figure 2. We only use this explicitly in the sequencing case because the other cases are simple enough to implement directly.

```
ν₀ : Dec (ν P) → ∀ D → Dec (ν (⟦ D ⟧₀ P))
ν₀ _ ∅       = no λ ()
ν₀ _ ϵ       = yes refl
ν₀ _ (' c)   = no λ ()
ν₀ z (D ∪ D₁) = ν₀ z D ⊎-dec ν₀ z D₁
ν₀ z (D ∗ D₁) = Dec.map ν∗ (ν₀ z D ×-dec ν₀ z D₁)
ν₀ z (x · D)  = x ×-dec ν₀ z D
ν₀ z var     = z
ν₀ _ (μ D)   = νD D
```

Naively, we might try νD $D = \nu_o$ (νD D) D, but that will not terminate. Consider the language ⟦ var ⟧, to determine the nullability of this language we would need to know the its nullability. Instead, we use the following lemma.

Lemma 1. *The nullability of a fixed point is determined completely by a single application of the underlying functor to the empty language.*

```
νDØ⇔νD : ν (⟦ D ⟧ₒ ⋄Ø) ⇔ ν ⟦ D ⟧
```

Proof. The forward direction is easily proven by noting that nullability and the semantics of a description are functors and that the empty language is initial. It is also straightforward to write the proof directly.

```
νDØ→νD : ∀ D → ν (⟦ D ⟧ₒ ⋄Ø) → ν (⟦ D ⟧ₒ ⟦ D₀ ⟧)
```

The backwards direction is more difficult. We prove a more general lemma from which our desired result follows. The generalized lemma states that, if the application of a descriptor functor to a fixed point of another descriptor is nullable, then either the fixed point plays no role and the descriptor functor is also nullable if applied to the empty language, or the other descriptor (that we took the fixed point of) is nullable when applied to the empty language.

```
νDØ←νD : ∀ D → ν (⟦ D ⟧ₒ ⟦ D₀ ⟧) → ν (⟦ D ⟧ₒ ⋄Ø) ⊎ ν (⟦ D₀ ⟧ₒ ⋄Ø)
```

If we choose $D_0 = D$ then both cases of the resulting disjoint union have the same type, so we can just pick whichever of the two we get as a result using the reduce : $A \uplus A \to A$ function. Modulo wrapping and unwrapping of the fixed point (using the roll constructor), we now have the two functions which prove the lemma:

```
νDØ⇔νD {D} = mk⇔ (roll ∘ νDØ→νD D) (reduce ∘ νDØ←νD {D₀ = D} D ∘ unroll)
```

From Lemma 1, we know that it is sufficient to only look one layer deep when determining the nullability of a fixed point. We can safely assume the argument is not nullable.

```
νD = Dec.map νDØ⇔νD ∘ νₒ (no λ ())
```

Remark 4. Lemma 1 does not define an isomorphism on types. In particular, the backwards direction is not injective. Consider the brackets language. It has the following null element, where we first choose the third disjunct, var ∗ var, and then the first disjunct ϵ for both branches.

```
brackets₀ : ν brackets
brackets₀ = roll (inj₂ (inj₂ ([] , [] , refl , roll (inj₁ refl) , roll (inj₁ refl))))
```

When we round-trip this through our lemma, we get a different result:

```
brackets₀' : νDØ⇔νD {bracketsD} .to (νDØ⇔νD {bracketsD} .from brackets₀)
           ≡ roll (inj₁ refl)
brackets₀' = refl
```

It now directly takes the first disjunct, ϵ.

In practice, such problems should be avoided by using unambiguous languages, ensuring that there is only one valid parse result for each string. In that case, only one of the two solutions are possible and we would have to specify exactly which one we intend to allow when defining the language.

3.5 Derivatives

The final piece of the puzzle are derivatives. They tell us how the language descriptions change after parsing each input character.

In Section 3.2, we established that the meaning of self-references changes and thus they need to be replaced by local fixed points of the original language. We define a function $\sigma\mathsf{D}$ to perform this substitution. It is a simple recursive function which replaces the var constructor with a given D' description.

$$
\begin{array}{lll}
\sigma : \mathsf{Desc} \to \mathsf{Desc} \to \mathsf{Desc} & & \\
\sigma\ \emptyset & D' = \emptyset \\
\sigma\ \epsilon & D' = \epsilon \\
\sigma\ (`\ c) & D' = `\ c \\
\sigma\ (D \cup D_1) & D' = \sigma\ D\ D' \cup \sigma\ D_1\ D' \\
\sigma\ (D * D_1) & D' = \sigma\ D\ D' * \sigma\ D_1\ D' \\
\sigma\ (x\ \cdot\ D) & D' = x\ \cdot\ \sigma\ D\ D' \\
\sigma\ \mathsf{var} & D' = D' \\
\sigma\ (\mu\ D) & D' = \mu\ D
\end{array}
$$

It turns out that the only the sequencing case, $*$ leaves the variables untouched, thus we only need to apply the substitution there. This substitution does mean we need to keep track of the original description, D_0, through the recursion. Most other cases follow the structure we uncovered in Figure 2. For the self-reference case, var, we produce a self-reference again, which works because it now refers to the derivative. Finally, for the internal fixed point, μ, we can simply recursively call the derivative function. Thus, our derivative helper function is defined as follows:

$$
\begin{array}{lll}
\delta_o : \mathsf{Desc} \to \mathsf{Char} \to \mathsf{Desc} \to \mathsf{Desc} & & \\
\delta_o\ D_0\ c\ \emptyset & = \emptyset \\
\delta_o\ D_0\ c\ \epsilon & = \emptyset \\
\delta_o\ D_0\ c\ (`\ c') & = (c \stackrel{?}{=} c')\ \cdot\ \epsilon \\
\delta_o\ D_0\ c\ (D \cup D_1) & = \delta_o\ D_0\ c\ D \cup \delta_o\ D_0\ c\ D_1 \\
\delta_o\ D_0\ c\ (D * D_1) & = \nu_o\ (\nu\mathsf{D}\ D_0)\ D\ \cdot\ \delta_o\ D_0\ c\ D_1 \cup \delta_o\ D_0\ c\ D * \sigma\ D_1\ (\mu\ D_0) \\
\delta_o\ D_0\ c\ (x\ \cdot\ D) & = x\ \cdot\ \delta_o\ D_0\ c\ D \\
\delta_o\ D_0\ c\ \mathsf{var} & = \mathsf{var} \\
\delta_o\ D_0\ c\ (\mu\ D) & = \mu\ (\delta\mathsf{D}\ c\ D)
\end{array}
$$

At the top level, we simply delegate to the helper by passing $D_0 = D$.

$$\delta\mathsf{D}\ c\ D = \delta_o\ D\ c\ D$$

Lemma 2. *Substitution of a local fixed point into a description is the same as applying the corresponding functor to the semantic fixed point.*

$$\sigma\mu : \forall\ D \to [\![\ \sigma\ D\ (\mu\ D_0)\ ]\!]_o\ P\ w \equiv [\![\ D\ ]\!]_o\ [\![\ D_0\ ]\!]\ w$$

The proof follows directly by induction and computation.

To prove the correctness of the derivative, we consider both directions of the equivalence separately. Furthermore, it is important that we separate the top-level description, D_0, from the current description, D, as they need to change independently throughout the induction.

$$\delta\text{D-to} : \forall\ D \to [\![\ \delta_o\ D_0\ c\ D\ ]\!]_o\ [\![\ \delta\text{D}\ c\ D_0\ ]\!]\ w \to \delta\ c\ ([\![\ D\ ]\!]_o\ [\![\ D_0\ ]\!])\ w$$

$$\delta\text{D-from} : \forall\ D \to \delta\ c\ ([\![\ D\ ]\!]_o\ [\![\ D_0\ ]\!])\ w \to [\![\ \delta_o\ D_0\ c\ D\ ]\!]_o\ [\![\ \delta\text{D}\ c\ D_0\ ]\!]\ w$$

The proofs follow from induction, the equivalances in Figure 2, and Lemma 2. Finally, we combine these two direction into our desired equivalance:

$$\delta\text{D-correct}\ \{D = D\} = \text{mk}\Leftrightarrow\ (\text{roll} \circ \delta\text{D-to}\ D \circ \text{unroll})\ (\text{roll} \circ \delta\text{D-from}\ D \circ \text{unroll})$$

This completes our proof.

4 Discussion

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

Expressiveness Our language descriptions are similar to μ-regular expressions, which have been shown to be equivalent to context-free grammars [16]. The major difference is that we only allow variables to refer to the nearest enclosing fixed point and not back to earlier fixed points. Each fixed point roughly corresponds to a nonterminal in the context-free grammar, so we conjecture this limitation means we only support context-free grammars without mutual recursion. We hope to expand our work to cover mutually recursive context-free grammars in future work.

Furthermore, an anonymous reviewer of a draft of this paper has pointed out that simply showing an equivalence of expressive power does not mean μ-regular expressions are as easy to use as context-free grammars in practice. In future work, we hope to explore this further by specifying and parsing more representative and practical languages.

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. In future work, we want to explore using our framework as a foundation for such extensions.

Performance For a parser to practically useful, it must at least have linear asymptotic complexity for practical grammars. Might et al. [13] 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 all unambiguous grammars. In future work, 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 Elliott's type theoretic formalization of languages [9] is its simplicity of implementation and proof. To be able to extend his approach to context-free languages we have had to introduce language descriptors, which are a slight complication. For example, descriptors made it harder to parameterize the symbolic representation of a language with its semantics as Elliott does in his paper, allowing him to write a correct by construction derivative function. Instead, we have split the derivative function into a syntactic transformation and an external correctness proof. Nevertheless, we think our approach retains much of the simplicity of Elliott's work.

5 Related Work

Formal languages have a long history. We refer the interested reader to Hopcroft et al. [10] which is an overview of traditional formal language theory.

The main inspiration for our work is the work by Elliott on automatic and symbolic differentiation of languages [9]. As the title suggests, it shows a duality between two styles of implementing language derivatives in Agda. In this paper, we focus on the symbolic approach to differentiation, but we still benefit from Elliott's clear and concise presentation. Our work is an extension of Elliott's symbolic differentiation to a more expressive subset of context-free languages.

Previous work has implemented context-free (or similar) languages in Agda. For example, Danielsson [8] and Allais [2] show how to implement a form of parser combinators in Agda. Both ensure termination by requiring that parsers consume at least some input each recursion. In our work, we lift this restriction, freeing programmers from having to ensure their parsers consume input.

Another approach to writing context-free grammars in Agda is to first convert arbitrary context-free grammars to a form more amenable to parsing. For example, Brink et al. [5] show how to formalize the left-corner transformation in Agda, which removes left-recursion from the grammar, thus allowing the use of a more naive parsing algorithm. Another example of this approach is by Bernardy and Jannson, who first transform the grammar into Chomsky Normal Form and subsequently formalize and the efficient Valiant's algorithm. For the sake of simplicity, we avoid such pre-processing transformations in our work.

Perhaps the closest related work to ours can be found outside of Agda formalizations, namely Thiemann's [16] work on partial derivatives for context-free languages. His work does cover mutually recursive nonterminals and furthermore relates derivative-based parsers to pushdown automata. In contrast to our type-theoretic approach, Thiemann's approach is based on set theory which means languages are just sets of strings and the result of parsing is only the boolean which tells you whether the input string is in the language or not. In this way, the information about the tree structure that naturally results from parsing—and which is often desired in practice—remains implicit. Furthermore, our proofs are mechanized in Agda, which gives us confidence in the correctness, but also facilitates computer-aided experimentation.

Finally, there are some works which focus less on formalization and more on the practical implementation of parsing μ-regular languages. Might et al. [13] show how to parse μ-regular languages using derivatives. They use laziness and memoization to avoid nontermination. Krishnaswami and Yallop [12] propose an alternative approach to parsing μ-regular expressions. They introduce a type system which enforces their languages to be in LL(1), which they parse efficiently using staging.

6 Conclusion

In conclusion, we have formalized (non mutually recursive) 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. Allais, G.: Agdarsec - total parser combinators. pp. 45–59 (Feb 2018), publisher Copyright: © JFLA 2018 - Journees Francophones des Langages Applicatifs. All rights reserved. Sylvie Boldo, Nicolas Magaud. Journées Francophones des Langages Applicatifs 2018. Sylvie Boldo; Nicolas Magaud. Journées Francophones des Langages Applicatifs 2018, Jan 2018, Banyuls-sur-Mer, France. publié par les auteurs, 2018. ⟨hal-01707376⟩; Vingt-neuviemes Journees Francophones des Langages Applicatifs, JFLA 2018 - 29th French-Speaking Conference on Applicative Languages, JFLA 2018 ; Conference date: 24-01-2018 Through 27-01-2018
3. Basten, B.: Ambiguity Detection for Programming Language Grammars. Theses, Universiteit van Amsterdam (Dec 2011), https://theses.hal.science/tel-00644079
4. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Sci. Comput. Program. **75**(3), 176–191 (2010). https://doi.org/10.1016/J.SCICO.2009.11.002, https://doi.org/10.1016/j.scico.2009.11.002
5. Brink, K., Holdermans, S., Löh, A.: Dependently typed grammars. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) Mathematics of Program Construction. pp. 58–79. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
6. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (Oct 1964). https://doi.org/10.1145/321239.321249
7. Chapman, J., Dagand, P.E., McBride, C., Morris, P.: The gentle art of levitation. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. p. 3–14. ICFP '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1863543.1863547, https://doi.org/10.1145/1863543.1863547
8. Danielsson, N.A.: Total parser combinators. SIGPLAN Not. **45**(9), 285–296 (Sep 2010). https://doi.org/10.1145/1932681.1863585, https://doi.org/10.1145/1932681.1863585
9. Elliott, C.: Symbolic and automatic differentiation of languages. Proc. ACM Program. Lang. **5**(ICFP) (Aug 2021). https://doi.org/10.1145/3473583
10. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, 3rd Edition. Pearson international edition, Addison-Wesley (2007)
11. Hutton, G., Meijer, E.: Monadic parsing in haskell. Journal of Functional Programming **8**(4), 437–444 (1998). https://doi.org/10.1017/S0956796898003050
12. Krishnaswami, N.R., Yallop, J.: A typed, algebraic approach to parsing. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 379–393. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314625, https://doi.org/10.1145/3314221.3314625
13. 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

14. Poiret, J., Escot, L., Ceulemans, J., Altenmüller, M., Nuyts, A.: Read the mode and stay positive. In: 29th International Conference on Types for Proofs and Programs, Location: Valencia, Spain (2023)
15. Schwerdfeger, A., Wyk, E.V.: Verifiable parse table composition for deterministic parsing. In: van den Brand, M., Gasevic, D., Gray, J. (eds.) Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5969, pp. 184–203. Springer (2009). https://doi.org/10.1007/978-3-642-12107-4_15, https://doi.org/10.1007/978-3-642-12107-4_15
16. Thiemann, P.: Partial derivatives for context-free languages - from \mu -regular expressions to pushdown automata. In: Esparza, J., Murawski, A.S. (eds.) Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10203, pp. 248–264 (2017). https://doi.org/10.1007/978-3-662-54458-7_15, https://doi.org/10.1007/978-3-662-54458-7_15