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] Elliott has kicked off this effort...

Recent work by Elliott 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 [2]. 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 Elliott'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 Elliott'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 Finite Languages

In this section, we introduce background information, namely how we define languages, basic language combinators, and parsers. Our exposition follows Elliott [2]. 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

$$\langle \mathit{permissions} \rangle ::= \langle \mathit{read} \rangle \ \langle \mathit{write} \rangle \ \langle \mathit{execute} \rangle$$

$$\langle \mathit{read} \rangle ::= \text{`\texttt{-}'} \mid \text{`\texttt{r}'}$$

$$\langle \mathit{write} \rangle ::= \text{`\texttt{-}'} \mid \text{`\texttt{w}'}$$

$$\langle \mathit{execute} \rangle ::= \text{`\texttt{-}'} \mid \text{`\texttt{x}'}$$

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.

$$
\begin{array}{ll}
\text{‘\_} : \mathsf{Char} \to \mathsf{Lang} & \emptyset : \mathsf{Lang} \\
(\text{‘}\ c)\ w = w \equiv c :: [] & \emptyset\ \_ = \bot \\[1em]
\_\cup\_ : \mathsf{Lang} \to \mathsf{Lang} \to \mathsf{Lang} & \epsilon : \mathsf{Lang} \\
(P \cup Q)\ w = P\ w \uplus Q\ w & \epsilon\ w = w \equiv [] \\[1em]
\_*\_ : \mathsf{Lang} \to \mathsf{Lang} \to \mathsf{Lang} & \_\cdot\_ : \mathsf{Type} \to \mathsf{Lang} \to \mathsf{Lang} \\
(P * Q)\ w = \exists[\ u\ ]\ \exists[\ v\ ]\ w \equiv u +\!\!+\ v \times P\ u \times Q\ v & (A\ \cdot\ P)\ w = A \times P\ w
\end{array}
$$

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.

```
ν P = P []                                  (δ c P) w = P (c :: w)

A ⇔ B = (A → B) × (B → A)                   P ⟺ Q = ∀ {w} → P w ⇔ Q w

ν∅  : ⊥                ⇔ ν ∅                δ∅  : ∅                          ⟺ δ c ∅
νϵ  : ⊤                ⇔ ν ϵ                δϵ  : ∅                          ⟺ δ c ϵ
ν·  : (A × ν P)        ⇔ ν (A · P)          δ·  : (A · δ c P)                ⟺ δ c (A · P)
ν‘  : ⊥                ⇔ ν (‘ c’)           δ‘  : ((c ≡ c’) · ϵ)             ⟺ δ c (‘ c’)
ν∪  : (ν P ⊎ ν Q)      ⇔ ν (P ∪ Q)          δ∪  : (δ c P ∪ δ c Q)            ⟺ δ c (P ∪ Q)
ν∗  : (ν P × ν Q)      ⇔ ν (P ∗ Q)          δ∗  : (ν P · δ c Q ∪ δ c P ∗ Q)
                                                                             ⟺ δ c (P ∗ Q)
```

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. Elliott [2] 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. An essential feature of many languages is the ability to recognize balanced brackets. A minimal example language with balanced brackets is the following:

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

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.

[JR] todo: flesh out this outline

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 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, with that exact type, cannot be implemented
- The Lang → Lang function needs to be restricted

[JR] Can we give a concrete example of how Lang → Lang is too general?

3 Context-free Languages

3.1 Fixed Points

[JR] Make it clear that we depart from Elliott's work at this point.

- If F : Type → Type is a strictly positive functor, then we know its fixed point is well-defined.
- 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 such as those used in the paper ”gentle art of levitation”.

[JR] todo:

```
data Desc : Type₁ where
  ∅     : Desc
  ϵ     : Desc
  ‘_    : Char → Desc
  _∪_   : Desc → Desc → Desc
  _∗_   : Desc → Desc → Desc
  -- We need Dec if we want to be able to write parsers
  -- but for specifiction it is not really needed
  _·_   : {A : Type} → Dec A → Desc → Desc
  var   : Desc
```

We can give semantics to our descriptions in terms of languages that we defined in the previous section.

[JR] todo: proper ref

```
⟦_⟧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
```

So we can finally define the brackets language.[5]

[JR] Brackets is one example, but can we characterise the whole class of languages we can define using these descriptions?

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

This representation is not modular, however. We cannot nest fixed points in descriptions. This problem comes up naturally when considering reduction, which we discuss next.

[JR] This modularity and nesting is not clear enough.

3.2 Reduction 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 *reduction*. For example, we could consider what happens to our brackets languages after

[4] There is work on implementing positivity annotations.

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

one opening brackets has been parsed: δ '[' brackets. In this section, we search for a description of this reduced language (the *reduct*).

We can mechanically derive this new 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. 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 the second is reduced and one where the first is reduced and the second is unchanged. In this case both sequents are self-references, so we need to know three things: [JR] Why? That is what we saw in Section 2

1. Does the original language match the empty string?
2. What is the reduct of the language? (With reduct I mean the new language that results after one character is matched.)
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 reducing 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 ⟧
```

[JR] How is this used in our example?

The first question is easy to answer: yes, the first disjunct of brackets is epsilon 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 reduct of brackets by using self-reference.

This enables us to write the reduct of brackets with respect to the opening bracket.

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

Conclusion:

- 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 reduction.
- Reducing a self-reference simply results in a self-reference again, because self-references in the reduct refer to the reduct.

Again, we do not want to have to do this reduction manually. 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:

$$\mathsf{parse} : \forall\ D \to \diamond.\mathsf{Parser}\ [\![\ D\ ]\!]$$

We approach this by decomposing parsing into ν and δ.

$$\begin{aligned}
&\nu\mathsf{D} : \forall\ D \to \mathsf{Dec}\ (\diamond.\nu\ [\![\ D\ ]\!]) \\
&\delta\mathsf{D} : \mathsf{Char} \to \mathsf{Desc} \to \mathsf{Desc}
\end{aligned}$$

The $\nu\mathsf{D}$ function can easily be written to be correct by construction, however $\delta\mathsf{D}$ must be proven correct separately as follows:

$$\delta\mathsf{D\text{-}correct} : [\![\ \delta\mathsf{D}\ c\ D\ ]\!]\ \diamond.\!\Longleftrightarrow \diamond.\delta\ c\ [\![\ D\ ]\!]$$

The actual parsing can now be done character by character:

$$\begin{aligned}
&\mathsf{parse}\ D\ [] = \nu\mathsf{D}\ D \\
&\mathsf{parse}\ D\ (c :: w) = \mathsf{Dec.map}\ \delta\mathsf{D\text{-}correct}\ (\mathsf{parse}\ (\delta\mathsf{D}\ c\ D)\ w)
\end{aligned}$$

That is the main result of this paper. The remainder of the paper concerns the implementation of $\nu\mathsf{D}$, $\delta\mathsf{D}$, $\delta\mathsf{D\text{-}correct}$.

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.

[JR] Reiterate that the cases for the basic combinators are the same as in Figure 2.

$$\begin{aligned}
&\nu_o : \mathsf{Dec}\ (\diamond.\nu\ P) \to \forall\ D \to \mathsf{Dec}\ (\diamond.\nu\ ([\![\ D\ ]\!]_o\ P)) \\
&\nu_o\ \nu P\ \emptyset && = \mathsf{no}\ \lambda\ () \\
&\nu_o\ \nu P\ \epsilon && = \mathsf{yes\ refl} \\
&\nu_o\ \nu P\ (`\ c) && = \mathsf{no}\ \lambda\ () \\
&\nu_o\ \nu P\ (D \cup D_1) && = \nu_o\ \nu P\ D\ \uplus\mathsf{\text{-}dec}\ \nu_o\ \nu P\ D_1 \\
&\nu_o\ \nu P\ (D * D_1) && = \mathsf{Dec.map}\ \diamond.\nu{*}\ (\nu_o\ \nu P\ D\ \times\mathsf{\text{-}dec}\ \nu_o\ \nu P\ D_1) \\
&\nu_o\ \nu P\ (x\ \cdot\ D) && = x\ \times\mathsf{\text{-}dec}\ \nu_o\ \nu P\ D \\
&\nu_o\ \nu P\ \mathsf{var} && = \nu P \\
&\nu_o\ \nu P\ (\mu\ D) && = \nu\mathsf{D}\ D
\end{aligned}$$

- Naively we might try $\nu\mathsf{D}\ D = \nu_o\ (\nu\mathsf{D}\ D)\ D$
- But that obviously will not terminate (consider the language $[\![\ \mathsf{var}\ ]\!]$).
- Instead we use Lemma 1

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

$$\nu\mathsf{D}\emptyset{\Leftrightarrow}\nu\mathsf{D} : \diamond.\nu\ (⟦\ D\ ⟧_o\ \diamond.\emptyset) \Leftrightarrow \diamond.\nu\ ⟦\ D\ ⟧$$

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.

$$\nu\mathsf{D}\emptyset{\rightarrow}\nu\mathsf{D} : \forall\ D \rightarrow \diamond.\nu\ (⟦\ D\ ⟧_o\ \diamond.\emptyset) \rightarrow \diamond.\nu\ (⟦\ D\ ⟧_o\ ⟦\ D_0\ ⟧)$$

The backwards direction is more difficult. We prove a more general lemma from which our disired 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.

$$\nu\mathsf{D}\emptyset{\leftarrow}\nu\mathsf{D} : \forall\ D \rightarrow \diamond.\nu\ (⟦\ D\ ⟧_o\ ⟦\ D_0\ ⟧) \rightarrow \diamond.\nu\ (⟦\ D\ ⟧_o\ \diamond.\emptyset) \uplus \diamond.\nu\ (⟦\ D_0\ ⟧_o\ \diamond.\emptyset)$$

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 $\mathsf{reduce} : A \uplus A \rightarrow A$ function. Modulo wrapping and unwrapping of the fixed point (using the roll constructor), we now have the two functions which prove the lemma:

$$\nu\mathsf{D}\emptyset{\Leftrightarrow}\nu\mathsf{D}\ \{D\} = \mathsf{mk}{\Leftrightarrow}\ (\mathsf{roll} \circ \nu\mathsf{D}\emptyset{\rightarrow}\nu\mathsf{D}\ D)\ (\mathsf{reduce} \circ \nu\mathsf{D}\emptyset{\leftarrow}\nu\mathsf{D}\ \{D_0 = D\}\ D \circ \mathsf{unroll})$$

Using Lemma 1, we can easily define nullability for our description functors.

$$\nu\mathsf{D} = \mathsf{Dec.map}\ \nu\mathsf{D}\emptyset{\Leftrightarrow}\nu\mathsf{D} \circ \nu_o\ (\mathsf{no}\ \lambda\ ())$$

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, $\mathsf{var} * \mathsf{var}$, and then the first disjunct ϵ for both branches.

$$\begin{aligned}&\mathsf{brackets}_0 : \diamond.\nu\ \mathsf{brackets}\\&\mathsf{brackets}_0 = \mathsf{roll}\ (\mathsf{inj}_2\ (\mathsf{inj}_2\ ([]\ ,\ []\ ,\ \mathsf{refl}\ ,\ \mathsf{roll}\ (\mathsf{inj}_1\ \mathsf{refl})\ ,\ \mathsf{roll}\ (\mathsf{inj}_1\ \mathsf{refl}))))\end{aligned}$$

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

$$\begin{aligned}&\mathsf{brackets}_0\text{'} : \nu\mathsf{D}\emptyset{\Leftrightarrow}\nu\mathsf{D}\ \{\mathsf{bracketsD}\}\ .\mathsf{to}\ (\nu\mathsf{D}\emptyset{\Leftrightarrow}\nu\mathsf{D}\ \{\mathsf{bracketsD}\}\ .\mathsf{from}\ \mathsf{brackets}_0)\\&\qquad\quad \equiv \mathsf{roll}\ (\mathsf{inj}_1\ \mathsf{refl})\\&\mathsf{brackets}_0\text{'} = \mathsf{refl}\end{aligned}$$

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.

[JR] todo: give recommendations for future work, for example to use data-dependent grammars.

3.5 Reduction

The final piece of the puzzle is reduction. This tells 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.

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

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.

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

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

$$
\begin{aligned}
&\delta\mathsf{D\text{-}to} : \forall\ D \to [\![\ \delta_o\ D_0\ c\ D\ ]\!]_o\ [\![\ \delta\mathsf{D}\ c\ D_0\ ]\!]\ w \to \diamond.\delta\ c\ ([\![\ D\ ]\!]_o\ [\![\ D_0\ ]\!])\ w\\
&\delta\mathsf{D\text{-}to}\ (\text{‘}\ c') = \diamond.\delta\text{‘}\ .\mathsf{to}\\
&\delta\mathsf{D\text{-}to}\ (D \cup D_1)\ (\mathsf{inj}_1\ x) = \mathsf{inj}_1\ (\delta\mathsf{D\text{-}to}\ D\ x)\\
&\delta\mathsf{D\text{-}to}\ (D \cup D_1)\ (\mathsf{inj}_2\ y) = \mathsf{inj}_2\ (\delta\mathsf{D\text{-}to}\ D_1\ y)\\
&\delta\mathsf{D\text{-}to}\ (D * D_1)\ (\mathsf{inj}_1\ (x\ ,\ y)) = []\ ,\ \_\ ,\ \mathsf{refl}\ ,\ x\ ,\ \delta\mathsf{D\text{-}to}\ D_1\ y\\
&\delta\mathsf{D\text{-}to}\ (D * D_1)\ (\mathsf{inj}_2\ (u\ ,\ v\ ,\ \mathsf{refl}\ ,\ x\ ,\ y)) = (\_ :: \_)\ ,\ \_\ ,\ \mathsf{refl}\ ,\ \delta\mathsf{D\text{-}to}\ D\ x\ ,\ \mathsf{subst}\ \mathsf{id}\ (\sigma\mu\ D_1)\ y\\
&\delta\mathsf{D\text{-}to}\ (A\ \cdot\ D)\ (x\ ,\ y) = x\ ,\ \delta\mathsf{D\text{-}to}\ D\ y\\
&\delta\mathsf{D\text{-}to}\ \{D_0 = D\}\ \mathsf{var}\ (\mathsf{roll}\ x) = \mathsf{roll}\ (\delta\mathsf{D\text{-}to}\ D\ x)\\
&\delta\mathsf{D\text{-}to}\ (\mu\ D)\ (\mathsf{roll}\ x) = \mathsf{roll}\ (\delta\mathsf{D\text{-}to}\ D\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\delta\mathsf{D\text{-}from} : \forall\ D \to \diamond.\delta\ c\ ([\![\ D\ ]\!]_o\ [\![\ D_0\ ]\!])\ w \to [\![\ \delta_o\ D_0\ c\ D\ ]\!]_o\ [\![\ \delta\mathsf{D}\ c\ D_0\ ]\!]\ w\\
&\delta\mathsf{D\text{-}from}\ (\text{‘}\ c') = \diamond.\delta\text{‘}\ .\mathsf{from}\\
&\delta\mathsf{D\text{-}from}\ (D \cup D_1)\ (\mathsf{inj}_1\ x) = \mathsf{inj}_1\ (\delta\mathsf{D\text{-}from}\ D\ x)\\
&\delta\mathsf{D\text{-}from}\ (D \cup D_1)\ (\mathsf{inj}_2\ y) = \mathsf{inj}_2\ (\delta\mathsf{D\text{-}from}\ D_1\ y)\\
&\delta\mathsf{D\text{-}from}\ (D * D_1)\ ([]\ ,\ w\ ,\ \mathsf{refl}\ ,\ x\ ,\ y) = \mathsf{inj}_1\ (x\ ,\ \delta\mathsf{D\text{-}from}\ D_1\ y)\\
&\delta\mathsf{D\text{-}from}\ (D * D_1)\ (c :: u\ ,\ v\ ,\ \mathsf{refl}\ ,\ x\ ,\ y) = \mathsf{inj}_2\ (u\ ,\ v\ ,\ \mathsf{refl}\ ,\ \delta\mathsf{D\text{-}from}\ D\ x\ ,\ \mathsf{subst}\ \mathsf{id}\ (\mathsf{sym}\ (\sigma\mu\\
&\delta\mathsf{D\text{-}from}\ (A\ \cdot\ D)\ (x\ ,\ y) = x\ ,\ \delta\mathsf{D\text{-}from}\ D\ y\\
&\delta\mathsf{D\text{-}from}\ \{D_0 = D\}\ \mathsf{var}\ (\mathsf{roll}\ x) = \mathsf{roll}\ (\delta\mathsf{D\text{-}from}\ D\ x)\\
&\delta\mathsf{D\text{-}from}\ (\mu\ D)\ (\mathsf{roll}\ x) = \mathsf{roll}\ (\delta\mathsf{D\text{-}from}\ D\ x)
\end{aligned}
$$

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

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. .

[JR] mention that we only support context-free languages without mutual recursion and how we use a subset of μ-regular 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. 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. [3] 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

Simplicity One of the main contributions of Elliott's type theoretic formalization of languages [2] 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.

[JR] TODO: finish this paragraph

5 Related Work

- Jeremy Yallop - performance
- Peter Thiemann - derivatives of μ-regular expressions. This is the closest to our work, we have a mechanized proof and use type theory instead of set theory.
- Guillome Allais' Agdarsec
- Danielsson's coinductive parser combinators
- ”Certified Parsing of Dependent Regular Grammars” John Sarracino; Gang Tan; Greg Morrisett
- Brink et al. (MPC 2010), they formalize the left-corner transformation
- Jean-Philippe Bernardy and Patrik Jansson, ”Certified Context-Free Parsing: A formalisation of Valiant's Algorithm in Agda.” This is a formalization of a performant matrix-based parsing algorithm.
- Conal Elliott, of course
- Introduction to Automata Theory, Languages, and Computation - Hopcroft, Motswani, Ullman

6 Conclusion

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. Elliott, C.: Symbolic and automatic differentiation of languages. Proc. ACM Program. Lang. **5**(ICFP) (Aug 2021). https://doi.org/10.1145/3473583
3. 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