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 Section 3.2. These are the only postulates that we have yet to prove.

2 Languages

In this section, we summarize the basic definitions from previous work by Elliot [3]. We leave out details and proofs in some places and refer the interested reader to his work.

2.1 Languages

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

```
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 λ 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 ('a :: 'a :: 'b :: 'b :: 'c :: 'c :: [])
aabbcc = suc (suc zero) , 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.

2.2 Nullability and Derivatives

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 : Lang → Char → Lang
δ ℒ c = 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
```

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

2.3 Decidability

From our type theoretic perspective, parsing a string is the same thing as producing an element of the result type of a language for that given input string, or showing that no such element can exist. In Agda, we encode this using the following Dec data type which is parameterized by a type A and contains a constructor yes for when you can produce an element of A or no if you can show that no such element exists.

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

Sometimes we want to change the parameter type of a Dec. For that we need to provide conversion functions between the old and the new type in both ways.

```
map? : (A ↔ B) → Dec A → Dec B
map? f (yes x) = yes (to f x)
map? f (no ¬A) = no λ x → ¬A (from f x)
```

2.4 Grammars and Parsing

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 Gram : Lang → Type₁ where
  ∅     :                          Gram (λ _ → ⊥)
  ϵ     :                          Gram (λ w → w ≡ [])
  char  : (c : Char)               → Gram (λ w → w ≡ c :: [])
  _·_   : Dec A → Gram ℒ           → Gram (λ w → A × ℒ w)
  _∪_   : Gram ℒ₁ → Gram ℒ₂ → Gram (λ w → ℒ₁ w ⊎ ℒ₂ w)
  _*_   : Gram ℒ₁ → Gram ℒ₂
        → Gram (λ w → Σ String λ u → Σ String λ v → (w ≡ u ++ v) × ℒ₁ u × ℒ₂ v)
  _◁_   : (ℒ₁ ⇔ ℒ₂) → Gram ℒ₁ → Gram ℒ₂
```

Remark 2. The Gram data type is parameterized by its language. This ties the constructors directly to their semantics.

By recursion over this data type of grammars, we can define a decision procedure for nullability and derivative function; both are correct by construction.

```
ν? : Gram ℒ → Dec (ν ℒ)
δ? : Gram ℒ → (c : Char) → Gram (δ ℒ c)
```

Together, decidable nullability and the derivative function can be combined to decide whether any string is in the language described by a grammar.

```
parse : Gram ℒ → (w : String) → Dec (ℒ w)
parse G [] = ν? G
parse G (c :: w) = parse (δ? G c) w
```

Thus, we have defined a parser for our simple grammars.

3 Context-free Languages

Regular languages can be useful for describing patterns in program text, but they are not sufficient to model the full language of a programming language. For example, balanced brackets are a common syntactic element in programming languages.

Example 2. We can boil the problem down to the following language which consists only of balanced brackets:

```
brackets_k : ℕ → Lang
brackets_k zero _ = ⊥
brackets_k (suc k) w = (w ≡ [])
                     ⊎ (∃[ u ] (w ≡ '[ :: [] ++ u ++ '] :: []) × brackets_k k u)
                     ⊎ (∃[ u ] ∃[ v ] (w ≡ u ++ v) × brackets_k k u × brackets_k k v)


brackets : Lang
brackets w = ∃[ k ] brackets_k k w
```

Remark 3. The $\mathsf{brackets}_k$ function is truncated after k recursive calls to ensure termination, which is required for all functions in type theory. The proper language brackets asserts that, for a string to be in the language, there must exist a k which is large enough that the truncation becomes irrelevant for that particular string.

3.1 Context-free Grammars

This language of balanced brackets is famously context-free. To support languages such as these we add variables, var, and fixed points, μ, to our grammars.

```
data Gram (n : ℕ) : Set₁ where
  ∅ ϵ : Gram n
  char : Char → Gram n
  _ · _ : Dec A → Gram n → Gram n
  _∪_ _*_ : Gram n → Gram n → Gram n
  var : Fin n → Gram n
  μ : Gram (suc n) → Gram n


⟦_⟧_k : Gram n → (Fin n → Lang) → ℕ → Lang
```

```
⟦ var i ⟧ₖ Γ k w = Γ i w
⟦ μ G ⟧ₖ Γ zero w = ⊥
⟦ μ G ⟧ₖ Γ (suc k) w = ⟦ G ⟧ₖ (⟦ μ G ⟧ₖ Γ k ::> Γ) k w


⟦_⟧ : Gram n → (Fin n → Lang) → Lang
⟦ G ⟧ Γ w = ∃[ k ] ⟦ G ⟧ₖ Γ k w
```

Example 3. This allows us to write a grammar for the language of balanced brackets.

```
bracketsG : Gram n
bracketsG = μ (ϵ ∪ char '[ ∗ var zero ∗ char '] ∪ var zero ∗ var zero)
```

Lemma 1. *We can map over context and the fuel of the truncated semantics.*

```
mapΓ : (G : Gram n) (Γ Γ' : Fin n → Lang)
      → ((i : Fin n) → {w : String} → Γ i w → Γ' i w)
      → ⟦ G ⟧ₖ Γ k w → ⟦ G ⟧ₖ Γ' k w


mapk : k ≤ k' → ⟦ G ⟧ₖ Γ k w → ⟦ G ⟧ₖ Γ k' w
```

Lemma 2. *We can map a change of variables over a grammar and we can substitute variables. This essentially shows that grammars form a relative monad.*

```
rename : (Fin n → Fin m) → Gram n → Gram m


subst : Gram n → (Fin n → Gram m) → Gram m
```

3.2 Parsing

Parsing our context-free grammar follows the same structure as the simple grammars from Section 2.4. Concretely, we define functions that compute the nullability, $\nu?$, and derivatives, $\delta?$. For this section we have taken inspiration from a blog post by Grenrus [4].

Example 4. Let us consider the balanced bracket grammar example. We can see that it is nullable because it contains an ϵ in the fixed point. It is also possible to parse the empty string by taking one iteration of the fixed point using the var zero ∗ var zero part and then the ϵ for both recursive calls, but note that we always need to end in an empty base case. Thus, for a fixed point to be nullable, it must be nullable even if we do not consider the recursive calls.

The derivative of the balanced bracket grammar can be taken with respect to any character, but only the character '[results in anything interesting because any string in the balanced bracket language needs to start with an opening bracket. The first thing we might try is to unroll the fixed point one step, yielding the following grammar:

```
bracketsG₁ : Gram n
bracketsG₁ = ϵ ∪ char '[ ∗ bracketsG ∗ char '] ∪ bracketsG ∗ bracketsG
```

We know how to take the derivative of the first two parts, but bracketsG ∗ bracketsG seems problematic because its derivative depends on the derivative of bracketsG itself. Luckily, we can introduce a new fixed point when describing the derivative to refer to the derivative itself.

```
bracketsG' : Gram n
bracketsG' = μ (bracketsG ∗ char '] ∪ var zero ∗ bracketsG)
```

Nullability Computing the nullability now requires us to deal with grammars that contain free variables, but we can make use of a context $\Gamma\nu$ which tells us how to compute the nullability of those variables.

```
ν? : (G : Gram n) (Γν : (i : Fin n) → Dec (ν (Γ i))) → Dec (ν (⟦ G ⟧ Γ))
```

The simple cases remain the same except that $\Gamma\nu$ now has to be passed properly to recursive calls. We skip to the two new cases: variables and fixed points. For both cases we need a helper. In the case of variables this helper just deals with converting between the truncated semantics and the proper semantics.

```
νΓi↔ν⟦vari⟧Γ : ν (Γ i) ↔ ν (⟦ var i ⟧ Γ)
to νΓi↔ν⟦vari⟧Γ x = zero , x
from νΓi↔ν⟦vari⟧Γ (_ , x) = x
```

For the fixed point, we need to formalize the intuition from Example 4. Recall that we noted how determining the nullability of a fixed point only requires unrolling it once and no more.

```
νG⊥↔νμG : ν (⟦ G ⟧ ((λ _ → ⊥) ::> Γ)) ↔ ν (⟦ μ G ⟧ Γ)
```

We are still working on a proof of this property, but we have been able to reduce it to the following postulate which states that, if a grammar with free variables is nullable, either the nullability is independent of that variable, or that variable itself needs to be nullable.

```
postulate νGℒ→νG⊥⊎νℒ : ν (⟦ G ⟧ₖ (ℒ ::> Γ) k) → ν (⟦ G ⟧ₖ ((λ _ → ⊥) ::> Γ) k) ⊎ ν ℒ
```

Using these two helpers, we can define the nullability of variables and fixed points as follows:

```
ν? {Γ = Γ} (var i) Γν = map? (νΓi↔ν⟦vari⟧Γ {Γ = Γ}) (Γν i)
ν? (μ G) Γν = map? νG⊥↔νμG (ν? G (no (λ ()) ::> Γν))
```

Derivatives Computing the derivative also requires us to deal with free variables in our grammar. For derivatives, we need to keep track of four different environments:

1. The language environment, Γ, which contains the language of each variable.
2. The nullability environment, $\Gamma\nu$, which tells us the nullability of all variables.
3. The derivative environment, $\Gamma\delta$, which keeps track of the derivative of each variable.

4. The unrolling environment, $\Gamma\sigma$, which allows us to replace each variable by the fixed point that bound it, thus unrolling the fixed point.

The Gram data type is no longer parameterized by its semantics, so we first define a syntactic derivative function $\delta?$ and afterwards prove that it corresponds to the semantic derivative.

```
δ? : (Γ : Fin n → Lang) (Γν : (i : Fin n) → Dec (ν (Γ i))) (Γδ : Fin n → Gram m)
     (Γσ : Fin n → Gram m)
   → Gram n → Char → Gram m
```

Again, all simple cases are the same except for passing around the environments correctly to recursive calls, so we skip to the two new cases for variables and fixed points. For variables, we simply look up their derivative in the derivative environment. For fixed points, we need to show how to extend each of the four environments. Here we apply the same trick as we discovered in Example 4, namely that we introduce a new fixed point which allows us to refer to the derivative itself.

```
δ? _ _ Γδ _ (var i) _ = Γδ i
δ? Γ Γν Γδ Γσ (μ G) c =
  μ (δ? (⟦ μ G ⟧ Γ                        ::> Γ)
        (ν? {Γ = Γ} (μ G) Γν               ::> Γν)
        (var zero                          ::> (rename suc ∘ Γδ))
        (subst (μ G) (rename suc ∘ Γσ) ::> (rename suc ∘ Γσ))
        G c)
```

We show the correctness of the syntactic derivative by showing that every string accepted by the result of taking the syntactic derivative of a grammar is also accepted by the semantic derivative of the original grammar and vice versa. The last two arguments specify that the unrolling and derivative environment actually contain what they are supposed to contain.

```
δ?↔δ : (G : Gram n) {Γ : Fin n → Lang} {Γ' : Fin m → Lang}
       {Γν : (i : Fin n) → Dec (ν (Γ i))} {Γδ : Fin n → Gram m} {Γσ : Fin n → Gram m}
     → ((i : Fin n) → ⟦ Γσ i ⟧ Γ' ⇔ Γ i)
     → ((i : Fin n) → ⟦ Γδ i ⟧ Γ' ⇔ δ (Γ i) c)
     → ⟦ δ? Γ Γν Γδ Γσ G c ⟧ Γ' ⇔ δ (⟦ G ⟧ Γ) c
```

We are still working on proofs for two parts of this correspondence. First, if a substitution corresponds pointwise to a change of environment, substituting all variables in a grammar also corresponds to a change of environment.

```
postulate substΓσ : {Γσ : Fin n → Gram m} (G : Gram n)
                  → ((i : Fin n) → ⟦ Γσ i ⟧ Γ' ⇔ Γ i) → ⟦ subst G Γσ ⟧ Γ' ⇔ ⟦ G ⟧ Γ
```

Second, we are still working on proving the correctness of the syntactic derivative of fixed points.

```
postulate
  δ?↔δμ : (G : Gram (suc n)) {Γ : Fin n → Lang} {Γ' : Fin m → Lang}
          {Γν : (i : Fin n) → Dec (ν (Γ i))} {Γδ : Fin n → Gram m} {Γσ : Fin n → Gram m}
        → ((i : Fin n) → ⟦ Γσ i ⟧ Γ' ⇔ Γ i)
```

```
  → ((i : Fin n) → ⟦ Γδ i ⟧ Γ' ⇔ δ (Γ i) c)
  → ⟦ δ? Γ Γν Γδ Γσ (μ G) c ⟧ Γ' ⇔ δ (⟦ μ G ⟧ Γ) c
```

With the exception of these two postulates, we have proven the correctness of our syntactic derivative function.

Parsing Tying it all together, we show how to parse a string following a grammar. We only care about grammars without variables, so all the environments are empty (λ ()).

```
parse : (G : Gram zero) → (w : String) → Dec (⟦ G ⟧ (λ ()) w)
parse G [] = ν? G (λ ())
parse G (c :: cs) = map? (δ?↔δ G (λ ()) (λ ())) (parse (δ? (λ ()) (λ ()) (λ ()) (λ ()) G c) cs)
```

This is a correct parser for context-free grammars.

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