Context-free languages: generalizing & characterizing

Keny Chatain

March 15, 2020

In this note, I propose to characterize a constructive characterization of the class of context-free languages. The goal is to have a characterization similar to the characteriation of the class of rational languages, as the class of smallest languages closed under some natural operations.

The formal details are interspersed with a Haskell implementation.

1 The word setting

1.1 Introduction

Here, we assume an alphabet Σ (Char in the Haskell implementation) and the set of strings on that alphabet, Σ^* (String).

A language is a collection of strings. The use of the word "*collection*", instead of "*set*", is meant to reflect the fact that not all sets operations will be available to us. A collection, only makes available some operations like union and singleton set and certain form of set comprehension (the latter two are properties of Applicative):

```
instance Collection [] where
        union = (++)


type Language = forall f . Collection f => f String
```

Using an applicative allows us to "*lift*" any operation and any element defined on words into the realm of languages. To an element, corresponds the singleton language of that element. To word concatenation, corresponds language concatenation:

```
cat :: Language -> Language -> Language
cat l1 l2 = ((pure (++)) <*> l1) <*> l2
```

So languages (i.e. collection of words) have two types of operations defined on them, those that can be imported from the underlying structure of words (concatenate, elements) and those that follow from the structure of a collection (e.g. union). With this, one may define some finite languages:

```
-- language1 : ab + ba
language1 :: Language
language1 = ((pure "a") `cat` (pure "b")) `union` ((pure "b") `cat` (pure
      "a"))

-- language2 : ab(a+b)
language2 :: Language
language2 = (pure "a") `cat` (pure "b") `cat` ((pure "a") `union` (pure "
     b"))
```

In fact, taking advantage of recursivity/lazy evaluation, we can also define infinite languages as well:

```
-- language3: a*
language3 :: Language
language3 = (pure "") `union` ((pure "a") `cat` language3)
```

Kleene star can be obtained through a form of recursivity as well:

```
-- Kleene start \L -> L*
star :: Language -> Language
star l = (pure "") `union` (l `cat` (star l))
```

So the class of languages we can define is also closed under union, concatenation and Kleene star and contains all finite sets. A minima then, we can define any regular language:

```
-- language4: a*b*
language4 :: Language
language4 = (star $ pure "a") `cat` (star $ pure "b")
```

But recursivity allows us to define more languages than just that. For instance, the famed non-regular $\{a^n b^n \mid n \geq 0\}$:

```
-- language5: {a^nb^n | n >= 0}
language5 :: Language
language5 = pure "" `union` (pure "a" `cat` language5 `cat` pure "b")
```

In fact, we can *build* a context-free grammar, by making use of crossed recursivity:

```
{-
Grammar:
TUPLE -> ( VALUES )
VALUES -> VALUES , TUPLE
VALUES -> empty string
-}
tuple :: Language
values :: Language
tuple = (pure "(") `cat` values `cat` (pure ")")
values = (pure "") `union` (values `cat` pure "," `cat` tuple)
```

So all context-free languages can be defined in Haskell. Can we do even more than that? Interestingly no. The class of languages definable in Haskell, using only the operations made available by collections, words and recursivity, is the class of context-free languages. In some sense, context-free languages is the smallest class of languages where one can make use of the full power of recursivity

In the next section, I set out to prove that result in its formal details

1.2 Formalization

Before we do so, the notion of recursivity needs to be formalized. Informally, recursivity is a way of generating fixed points.

```
-- language3prime: a*
rec_l3 :: Language -> Language
rec_l3 l = (pure "") `union` ((pure "a") `cat` l)
language3prime :: Language
language3prime = fix rec
```

Of course, in Haskell, these is no guarantee of convergence, hence no guarantee that there is a fixed-point to any function. However, in the limited realm of functions that we consider, such guarantees are possible:

Proposition 1. *If f is an* increasing *function from languages to languages (i.e. $L \subset L' \Rightarrow f(L) \subset f(L')$), then*

$$\mathbf{FixPt}(f) = \bigcap \left\{ L \mid f(L) \subset L \right\}$$

is a fixed point. It is the smallest fixed point in fact (for subsethood).

Coincidentally, all the functions we can define with union and concatenation are increasing. Had we been working with sets and allowed such operations as complementation, we could construct non-monotonic function and the fixed-point guarantee would have vanished. A further thing to note is that in some high-order sense, **FixPt** is increasing

Proposition 2. *If f and f' are two increasing functions, and f' dominates f (i.e. $f(L) \subset f'(L)$ for all L), then*

$$\mathbf{FixPt}(f) \subset \mathbf{FixPt}(f')$$

So all the operations at our disposal - union, concatenation, fixed-point - will only ever generate increasing functions. Hence, any function we might build from these primitives has a least fixed point **FixPt**. In other words, we can apply **FixPt** multiple times to an input

1.3

We are aiming for a characterization of context-free languages in terms of some primitive elements and operations, the same way regular languages are defined in terms of singletons and rational operations. The difference is that here, our primitive operations include **FixPt**, which operates on functions. Therefore, our expressions will need to model both languages and function on languages at the same time.

Definition 1. *A context-free expression is:*

- *a word (e.g. a, $abba$) (type* Language*)*
- *a variable over languages (type* Language*)*
- *the union of two context-free expressions of type* Language*, denoted $L+L'$*
- *the concatenation of two context-free expressions of type* Language*, denoted LL'*
- *the least fixed-point of a context-free function (type* Language→Language*), denoted $\mathbf{Fix}(L)$*
- *if E is a context-free expression of type a, $\lambda X.E$ is an expression of type* Language $\to$ *a*
- *if E is a context-free expression of type $a \to b$ and E' an expression of type a, $E(E')$ is an expression of type b*

We can provide a straightforward semantics for these expressions

Definition 2. *The object denoted by a context-free expression S*

- $[\![w]\!]^g = \{w\}$
- $[\![X]\!]^g = g(X)$
- $[\![L+L']\!]^g = [\![L]\!]^g \cup [\![L']\!]^g$
- $[\![LL']\!]^g = [\![L]\!]^g\,[\![L']\!]^g$
- $[\![\lambda X.\ E]\!]^g = L \mapsto [\![E]\!]^{g[X \leftarrow L]}$
- $[\![E(E')]\!]^g = [\![E]\!]^g\left([\![E']\!]^g\right)$
- $[\![\mathbf{Fix}(\mathbf{E})]\!]^g = \mathbf{FixPt}([\![\mathbf{E}]\!]^g)$

Proofs

Proof. Because $\mathbf{FixPt}(f)$ is an intersection:

$$\forall L,\ f(L) \subset L \Rightarrow \mathbf{FixPt}(f) \subset L \tag{1}$$

Because f is increasing, we conclude:

$$\forall L,\ f(L) \subset L \Rightarrow f\left(\mathbf{FixPt}(f)\right) \subset f(L)$$

Because L is a subset of $f(L)$:

$$\forall L,\ f(L) \subset L \Rightarrow f\left(\mathbf{FixPt}(f)\right) \subset L$$

So, by intersection:

$$f\left(\mathbf{FixPt}(f)\right) \subset \bigcap \left\{L \mid f(L) \subset L\right\} = \mathbf{FixPt}(f)$$

This proves one inclusion. To prove the reverse inclusion, we notice that because f is increasing, we also have

$$f(f(\mathbf{FixPt}(f))) \subset f(\mathbf{FixPt}(f))$$

Plugging in this result in eqn 1, we get:

$$\mathbf{FixPt}(f) \subset f(\mathbf{FixPt}(f))$$

So by double inclusion, these two languages are equal. Because all fixed points L are such that $f(L) \subset L$ and $\mathbf{FixPt}(f)$ is the smallest of these languages, $\mathbf{FixPt}(f)$ is the smallest fixed-point □

Proof. Because f' dominates f:

$$f(\mathbf{FixPt}(f')) \subset f'(\mathbf{FixPt}(f')) = \mathbf{FixPt}(f')$$

By definition of $\mathbf{FixPt}(f)$, this entails that

$$\mathbf{FixPt}(f) \subset \mathbf{FixPt}(f')$$

□