Context-free languages: generalizing & characterizing

Keny Chatain

March 29, 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):

```
class Applicative f => Collection f where
        union :: f a -> f a -> f a


instance Collection [] where
        union = (++)
-- use "show $ take 10 l" to display elements from any given language "l"
-- TODO: in practice, we'd be better off with something that interspersed
     elements from both lists; otherwise, listing elements form the
     language is not exhaustive
type ContextFree = 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 :: ContextFree -> ContextFree -> ContextFree
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 :: ContextFree
language1 = ((pure "a") `cat` (pure "b")) `union` ((pure "b") `cat` (pure
      "a"))

-- language2 : ab(a+b)
language2 :: ContextFree
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 :: ContextFree
language3 = (pure "") `union` ((pure "a") `cat` language3)
```

More generally, Kleene star can be defined through a form of recursivity as well:

```
-- Kleene start \L -> L*
star :: ContextFree -> ContextFree
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 :: ContextFree
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 :: ContextFree
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 :: ContextFree
values :: ContextFree
```

```
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 closed under union and concatenation and recursive functions made from them.

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.

```
-- defining recursivity in terms of fixed point
fix :: (ContextFree -> ContextFree) -> ContextFree
fix f = f $ fix f

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

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*

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

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, instead of collections, 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- will only ever generate increasing functions. Any function formed from **FixPt** will also be increasing and **FixedPt** can apply to it. So it makes sense to talk about the smallest set containing finite languages closed under these 3 operations: union, concatenation, and fixed point. The next section makes that precise.

1.3 Context-free expressions

We are aiming for a characterization of context-free languages in terms of some primitive elements and operations, the same way rational languages are defined as the smallest class of languages containing singletons and closed under rational operations (union, concatenation, Kleene star). The difference is that here, our primitive operations include **FixPt**, which operates on functions. Therefore, our expressions will need to model both languages and functions on languages at the same time.

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

- *a word (e.g. a, $abba$) (type ContextFree)*
- *a variable over languages (type ContextFree)*
- *the union of two context-free expressions of type ContextFree, denoted $L + L'$*
- *the concatenation of two context-free expressions of type ContextFree, denoted LL'*
- *the least fixed-point of a context-free function (type ContextFree→ContextFree), denoted* $\mathbf{Fix}(L)$
- *if E is a context-free expression of type a, $\lambda X.E$ is an expression of type ContextFree → 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[1]:

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

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

Because of the type system and the facts about increasingness discussed in the last subsection, all of these operations are going to be well-defined.

[1]Because of variables, I have to define rational expressions syntactically and then provide a semantics for these expressions. To avoid this extra step, we could use variable-free representations using combinators from e.g. combinatory logic. However, what we gain in conceptual transparency, we lose in notational transparency: the simplest function require a lot of combinators to write.

Original grammar	Factorizing left-hand sides	Convert to a system of equation
$T \to aVb$	$T \to aVb$	$L_T = aL_Vb$
$V \to \epsilon$	$V \to \epsilon + TcV$	$L_V = \epsilon + L_TcL_V$
$V \to TcV$		

1.4 Equivalence to context-free languages

The context-free languages are those languages denoted by a context-free expression. To prove this, two steps are needed. We need to be able to convert from a context-free grammar to context-free expression and vice-versa.

Obtaining a context-free grammar from a context-free expression. To each expression E, we associate a set of context-free rewrite rules **Rules(E)** and a distinguished start symbol **Start(E)** as below. I assume that to all language variables correspond a non-terminal and S_i will stand for any nonterminal that hasn't be used so far.

- $\textbf{Rules}[w] = [S \to w]$
  $\textbf{Start}[w] = S$
- $\textbf{Rules}[X] = []$
  $\textbf{Start}[X] = X$
- $\textbf{Rules}[EE'] = [S \to \textbf{Start}(E)\textbf{Start}(E')]$
  $\textbf{Start}[EE'] = S$
- $\textbf{Rules}[E+E'] = [S \to \textbf{Start}(E), S \to \textbf{Start}(E')]$
  $\textbf{Start}[E+E'] = S$
- $\textbf{Rules}[\lambda X.E] = \textbf{Rules}(E)$
  $\textbf{Start}[\lambda X.E] = \textbf{Start}(E)$
- $\textbf{Rules}[(\lambda X.E)E'] = \textbf{Rules}(E) +\!\!+ \textbf{Rules}(E') +\!\!+ [X \to \textbf{Start}(E')]$
  $\textbf{Start}[(\lambda X.E)E'] = \textbf{Start}(E)$
- $\textbf{Rules}[\textbf{Fix}(\lambda X.E)] = \textbf{Rules}(E) +\!\!+ [X \to \textbf{Start}(E)]$
  $\textbf{Start}[\textbf{Fix}(\lambda X.E)] = X$

Obtaining a context-free expression from a context-free grammar. The following is extremely reminiscent of the method used to resolve system of language equations for regular expressions (cf proof of Kleene's theorem). The first step is to rewrite a context free grammar as a system of equations over languages. I will show how this is done on a simple example ; hopefully, this example will make a more general statement superfluous. To each terminal X corresponds the unknown L_X

There is of course a relation between the solution of this system of equations and the context-free language itself:

Proposition 3. *The system of equations obtained from a context-free grammar has a smallest solution (for inclusion). This smallest solution is the tuple* $(L_{X_1} = L(X_1), \ldots, L_{X_n} = L(X_n))$ *where* $L(X_i)$ *is the language generated by non-terminal* X_i

So in our example:, we can in particular write

$$\begin{array}{rcl} L(T) & = & aL(V)b \\ L(V) & = & \epsilon + L(T)cL(V) \end{array}$$

Each of these equation can be seen as stating that a certain $L(X)$ is the fixed point of a function[2]:

$$L(V) = (\lambda X.\, \epsilon + L(T)cX)\,(L(V))$$

Two things: first, each $L(X)$ has to be the smallest such fixed point. Otherwise, we could form a non-bigger solution by replacing $L(X)$ with a non-bigger fixed point in the solution tuple. Second, the function is increasing that each $L(X)$ is the fixed point of is increasing. In other words:

$$L(V) = \mathbf{Fix}\,(\lambda X.\, \epsilon + L(T)cX)$$

These remarks allow us to reduce any of the $L(X)$ into a context-free expression. The idea is the same as with any system of equations: express one unknown in terms of the other, replace the unknown with its expression in terms of the other in all the other equations, repeat until all unknowns are eliminated. So in our particular case, we can replace $L(V)$ with its expression above in the first equation of the system

$$L(T) = a\mathbf{Fix}\,(\lambda X.\, \epsilon + L(T)cX)\, b$$

This can be taken to mean that $L(T)$ is the fixed point of the function:

$$L(T) = (\lambda Y.\, a\mathbf{Fix}\,(\lambda X.\, \epsilon + YcX)\, b)\,(L(T))$$

With a similar reasoning as above, we can conclude that:

$$L(T) = \mathbf{Fix}\,(\lambda Y.\, a\mathbf{Fix}\,(\lambda X.\, \epsilon + YcX)\, b)$$

If T is the start symbol of the grammar, then we just found that the context-free language can be expressed by a context-free expression. This concludes the proof that every context-free language is expressible in terms of these expressions

[2]Note that I am temporarily suspending the distinction between context-free expressions and the language they denote. This distinction is a technical hassle.

1.5 Missing pieces

There is one aspect of the Haskellization which is regrettable. In Haskell, there is a way to write context-sensitive languages if we allow ourselves to lift more complex string-based operations than the primitive operation of word, concatenation. For instance, this implementation of $\{a^n b a^n b a^n \mid n \geq 0\}$:

```
-- context-sensitive languages, if we allow "non-primitive" lifted
    operations
-- language6 : a^nba^nba^n
language6 :: ContextFree
language6 = fmap (\x -> x ++ "b" ++ x ++ "b" ++ x) (star $ pure "a")
```

Added to that, it can be argued that the Haskell fragment is more expressive than context-free expressions. For instance, Haskell can express functions of type (ContextFree → ContextFree) → ContextFree → ContextFree, while context-free expressions do not. If such functions are available to Haskell, can Haskell express more languages than context-free expressions can?

Formally, this corresponds to asking whether we would obtain non-context-free languages if we allowed higher-order fixed-points, i.e. fixed points of (ContextFree → ContextFree) → ContextFree → ContextFree. We would need a theorem giving us the existence of a privileged fixed point for such expressions. There are such guarantees

Contrary to my own expectations, it turns out that context-sensitive languages can be expressed using higher-order fixed-points:

```
-- context-sensitive language with higher-order fixed-point
-- generator is the least fixed point of \F \L LLL + F(aL)
generator :: ContextFree -> ContextFree
generator l = (l `cat` l `cat` l) `union` (generator $ (pure "a") `cat` l
    )
-- language7 : a^nba^nba^nb
language7 :: ContextFree
language7 = generator $ pure "b"
```

2 Extensions

2.1 Monoids

In essence, all the work we have been doing did not depend, in any essential manner, on taking words to be the underlying monoid. In fact, we can define context-free languages in any monoid:

```
class Applicative f => Collection f where
        union :: f a -> f a -> f a

type ContextFree a = forall f . Collection f => f a
```

Kleene star and concatenation is definable in the same way:

```
-- concatenation
cat :: (Monoid a, Collection f) => f a -> f a -> f a
cat l1 l2 = (pure mappend) <*> l1 <*> l2

-- Kleene star
star :: Monoid a => ContextFree a -> ContextFree a
star l =  (pure mempty) `union` (l `cat` (star l))
```

This is useful in defining for instance context-free relations, the equivalent of rational relations/finite-state transducers. Here is for instance the duplicating function on strings:

```
type ContextFreeRelation = ContextFree (String, String)

-- establishes a relation between w and ww for any w made of a's and b's
duplicate :: ContextFreeRelation
duplicate  = (pure mempty) `union` ((pure ("a", "a")) `cat` duplicate `
    cat` (pure ("", "a"))) `union`
```

2.2 Arbitrary structures

In fact, more generally, context-free languages can be defined on any structure whose primitive operations can be lifted by an Applicative. For instance, trees:

[INSERT EXAMPLE]

What is a context-free language of trees? By an unfortunate accident of history, these are what have been called regular tree languages...

The only crucial piece for us to define a context-free language is the existence of a **FixedPt** function. **FixedPt** is only defined for increasing functions. Importantly, lifted operations always result in increasing functions:

Proposition 4. *If $\uparrow$ is a unary operation defined on S, then the lifted operation $\upuparrows$ on collections of S is increasing in its argument.*
If $\oplus$ is a binary operation defined on S, then the lifted operation $\circledast$ on collections of S is increasing in both arguments.
etc.

Proofs

2.3 There is a smallest fixed point

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 □

2.4 FixPt is increasing

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

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

□

2.5 *Not in main text*

Proof. This simply follows from the definition of the lifted version. If $L \subset L'$, then:

$$\upharpoonleft\upharpoonleft L = \{\upharpoonleft x \mid x \in L\} \subset \left\{\upharpoonleft x \mid x \in L'\right\} = \upharpoonleft\upharpoonleft L'$$

□