Derivative Parsing

Daniel Spiewak

December 2, 2010

1 Defining the Derivative

Def: Any context-free grammar, G, is composed entirely of rules with the following five forms (implicitly, four):

- $G = \mathsf{c}$ (terminal)
- $G = \epsilon$ (null)
- $G = G_1 \circ G_2$ (concatenation)
- $G = \mu g \,.\, G_1 \cup G_2$ (union)
- $G = g$ (recursion)

Note that this is merely an alternative way to formalize context-free grammars.

We can define a rewrite operation on a grammar which we call "the derivative with respect to c" (where c is some terminal). This operation on some grammar G will be expressed as $D_\mathsf{c}(G)$. The operation will produce a *new* grammar which obeys the following identity (let $L(G)$ mean "the language generated by G"):

$$\begin{aligned} L(G) &= \{\mathsf{c}w_1, w_2, \mathsf{c}w_3, \mathsf{c}w_4, \ldots\} \\ L(D_\mathsf{c}(G)) &= \{w_1, w_3, w_4, \ldots\} \end{aligned}$$

Thus, $L(D_\mathsf{c}(G))$ is precisely the language of G consisting of only the strings which start with c where the terminal c has been removed. Intuitively, we start out by assuming that c is in the First set of G. We then derive a new grammar, G', where First(G') is exactly Second(G), and analogously with higher-order sets. We are forcibly consuming c and rewriting the grammar to reflect this fact. Another way to put this would be:

$$\mathsf{c}w \in L(G) \Rightarrow w \in L(D_\mathsf{c}(G))$$

If c is *not* in the First set of G, then $D_\mathsf{c}(G) = \emptyset$.

We can define the derivative over all five grammatical forms in the following way:

$$\begin{aligned} D_\mathsf{c}(\mathsf{c}) &= \epsilon \\ D_\mathsf{c}(\epsilon) &= \emptyset \\ D_\mathsf{c}(\mu g \,.\, G_1 \cup G_2) &= \mu g' \,.\, D_\mathsf{c}(G_1) \cup D_\mathsf{c}(G_2) \\ D_\mathsf{c}(g) &= g' \text{ where } g \mapsto G \text{ and } g' \mapsto D_\mathsf{c}(G) \\ D_\mathsf{c}(G_1 \circ G_2) &= \begin{cases} D_\mathsf{c}(G_1) \circ G_2 & \text{if } \epsilon \notin L(G_1) \\ \mu g' \,.\, (D_\mathsf{c}(G_1) \circ G_2) \cup D_\mathsf{c}(G_2) & \text{otherwise} \end{cases} \end{aligned}$$

Note that there is a certain difficulty when it comes to the derivative $D_{\mathsf{c}}(g)$, where g is a recursive reference to an operation which we are *currently* deriving. This difficulty arrises from the fact that context-free grammars can be recursive (as formalized through the $\mu g \,.\, G_1 \cup G_2$ notation).

The solution to this problem is to use a call-by-need evaluation strategy, allowing incremental computation [1] of the derivative of a union operation. This incremental computation is critical as it allows the derivative to preserve the recursion which is present in the original grammar. Thus, the derivative will be recursive iff the original grammar is recursive. Further, the derivative will be recursive in *exactly* the same rules as the original grammar. This is born out by the definition of the derivative in the recursive case, which simply uses the derivative of the recursive operation which must have been previously *partially* computed.

2 Defining the Algorithm

Given this operation, we can define a context-free recognition algorithm for some input stream $\mathsf{c}t$, where c is the first token in the stream and t is the remainder:

$$\begin{aligned} \mathsf{recognize}(\mathsf{c}t, G) &= D_{\mathsf{c}}(G) \neq \emptyset \wedge \mathsf{recognize}(t, D_{\mathsf{c}}(G)) \\ \mathsf{recognize}(\epsilon, G) &= \epsilon \in L(G) \end{aligned}$$

Thus, we begin by assuming that the input stream is valid. If this is the case, then we can recursively take the derivative with respect to each successive token. If we at any point arrive at the null set, then there was no way that the grammar in question could have consumed the curent token and the recognition will be FALSE. If we reach the end of the stream and have *not* achieved a grammar who's language contains ϵ, then while we may have a valid prefix, we do not have a valid string and the recognition will again be FALSE. Otherwise, if we reach the end of the stream and have rewritten to a grammar which generates a language containing the empty string, then we have successfully recognized the input stream.

3 Proofs

3.1 Complexity of D_{c}

LEMMA The derivative has an asymptotic complexity of $O(k)$ where k is the number of concatenations in the grammar.

PROOF By induction.

CASE $D_{\mathsf{c}}(\mathsf{c})$

Requires $O(1)$ steps.

CASE $D_{\mathsf{c}}(\epsilon)$

Requires $O(1)$ steps.

CASE $D_{\mathsf{c}}(g)$

Uses a pre-computed partial result, and so requires $O(1)$ steps.

CASE $D_{\mathsf{c}}(\mu g \,.\, G_1 \cup G_2)$

By induction, taking the derivative of G_1 will require $O(k_1)$ (where k_1 is the number of concatenations in G_1). Analogously for G_2 requiring $O(k_2)$. Note that either G_1 or

G_2 (or both) may contain G. Thus, some memoization must be taking place at each level of the deconstruction. This implies that we will only compute the derivative for a particular operation at most *once*.

Thus, we don't need to concern ourselves with overlap between k_1 and k_2. In the worst-case, G_1 and G_2 will be entirely disjoint. In this case, the number of concatenations in G will be precisely the number in G_1 plus the number in G_2. Thus, taking the derivative of G will be $O(k)$, where k is the number of concatenations in G.

Case $D_{\mathsf{c}}(G_1 \circ G_2)$

By induction, taking the derivative of G_1 will require $O(k_1)$ (where k_1 is the number of concatenations in G_1). Analogously for G_2 requiring $O(k_2)$. The argument here is quite analogous to the argument we made for $G_1 \cup G_2$. However, the difference is that the number of concatenations in G is in fact $k_1 + k_2 + 1$, since the operation represented by the root of G is itself a concatenation.

This extra $+1$ is necessary as we require an extra step to derive concatenations. Recall the definition of the derivative for concatenation:

$$\begin{cases} D_{\mathsf{c}}(G_1) \circ G_2 & \text{if } \epsilon \notin L(G_1) \\ \mu g' \ . \ (D_{\mathsf{c}}(G_1) \circ G_2) \cup D_{\mathsf{c}}(G_2) & \text{otherwise} \end{cases}$$

In the worst case, we are creating a new union operation and effectively deepning our tree by one level. We will assume that this requires 1 step. Thus, the derivative of G where G is a concatenation requires $O(k_1 + k_2 + 1) = O(k)$.

The important point here is not the number of concatenations in the grammar, but rather that the complexity of D_{c} is linearly proportional to the number of operations in the grammar.

3.2 Complexity of the Nullability Test

You will note that the parsing algorithm (as well as the derivative algorithm) requires a test to see whether or not $\epsilon \in L(G)$. This operation itself requires $O(k)$ where k is the number of operations in G. This proof is trivial and is omitted so that I can get to the other, more interesting proofs.

3.3 Size Increase of G'

When we take the derivative, the resulting grammar G' may (in the worst case) be *more* complex than the original grammar G. This is because the derivative of the concatenation operation can create a new operation, deepening the tree as noted in Section 3.1.

Theorem Let $C(G)$ be the number of concatenations in G (this function is trivial to define). Let $|G|$ be the total number of operations in G. Prove that $|D_{\mathsf{c}}(G)| \leq |G| + C(G)$.

Proof By induction.

Case $D_{\mathsf{c}}(\mathsf{c})$

$|G| = 1$ and $C(G) = 0$. The derivative of G is a grammar G' such that $L(G') = \{\epsilon\}$. Since $|G'| = 1$, we have that $|G'| \leq |G| + C(G)$.

Case $D_{\mathsf{c}}(\epsilon)$

In this case, the result is the null set (or rather, the null grammar). The total number of operations in this grammar is 0, which is clearly less than $|\epsilon|$.

Case $D_{\mathsf{c}}(g)$

In this case, the derivative will be a recursive reference to a higher point in the derived grammar G'. This doesn't add any complexity however, because G already contained a recursive reference to that same point in the form of g. Thus, $|G'|$ will be precisely the size of the derivative of the operation referenced by g. As is intuitive, the recursive case follows by trivial induction.

Case $D_{\mathsf{c}}(\mu g \,.\, G_1 \cup G_2)$

At this point, I'm going to get a little hand-wavy because the symbols get a bit thick. We can safely argue that the derivative is not *adding* any operations in this case. Rather it is recursively operating on the left- and right-hand sides of the union and then putting the results back together with a single union operation. Thus, the net *gain* in in terms of operations is precisely 0. Thus, $|D_{\mathsf{c}}(G)| = |G|$ with the addition of whatever operations we gained from our recursive operations. Inductively, we know that we could not have gained more operations than we have concatenations, thus our bound is preserved.

Case $D_{\mathsf{c}}(G_1 \circ G_2)$

There are two cases here. In the case where $\epsilon \notin L(G_1)$, the resulting construction will have gained 0 operations over the source construction (we simply concatenate the result with G_2). Thus, it follows by an argument similar to what we used for $\cup$ that the bound is preserved.

In the second case, the result of the derivative will look like $\mu g' \,.\, .\, (D_{\mathsf{c}}(G_1) \circ G_2) \cup D_{\mathsf{c}}(G_2)$. Here we have not only concatenated our results back together (for a net gain of 0 operations), but we have also *added* a union operation. Thus, our net gain will be 1. This is within our bounds however, as we are working with a concatenation operation.

Thus, in the worst case, taking the derivative of a concatenation operation adds exactly 1 operation to the resulting grammar (plus whatever was added by the recursive derivations), which is precisely the number of concatenation operations in the grammar (by induction). This preserves our bounds.

In more straightforward terms, this is saying that while the rewritten grammar G' may be more complex than the original grammar G, that increase in complexity is bounded by the number of concatenation operations in the original grammar.

3.4 Complexity of Derivative Recognition

Here's the money proof. We have that the the grammar resulting from the derivative may be larger than the original grammar, but that size increase is bounded by the number of concatenation operations. Critically, *the number of concatenation operations does not increase.* The derivative will never *add* a concatenation to the grammar. It only adds union operations. Thus, we can repeatedly take the derivative of a grammar G, and the increase after each *subsequent* derivation will be bounded by $C(G)$, which is to say, the number of concatenations in the original grammar G.

We have previously shown that the number of steps required to take the derivative of some grammar G is $O(k)$, where $k = |G|$. We also have (by construction) that the derivative recognition algorithm requires exactly n sequential derivations where n is the length of the input. Thus, we have the following expansion for the worst-case complexity of the recognition algorithm:

$$k + 2k + 3k + \cdots + nk$$

This follows because $k = |G| = C(G)$ (in the worst case, where the entire grammar is concatenation), and each subsequent derivation requires k' steps, where $k' = |G'|$. Recall that $|G'| \leq |G| + C(G)$ and $k = C(G)$. Thus, each subsequent derivation will "add" k steps to the next derivation, and the first derivation will require at most k steps.

This expansion can be rewritten as the following sum:

$$\sum_{i=1}^{n} ik = k \sum_{i=1}^{n} i < kn^2$$

Thus, derivative recognition is $O(n^2)$ where n is the length of the input.

References

[1] R.S. Sundaresh and Paul Hudak, *Incremental computation via partial evaluation*, Conference Record of the 18th Annual ACM Symposium on POPL, 1991, pp. 1–13.