A Typed, Algebraic Approach to Parsing

NEEL KRISHNASWAMI, University of Cambridge

In this paper, we recall the definition of the *context-free expressions* (or μ-regular expressions), an algebraic presentation of the context-free languages. Then, we define a core type system for the context-free expressions to pick out those expressions which can be parsed unambiguously by predictive algorithms such as recursive descent or LL(k) style tabular parsing. Next, we extend the notion of Brzozowski derivative from regular expressions to the typed context-free expressions. Finally, we define a type system for context-free expressions which permits left-recursive and non-left-factored expressions, and which elaborates them into equivalent into core terms. All of these type systems and algorithms are proved sound with respect to the denotational semantics of context-free expressions.

CCS Concepts: •**Theory of computation → Grammars and context-free languages; Type theory;**

Additional Key Words and Phrases: parsing, context-free languages, type theory, Brzozowski derivatives, Kleene algebra

ACM Reference format:
Neel Krishnaswami. 2017. A Typed, Algebraic Approach to Parsing. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 25 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The theory of parsing is one of the oldest and most well-developed areas in computer science: the bibliography to Grune and Jacobs's *Parsing Techniques: A Practical Guide* lists over 1700 references! Nevertheless, the foundations of the subject have remained remarkably stable: context-free languages are specified in Backus–Naur form, and parsers for these specifications are implemented using algorithms derived from automata theory. This integration of theory and practice has yielded many benefits: we have algorithms for parsing unambiguous grammars efficiently (Knuth 1965; Lewis and Stearns 1968) and with excellent error reporting for bad input strings (Jeffery 2003). But despite its (in many ways justified) dominance, this tradition has long had its dissidents as well.

BNF is a very pleasant notation for specifying whole languages, but it adheres surprisingly poorly to the canons of programming language design. In particular, it has no notion of binding structure or variable, and consequently it has no notion of substitution either. Variables can be added in a *post hoc* fashion (as the Menhir tool (Pottier and Régis-Gianas) does), but the core parse table generation must still operate upon the fully-expanded grammar.

This worsens one of the other, already-problematic, issues with automata-theoretic approaches to parsing – namely, problems (e.g., ambiguity or otherwise falling outside of the class of accepted inputs) are not diagnosed in terms of the input grammar. Instead, users of parser generators are presented with the intermediate state of a failed attempt to build a parse table. Without a good understanding of the precise compilation algorithm in use, it becomes very difficult to diagnose what the error is. This is a problem programmers do not tolerate in any other context: nearly every language above the level of machine code has facilities for variable binding and abstraction, and compilers and interpreters report type errors in terms of the input program rather than dumping the compiler's internal intermediate representations on the user.

Techniques for addressing the first issue – the lack of good binding structure – have been known for a surprisingly long time. This paper begins by recalling that work: the context-free languages can be understood as the extension of regular expressions with variables and a least-fixed point operator (typically dubbed the "context-free expressions" or "μ-regular expressions"). This permits replacing the concept of nonterminal from

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

traditional grammatical formalisms with standard concepts of semantics such as variables and explicit fixed-point operators. We also give the denotational semantics, and show how context-free expressions can be interpreted as languages, and that this semantics validates the expected equations. Our presentation is by no means novel, but we hope it will draw attention to a line of work that deserves to be more widely known.

The stage now set, we can move on to the contributions of this paper:

- First, we extend the framework of pure Kleene algebra or regular expressions by defining a semantic notion of type for languages. We then use this notion of type as the basis for a syntactic type system which checks whether a context-free expression is suitable for predictive (or recursive descent) parsing – i.e., we give a type system for left-factored, non-left-recursive, unambiguous grammars. We then prove that this type system is well-behaved (i.e., syntactic substitution preserves types, and sound with respect to the denotational semantics), and also prove that well-typed grammars are unambiguous.
- Next, we show how Brzozowski's notion of a derivative of a regular expression can be extended to the notion of a derivative of a context-free expression. We use the fact that our derivative algorithm acts upon well-typed inputs to prove that it is always well-defined, and then additionally prove that it is semantically correct (i.e., the syntactic derivative operation yields a grammar whose semantics is the Brzozowski derivative). Furthermore, typing ensures that a number of performance problems with derivative parsing are statically ruled out.
- Next, we show how to give a relaxed type system admitting a more liberal class of grammars. In particular, the relaxed type system permits writing grammars which use left-recursion and are not left-factored. We formulate the surface type system as an elaboration semantics translating every well-typed relaxed context-free expression into an equivalent expression typeable in the core type system. In addition to being a type- and semantics-preserving translation, we additionally show the elaboration actually commutes with substitution. (That is, if g translates to $\hat{g}$ and g' translates to $\hat{g}'$, then $[g/x]g'$ translates to $[\hat{g}/x]\hat{g}'$.)

  As a result, a programmer can think about their grammars purely in terms of the surface syntax, without having to worry about what the grammars elaborate into, and with errors reported in terms of problems in the input syntax. On the other hand, the implementer can implement parsing with an algorithm hardly more complex than Brzozowski's original algorithm for taking derivatives of regular expressions.

It is worth remarking that all of our proofs use standard techniques from type theory and denotational semantics. It is a bit surprising how easily the techniques of semantics transfer to a different domain.

2 CONTEXT FREE EXPRESSIONS

We begin by defining the grammar of *context-free expressions* in Figure 1. Given a finite set of characters Σ, the context free expressions are ϵ, denoting the language containing only the empty string; the expression c, denoting the language containing the 1-element string c; the expression $g \cdot g'$, denoting the strings which are concatenations of strings from g and strings from g'; the expression $\bot$, denoting the empty language; the expression $g \vee g'$, denoting the language which is the union of the languages denoted by g and g'; $[g]$, which denotes all the non-empty strings in g; and the variable reference x and the least fixed point operator $\mu x.\ g$. Variable scoping is handled as usual; the fixed point is considered to be a binding operator, free and bound variables are defined as usual, and terms are considered only up to α-equivalence. As a notational convention, we use the variables x, y, z to indicate variables, and a, b, c to represent characters from the alphabet Σ. In examples of grammars, we will also sometimes write write variables as capitalised words (e.g. *Exp*) and characters c in boxes (i.e., as $\boxed{c}$) to help distinguish variables and terminal symbols from each other and other punctuation.

Intuitively, the context free expressions can be understood as an extension of the regular expressions with a least fixed point operator. We omit the Kleene star $g*$ from the core syntax since it is definable by means of a fixed point: $g* \triangleq \mu x.\ \epsilon \vee g \cdot x$. The nonemptiness operator $[g]$ is an admissible operator in ordinary regular algebra, but for our purposes it is much more convenient to introduce it as a primitive.

2.1 Examples

Below, we give some examples of context-free expressions. First, we give a grammar for Lisp style s-expressions.

$$\begin{array}{ll} \mu Sexp. & a \vee \\ & \boxed{(} \cdot Sexp * \cdot \boxed{)} \end{array}$$

An s-expression is either an atom a, or a sequence of s-expressions surrounded by parentheses. Recall that the Kleene star is just an abbreviation for a nested fixed point, so the previous grammar is an abbreviation for:

$$\begin{array}{ll} \mu Sexp. & a \vee \\ & \boxed{(} \cdot (\mu x.\ \epsilon \vee Sexp \cdot x) \cdot \boxed{)} \end{array}$$

Arithmetic expressions can also be expressed as context-free expressions (where we take n to range over numeric literals):

$$\begin{array}{ll} \mu Exp. & n \vee \\ & \boxed{(} \cdot Exp \cdot \boxed{)} \vee \\ & Exp \cdot \boxed{+} \cdot Exp \end{array}$$

The resemblance to context free grammars should be very clear. However, as we have already seen, we are free to nest fixed points, and in addition we can also concatenate expressions freely. So we can express a version of the previous example so that it is left-factored and removes left-recursion as follows:

$$\mu Exp. \quad (n \ \vee \boxed{(} \cdot Exp \cdot \boxed{)}) \cdot (\boxed{+} \cdot Exp)*$$

This transformation is very familiar to those who have implemented recursive descent parsers, and as we will see below, it is an instance of one of the general equations that fixed points satisfy.

2.2 Semantics and Untyped Equational Theory

The denotational semantics of context-free expressions are also given in Figure 1. We interpret each context-free expression as a language (i.e., a subset of the set of all strings Σ^*). The interpretation function interprets each context-free expression as a function taking an interpretation of the free variables to a language. The meaning of $\bot$ is the empty set; the meaning of $g \vee g'$ is the union of the meanings of g and g'; the meaning of ϵ is the singleton set containing the empty string; the meaning of c is singleton set containing the one-character string c; and the meaning of $g \cdot g'$ are those strings formed from a prefix drawn from g and a suffix drawn from g'. The meaning of $[g]$ are the elements of g which are *not* the empty string; variables x are looked up in the environment; and $\mu x.\ g$ is interpreted as the least fixed point of g with respect to x.

Proposition 2.1. *The context-free expressions satisfy the equations of a idempotent semiring with* $(\vee, \bot)$ *as addition and its unit, and* $(\cdot, \epsilon)$ *as the multiplication.*

(1) $g_1 \vee (g_2 \vee g_3) = (g_1 \vee g_2) \vee g_3$
(2) $g \vee g' = g' \vee g$
(3) $g \vee \bot = g$
(4) $g \vee g = g$
(5) $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$
(6) $g \cdot g' = g' \cdot g$

$$g \quad ::= \quad \bot \mid g \vee g' \mid \epsilon \mid c \in \Sigma \mid g \cdot g' \mid [g] \mid x \mid \mu x.\, g$$

$$\begin{array}{lcl}
[\![\bot]\!]\,\gamma & = & \emptyset \\
[\![g \vee g']\!]\,\gamma & = & [\![g]\!]\,\gamma \;\cup\; [\![g']\!]\,\gamma' \\
[\![\epsilon]\!]\,\gamma & = & \{\varepsilon\} \\
[\![c]\!]\,\gamma & = & \{c\} \\
[\![g \cdot g']\!]\,\gamma & = & \{w \cdot w' \mid w \in [\![g]\!]\,\gamma \wedge w' \in [\![g']\!]\,\gamma'\} \\
[\![[g]]\!]\,\gamma & = & \{w \in [\![g]\!]\,\gamma \mid w \neq \varepsilon\} \\
[\![x]\!]\,\gamma & = & \gamma(x) \\
[\![\mu x.\, g]\!]\,\gamma & = & \mathit{fix}(\lambda X.\, [\![g]\!]\,(\gamma, X/x))
\end{array}$$

$$\mathit{fix}(f) \quad = \quad \bigcup_{i \in \mathbb{N}} L_i \text{ where } \begin{array}{lcl} L_0 & = & \emptyset \\ L_{n+1} & = & f(L_n) \end{array}$$

Fig. 1. Syntax and semantics of context-free expressions

(7) $g \cdot \epsilon = g$
(8) $(g_1 \vee g_2) \cdot g = (g_1 \cdot g) \vee (g_2 \cdot g)$
(9) $g \cdot (g_1 \vee g_2) = (g \cdot g_1) \vee (g \cdot g_2)$
(10) $g \cdot \bot = \bot \cdot g = \bot$

In addition, fixed points satisfy the following equations:

(1) $\mu x.\, g = [\mu x.\, g/x]g$
(2) $\mu x.\, g_0 \vee x \cdot g_1 = \mu x.\, g_0 \cdot g_1*$

In addition, the nonnull operator satisfies the following equations:

(1) $[\epsilon] = \bot$
(2) $[g \cdot g'] = ([g] \cdot g') \vee (g \cdot [g'])$
(3) $[\bot] = \bot$.
(4) $[g \vee g'] = [g] \vee [g']$.
(5) $[c] = c$.
(6) $g \vee [g] = g$.

The semiring equations are all standard. The first fixed point equation is the standard unrolling equation for fixed points. The second equation is perhaps less well-known to semanticists, but is familiar to anyone who has implemented a recursive descent parser: it is the rule for left-recursion elimination. Leiß (1991) gives a proof of this rule in the general setting of Kleene algebra with fixed points, but it is easily proved directly (by induction on the number of unrollings of the fixed point) as well.

The context free expressions are equivalent in expressive power to Backus-Naur form. Syntactically, the main difference between the two is that BNF offers a single n-ary mutually-recursive fixed point at the outermost level, and context-free expressions only have unary fixed points, but permit nesting them. As is well-known, these two forms of recursion are interderivable via Bekič's lemma (Bekič and Jones 1984). (See Grathwohl et al. (2014) for the proof in the context of language theory rather than domain theory.)

$$\text{Types} \quad \tau \quad \in \quad \{\textsc{Null} : 2;\ \textsc{First} : \mathcal{P}(\Sigma);\ \textsc{Followlast} : \mathcal{P}(\Sigma)\}$$

$$\begin{array}{lcl} \tau \,\#\, \tau' & \triangleq & \neg(\tau.\textsc{Null} \wedge \tau'.\textsc{Null}) \wedge (\tau.\textsc{First} \cap \tau'.\textsc{First} = \emptyset) \\ \tau \circledast \tau' & \triangleq & \tau.\textsc{Followlast} \cap \tau'.\textsc{First} = \emptyset \wedge \neg\tau.\textsc{Null} \end{array}$$

$$[\tau] = \{\textsc{Null} = \text{false};\ \textsc{First} = \tau.\textsc{First};\ \textsc{Followlast} = \tau.\textsc{Followlast}\}$$

$$\tau_\perp = \{\textsc{Null} = \text{false};\ \textsc{First} = \emptyset;\ \textsc{Followlast} = \emptyset\}$$

$$\tau_1 \vee \tau_2 = \left\{ \begin{array}{lcl} \textsc{Null} & = & \tau_1.\textsc{Null} \vee \tau_2.\textsc{Null} \\ \textsc{First} & = & \tau_1.\textsc{First} \cup \tau_2.\textsc{First} \\ \textsc{Followlast} & = & \tau_1.\textsc{Followlast} \cup \tau_2.\textsc{Followlast} \end{array} \right\}$$

$$\tau_\epsilon = \{\textsc{Null} = \text{true};\ \textsc{First} = \emptyset;\ \textsc{Followlast} = \emptyset\}$$

$$\tau_c = \{\textsc{Null} = \text{false};\ \textsc{First} = \{c\};\ \textsc{Followlast} = \emptyset\}$$

$$\tau_\perp \cdot \tau = \tau_\perp$$

$$\tau \cdot \tau_\perp = \tau_\perp$$

$$\tau_1 \cdot \tau_2 = \left\{ \begin{array}{lcl} \textsc{Null} & = & \tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null} \\ \textsc{First} & = & \tau_1.\textsc{First} \cup (\text{if } \tau_1.\textsc{Null} \text{ then } \tau_2.\textsc{First} \text{ else } \emptyset) \\ \textsc{Followlast} & = & \tau_2.\textsc{Followlast} \cup (\text{if } \tau_2.\textsc{Null} \text{ then } \tau_1.\textsc{Followlast} \text{ else } \emptyset) \end{array} \right\}$$

$$\tau* = \left\{ \begin{array}{lcl} \textsc{Null} & = & \text{true} \\ \textsc{First} & = & \tau.\textsc{First} \\ \textsc{Followlast} & = & \tau.\textsc{Followlast} \cup \tau.\textsc{First} \end{array} \right\}$$

Fig. 2. Definition of Types

However, parsing general context-free grammars require algorithms such as Earley, Tomita or CYK algorithms, all of which have cubic-time worst-case complexity[1], and furthermore parsing may be *ambiguous*, with multiple possible parse trees for the same string. However, it is well-known that grammars falling into more restrictive classes, such as the LL(k) and LR(k) classes, can be parsed efficiently in linear time. In this paper we will focus on devising type systems which identify languages suitable for parsing by predictive means (i.e., recursive descent).

2.3 Types for Languages

There are two primary sources of ambiguity in predictive parsing.

First, when we parse a string w against a grammar of the form $g_1 \vee g_2$, then we have to decide whether w belongs to g_1 or to g_2. If we cannot predict which branch to take, then we have to backtrack. (Naive parser combinators (Hutton 1992) are particularly prone to this problem.)

Second, when we parse a string w_1 against a grammar of the form $g_1 \cdot g_2$, we have to break it into two pieces w_1 and w_2 so that $w = w_1 \cdot w_2$ and w_1 belongs to g_1 and w_2 belongs to g_2. If there are many possible ways for a string to be split into the g_1-fragment and the g_2-fragment, then we have to try them all, again introducing backtracking into the algorithm.

[1]The current best bound via Valiant's algorithm (Valiant 1975), which is subcubic (currently $O(n^{2.378})$).

Hence we need properties we can use to classify the languages which can be parsed efficiently. To do so, we introduce the following functions on languages:

$$\begin{array}{lcl} \textsc{Null} & : & \Sigma^* \to 2 \\ \textsc{Null}\ (L) & = & \text{if } \epsilon \in L \text{ then true else false} \\ \\ \textsc{First} & : & \Sigma^* \to \mathcal{P}(\Sigma) \\ \textsc{First}\ (L) & = & \{c \mid \exists w \in \Sigma^*.\ c \cdot w \in L\} \\ \\ \textsc{Followlast} & : & \Sigma^* \to \mathcal{P}(\Sigma) \\ \textsc{Followlast}\ (L) & = & \{c \mid \exists w \in L, w' \in \Sigma^*.\ w \cdot c \cdot w' \in L\} \end{array}$$

$\textsc{Null}\ (L)$ returns true if the empty string is in L, and false otherwise. $\textsc{First}\ (L)$ is the set of characters that can start any string in L, and the $\textsc{Followlast}\ (L)$ set are the set of characters which can follow the last character of a string in L. (This will serve a function similar to the Follow sets used to construct LL(k) parse tables.)

We now define a type τ (see Figure 2) as a record of three fields, recording a nullability, First set, and Followlast set. We say that a language L satisfies a type τ (written $L \models \tau$), when:

$$L \models \tau \quad \iff \quad \begin{array}{l} \textsc{Null}\ (L) \implies \tau.\textsc{Null} \ \wedge \\ \textsc{First}\ (L) \subseteq \tau.\textsc{First} \ \wedge \\ \textsc{Followlast}\ (L) \subseteq \tau.\textsc{Followlast} \end{array}$$

Essentially, we want to ensure that the type τ is an overapproximation of the properties of L.

What makes this notion of type interesting are the following two lemmas.

Lemma 2.2. *(Unique decomposition) Suppose L and M are languages. Then:*

1. *If $\textsc{First}\ (L) \cap \textsc{First}\ (M) = \emptyset$ and $\neg(\textsc{Null}\ (L) \wedge \textsc{Null}\ (M))$, then $L \cap M = \emptyset$.*
2. *Suppose $\textsc{First}\ (L) \cap \textsc{Followlast}\ (M) = \emptyset$ and $\neg\textsc{Null}\ (L)$. If $w \in L \cdot M$, then there is a unique $w_L \in L$ and $w_m \in M$ such that $w_L \cdot w_M = w$.*

The first property ensures that when we take a union $L \cup M$, then each string in the union belongs uniquely to either L or M. This eliminates "disjunctive non-determinism" from parsing; if we satisfy this condition, then parsing an alternative $g \vee g'$ will lead to parsing exactly one or the other branch; there will never be a case where both g and g' contain the same string. Indeed, we can always tell whether to parse with g or g' just by looking at the first token of the input.

The second property eliminates "sequential non-determinism": it gives a condition ensuring that if we concatenate two languages L and M, each string in the concatenated languages can be *uniquely* broken into an L-part and an M-part. Therefore if L and M satisfy this condition, then as soon as we have recognised an L-word in the input, we can move immediately to parsing an M-word (when parsing for $L \cdot M$).

Of course, practical use of language types requires being able to easily calculate types from smaller types. Happily, this is possible, and in Figure 2, we give both the grammar of types (just a ternary record) , as well as a collection of basic types and operations on them. We define $\tau_\perp$ to be the type of the empty language, and so it is not nullable and has empty First and Followlast sets. We define τ_ϵ to be the type of the language containing just the empty string, and it *is* nullable and but otherwise has has empty First and Followlast sets. τ_c is the type of the language containing just the single-character string c, and so it is not nullable, has a first set of $\{c\}$, and has an empty Followlast set. The $[\tau]$ operation gives a type for removing the empty string for a language, by setting Null to false but leaving the First and Followlast fields alone. The $\tau_1 \vee \tau_2$ operation constructs the join of two types, by taking the logical-or of the Null fields, and the unions of the First and Followlast sets.

The $\tau_0 \cdot \tau_1$ operation calculates a type for a language concatenation by first taking the conjunction of the Null fields. Then, the First set is the First set of τ_0, unioned together with the First set of τ_1 when τ_0 is nullable. Conversely, the Followlast set is the Followlast set of τ_2, merged together with the Followlast set of τ_1 when τ_2 is nullable.

One fact of particular note is that these two definitions are *not correct* in general. That is, if $L \models \tau_0$ and $M \models \tau_1$, then in general $L \cdot M \not\models \tau_0 \cdot \tau_1$. It only holds when L and M are *separable*. That is they must meet the preconditions conditions of the decomposition lemma (Lemma 2.2). We define a separability predicate $\tau \circledast \tau'$ to indicate this, which holds when τ.Follow and τ'.First are disjoint, and τ.Null is false. Similarly, we must also define *apartness* $\tau \# \tau'$ for non-overlapping languages, by checking that at most one of τ.Null and τ'.Null hold, and that the First sets are disjoint.

This lets us prove the following properties of the type operators and language satisfaction:

Lemma 2.3. *(Properties of Satisfaction)*

(1) $L \models \tau_\bot$ *if and only if* $L = \emptyset$.
(2) $L \models \tau_\epsilon$ *if and only if* $L = \{\epsilon\}$.
(3) *If* $L = \{c\}$ *then* $L \models \tau_c$.
(4) *If* $L \models \tau$ *and* $M \models \tau'$ *and* $\tau \circledast \tau'$, *then* $L \cdot M \models \tau \cdot \tau'$.
(5) *If* $L \models \tau$ *and* $M \models \tau'$ *and* $\tau \# \tau'$, *then* $L \cup M \models \tau \vee \tau'$.
(6) *If* $L \models \tau$ *and* $\tau \circledast \tau$, *then* $L* \models \tau*$.
(7) *If* X *is a set and* L *is an* X*-indexed family of languages, then if* $L_x \models \tau$, *then* $\bigcup_{x \in X} L_x \models \tau$.

Example. Consider the example of s-expressions:

$$Sexp \triangleq \mu Sexp.\ a \ \vee \boxed{(} \cdot Sexp* \cdot \boxed{)}$$

So we can say that $[\![Sexp]\!] \models \tau$, where the type τ is:

$$\tau \triangleq \left\{ \begin{array}{lcl} \textsc{Null} & = & \text{false} \\ \textsc{First} & = & \{(, a\} \\ \textsc{Followlast} & = & \emptyset \end{array} \right\}$$

Since *Sexp* does not produce any empty strings, we know τ.Null = false. The First set if $\{(, a\}$, since an s-expression can begin only with an open parenthesis or an atom, and its Followlast set is *empty*, because every s-expression is explicitly and fully bracketed. On the other hand, the Followlast set of *Sexp*$*$ must be at least $\{(, a\}$, the same as its First set, since it denotes a sequence of s-expressions.

3 A TYPE SYSTEM FOR CONTEXT-FREE EXPRESSIONS

In this section, we use the semantic types and type operators defined in the previous system to define a syntactic type system for grammars parseable by recursive descent. The main judgement we introduce (in Figure 4) is the typing judgement $\Gamma; \Delta \vdash g : \tau$, which is read as "under ordinary hypotheses Γ and guarded hypotheses Δ, the grammar g has the type τ."

The distinctive feature of this judgement form is that there are *two* contexts for variables, one for ordinary variables and one for the so-called "guarded" variables, which identify variables which can only occur to the right of a nonempty string not containing that variable. So if $x : \tau$ is a guarded hypothesis, the grammar x is considered ill-typed, but the grammar $\boxed{c} \cdot x$ is permitted.

The CoreVar rule implements this restriction. It says that if $x : \tau \in \Gamma$, then x has the type τ. Note that it *does not* permit referring to variables in the guarded context Δ.

$$\begin{array}{lrcl} \text{Contexts} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, x : \tau \\ \text{Substitutions} & \gamma, \delta & ::= & \cdot \mid \gamma, L/x \end{array}$$

Fig. 3. Contexts and Substitutions

$$\boxed{\Gamma; \Delta \vdash g : \tau}$$

$$\frac{}{\Gamma; \Delta \vdash \epsilon : \tau_\epsilon}\ \textsc{CoreEps} \qquad \frac{}{\Gamma; \Delta \vdash c : \tau_c}\ \textsc{CoreChar} \qquad \frac{}{\Gamma; \Delta \vdash \bot : \tau_\bot}\ \textsc{CoreBot}$$

$$\frac{\Gamma; \Delta \vdash g : \tau}{\Gamma; \Delta \vdash [g] : [\tau]}\ \textsc{CoreNonempty} \qquad \frac{x : \tau \in \Gamma}{\Gamma; \Delta \vdash x : \tau}\ \textsc{CoreVar} \qquad \frac{\Gamma; \Delta, x : \tau \vdash g : \tau'}{\Gamma; \Delta \vdash \mu x : \tau.\ g : \tau}\ \textsc{CoreFix}$$

$$\frac{\Gamma; \Delta \vdash g : \tau \qquad \Gamma, \Delta; \cdot \vdash g' : \tau' \qquad \tau \circledast \tau'}{\Gamma; \Delta \vdash g \cdot g' : \tau \cdot \tau'}\ \textsc{CoreCat} \qquad \frac{\Gamma; \Delta \vdash g : \tau \qquad \Gamma; \Delta \vdash g' : \tau' \qquad \tau \# \tau'}{\Gamma; \Delta \vdash g \vee g' : \tau \vee \tau'}\ \textsc{CoreVee}$$

Fig. 4. Typing for Context-free Expressions

The CoreEps rule says that the empty string has the empty string type τ_ϵ, and similarly the rules for the other constants CoreChar and CoreBot return types τ_c and $\tau_\bot$ for the singleton string and empty grammar constants.

The CoreCat rule governs when a concatenation $g \cdot g'$ is well-typed. Obviously, both g and g' have to be well-typed at τ and τ', but in addition, the two types have to be separable (i.e., $\tau \circledast \tau'$ must hold). One consequence of the separability condition is that $\tau.\textsc{Null} = \text{false}$ – that is, g must be non-empty. As a result , we can allow g' to refer freely to the guarded variables, and so when type checking g', we move all of the guarded hypotheses into the unrestricted context. The CoreVee rule explains when a union is well-typed. If g and g' are well-typed at τ and τ', and the two types are apart (i.e., $\tau \# \tau'$), then the union is well-typed at $\tau \vee \tau'$.

This machinery is all put to use in the CoreFix rule. It says that a context-free expression[2] $\mu x : \tau.\ g$ is well-typed when, under the *guarded* hypothesis that x has type τ, the whole grammar g has type τ. Since the binder of the fixed point is a guarded variable, this ensures that the fixed point as a whole is guarded, and that no left-recursive definitions are typeable. (This is very similar to the typing rule for guarded recursive definitions in Atkey and McBride (2013); Krishnaswami (2013).)

The type system satisfies the expected syntactic properties, such as weakening and substitution.

Lemma 3.1. *(Weakening and Transfer) We have that:*

(1) *If* $\Gamma; \Delta \vdash g : \tau$, *then* $\Gamma, x : \tau; \Delta \vdash g : \tau'$.
(2) *If* $\Gamma; \Delta \vdash g : \tau$, *then* $\Gamma; \Delta, x : \tau \vdash g : \tau'$.
(3) *If* $\Gamma; \Delta, x : \tau' \vdash g : \tau$ *then* $\Gamma, x : \tau'; \Delta \vdash g : \tau$.

Proof. By induction on derivations. We prove these properties sequentially, first proving 1, then 2, then 3. □

[2]We have added a type annotation to the binder to make typing syntax-directed. As an abuse of notation, we will not distinguish annotated from unannotated context-free expression, assuming that annotations are silently dropped as needed.

We can weaken in both judgements, and additionally support a transfer property, which says that if a grammar g typechecks with $x : \tau$ in the guarded context, it also typechecks with $x : \tau$ in the unguarded context. The intuition behind transfer is that since guarded variables can be used in fewer places than unguarded ones, a term that typechecks with a guarded variable x will also typecheck when x is unguarded.

These properties then let us prove the syntactic substitution lemma.

Lemma 3.2. *(Syntactic Substitution) We have that:*

(1) *If* $\Gamma, x : \tau; \Delta \vdash g' : \tau'$ *and* $\Gamma; \Delta \vdash g : \tau$, *then* $\Gamma; \Delta \vdash [g/x]g' : \tau'$.
(2) *If* $\Gamma; \Delta, x : \tau \vdash g' : \tau'$ *and* $\Gamma, \Delta; \cdot \vdash g : \tau$, *then* $\Gamma; \Delta \vdash [g/x]g' : \tau'$.

Proof. By induction on the relevant derivations. □

We give two substitution principles, one for unguarded variables and one for guarded variables. Since guarded variables are always used in a guarded context, we do not need to track the guardedness in the term we substitute. As a result, the premise of the guarded substitution lemma only requires that the term g being satisfies have the typing $\Gamma, \Delta; \cdot \vdash g : \tau$ – it does not need to enforce any requirements on the guardedness of the term being substituted.

Next, we will show that the type system is sound – that the language each well-typed context-free expression denotes in fact satisfies the type that the type system ascribes to it. Before we can prove the semantic soundness of the type system, we first we extend satisfaction to contexts as follows:

Definition 3.3. (Context Satisfaction) We define context satisfaction $\gamma \models \Gamma$ as the recursive definition:

$$\begin{array}{rcll} \cdot & \models & \cdot & \triangleq \text{ always} \\ (\gamma, L/x) & \models & (\Gamma, x : \tau) & \triangleq \gamma \models \Gamma \text{ and } L \models \tau \end{array}$$

This says that a substitution γ satisfies a context Γ, if the language each variable in γ refers to satisfies the type that Γ ascribes to it. As usual, this is what lets us define semantic soundness for open terms:

Definition 3.4. (Semantic Soundness) We define context satisfaction $\Gamma; \Delta \models g : \tau$ as the formula:

$$\Gamma; \Delta \models g : \tau \quad \triangleq \quad \forall \gamma, \delta. \text{if } \gamma \models \Gamma \text{ and } \delta \models \Delta \text{ then } [\![g]\!]\ (\gamma, \delta) \models \tau$$

Now we can prove that the interpretation of a well-typed grammar satisfies its type, using context satisfaction to extend our induction hypothesis to handle open terms.

Theorem 3.5. *(Soundness) If* $\Gamma; \Delta \vdash g : \tau$ *then* $\Gamma; \Delta \models g : \tau$.

Proof. The proof is by induction on the typing derivation. All of the cases are straightforward, with the main point of interest being the proof of the fixed point rule.

- **Case** CoreFix : $\dfrac{\Gamma; \Delta, x : \tau \vdash g : \tau'}{\Gamma; \Delta \vdash \mu x : \tau.\ g : \tau}$

  We have $\gamma \models \Gamma$ and $\delta \models \Delta$. By induction, we know that if $(\delta, X/x) \models (\Delta, x : \tau)$, then $[\![g]\!]\ (\gamma, \delta, X/x) \models \tau$.

  Let $F = \lambda.\ [\![g]\!]\ (\gamma, \delta, X/x)$. We want to show that $\mu(F) \models \tau$, or by expanding the definition, that $\bigcup_{i \in \mathbb{N}} F^i(\emptyset) \models \tau$. By Lemma 2.3, it suffices to show that for every i, $L_i = F^i(\emptyset) \models \tau$.

  We show this by a nested induction. For the base case of $i = 0$, note that $\emptyset \models \tau$. Now suppose that $i = n + 1$ and that $L_n \models \tau$. We want to show that $L_{n+1} \models \tau$. This follows from the outer induction hypothesis, using the fact that $(\delta, L_n/x) \models \Delta, x : \tau$.

  □

Finally, the fact that we have types means we can extend the equational theory of grammars to exploit typing.

Definition 3.6. (Semantic Equality) We define $\Gamma; \Delta \models g \equiv g'$ as follows:

$$\Gamma; \Delta \models g \equiv g' \quad \triangleq \quad \forall \gamma \models \Gamma, \delta \models \Delta.\ [\![g]\!]\,(\gamma, \delta) = [\![g']\!]\,(\gamma, \delta)$$

Lemma 3.7. *(Properties of Semantic Equivalence)*

- *Semantic equivalence is an equivalence relation.*
- *Semantic equivalence is a congruence.*
- *All of the untyped equations in Proposition 2.1 are semantic equivalences.*
- *If* $\Gamma; \Delta \models g : \tau$ *and* $\neg\tau.\textsc{Null}$ *and* $\tau.\textsc{First} = \emptyset$*, then* $\Gamma; \Delta \models g \equiv \bot$.
- *If* $\Gamma; \Delta \models g : \tau$ *and* $\neg\tau.\textsc{Null}$ *, then* $\Gamma; \Delta \models g \equiv [g]$.

Example. Since substitution is sound, we feel free to use local bindings in our examples. We begin with a simple arithmetic expression grammar, written in the traditional style for recursive descent.

$$\begin{array}{ll} \mu Exp : \tau_{Exp}. & \textsf{let } Atom : \tau_{Atom} = n \vee (\boxed{(} \cdot Exp \cdot \boxed{)}) \textsf{ in} \\ & \textsf{let } Term : \tau_{Term} = Atom \cdot (\boxed{+} \cdot Atom)* \textsf{ in} \\ & Term \cdot (\boxed{\times} \cdot Term)* \end{array}$$

$$\begin{array}{lcl} \tau_{Atom} & \triangleq & \{\textsc{Null} = \mathsf{false};\ \textsc{First} = \{(, n\};\ \textsc{Followlast} = \emptyset\} \\ \tau_{Term} & \triangleq & \{\textsc{Null} = \mathsf{false};\ \textsc{First} = \{(, n\};\ \textsc{Followlast} = \{+\}\} \\ \tau_{Exp} & \triangleq & \{\textsc{Null} = \mathsf{false};\ \textsc{First} = \{(, n\};\ \textsc{Followlast} = \{+, \times\}\} \end{array}$$

This is basically the traditional presentation of arithmetic operations with precedence as a recursive descent parser. Note that we were able to make explicit the fact that the *Atom* and *Term* subgrammars are not recursive. Note also that the Followlast set grows with the new operators at each precedence level.

3.1 Typed Context-Free Expressions are Unambiguous

In this subsection, we prove that there is at most one way to parse a typed context-free expression. To show this, we first give a judgement, $g \Mapsto w$ (in Figure 5), which supplies inference rules explaining when a grammar g can produce a word w.

The rules are fairly straightforward. DEps states that the empty grammar can produce the empty string, and DChar states that a single-character grammar c can produce the singleton string c. The DCat rule says that if $g \Mapsto w$ and $g' \Mapsto w'$, then $g \cdot g' \Mapsto w \cdot w'$. The D$\vee$L rule says that a disjunctive grammar $g_1 \vee g_2$ can produce a string w if g_1 can, and symmetrically the D$\vee$R rule says that it can produce w if g_2 can; the DFix rule asserts that a fixed point grammar $\mu x : \tau.\ g$ can generate a word if its unfolding can; and finally the DNonempty rule says that $[g]$ can generate a word w if g can generate it and w is nonempty. There is no rule for the empty grammar $\bot$, since it denotes the empty language. There are also no rules for variables, since we only consider closed context-free expressions.

Observe that for general untyped context-free expressions, there can be multiple ways that a single string can be generated from the same grammar. For example, $c \vee c \Mapsto c$ either along the left branch or the right branch, using either the D$\vee$L or D$\vee$R derivation rules. This reflects the fact that the grammar $c \vee c$ is ambiguous: there are multiple possible derivations for it. So we will prove that our type system identifies unambiguous grammars by proving that for each typed, closed grammar g, and each word w, there is exactly one derivation $g \Mapsto w$ just when $w \in [\![g]\!]$.

Proving this directly on the syntax is a bit inconvenient, since unfolding a grammar can increase its size. So we will first identify a metric on typed grammars which is invariant under unfolding. Define the *rank* of a context-free expression as follows:

$$\boxed{g \Rightarrow w}$$

$$\frac{}{\epsilon \Rightarrow \epsilon}\ \textsc{DEps} \qquad \frac{}{c \Rightarrow c}\ \textsc{DChar} \qquad \frac{g \Rightarrow w \qquad g' \Rightarrow w'}{g \cdot g' \Rightarrow w \cdot w'}\ \textsc{DCat} \qquad \frac{g_1 \Rightarrow w}{g_1 \vee g_2 \Rightarrow w}\ \textsc{D}\vee\textsc{L} \qquad \frac{g_2 \Rightarrow w}{g_1 \vee g_2 \Rightarrow w}\ \textsc{D}\vee\textsc{R}$$

$$\frac{[\mu x : \tau.\ g/x]g \Rightarrow w}{\mu x : \tau.\ g \Rightarrow w}\ \textsc{DFix} \qquad \frac{g \Rightarrow w \qquad w \neq \epsilon}{[g] \Rightarrow w}\ \textsc{DNonempty}$$

Fig. 5. Inference Rules for Language Membership

Definition 3.8. (Rank of a context-free expression)

$$\text{rank}(g) \quad = \quad \begin{cases} 1 + \text{rank}(g') & \text{when } g = \mu x : \tau.\ g' \\ 1 + \text{rank}(g') & \text{when } g = [g'] \\ 1 + \text{rank}(g') + \text{rank}(g'') & \text{when } g = g' \vee g'' \\ 1 + \text{rank}(g') & \text{when } g = g' \cdot g'' \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, the rank of a context-free expression is the size of the subtree within which a guarded variable cannot appear at the front. Since the variables in a fixed point expression $\mu x : \tau.\ g$ are always guarded, the rank of the fixed point will not change when it is unrolled.

Lemma 3.9. *(Rank Preservation) If* $\Gamma; \Delta, x : \tau' \vdash g : \tau$ *and* $\Gamma, \Delta; \cdot \vdash g' : \tau'$*, then* $\text{rank}(g) = \text{rank}([g'/x]g)$.

Proof. This follows from an induction on derivations. Since guarded variables always occur underneath the left-hand side of a concatenation, substituting anything for one will not change the rank of the result. □

Theorem 3.10. *(Unambiguous Parse Derivations) If* $\cdot; \cdot \vdash g : \tau$ *then* $w \in [\![g]\!] \cdot$ *if and only there is a unique derivation* $\mathcal{D} :: g \Rightarrow w$.

Proof. (Sketch) The proof is by a lexicographic induction on the length of w and the rank of g. We then do a case analysis on the shape of g, analysing each case in turn. This proof relies heavily on the semantic soundness of typing. For example, soundness ensures that the interpretation of each branch of $g_1 \vee g_2$ is disjoint, which ensures that at most one of D∨L or D∨R can apply. □

Since this proof is entirely constructive, it already constitutes a (rather inefficient) parsing algorithm. It is possible to make it more efficient by instrumenting the $g \Rightarrow w$ judgement with more information, eventually yielding an abstract machine for parsing in the style of Thielecke (2012, 2014).

However, rather than doing this, we will take another approach.

3.2 Derivatives of Context Free Expressions

Regular expression derivatives are one of the shibboleths of functional programming: if you ask someone to implement a regular expression matcher, and they choose an algorithm based on Brzozowski derivatives (Brzozowski 1964), you have almost surely identified a functional programmer. Derivatives are so attractive to functional programmers because they offer a cleanly algebraic and inductive approach, without a detour into related formalisms like finite automata.

In this section, we will extend the Brzozowski derivative algorithm to handle typed grammars. The two main – indeed, only – questions facing attempts to extend Brzozowski derivatives to context-free expressions are (a)

the question of how to extend the derivative to handle fixed points, and (b) the question of how to handle free variables.

For untyped context-free languages, this is in general quite difficult. Since a variable can stand for any possible grammar, in general it is not possible to symbolically take its derivative. We may attempt to evade this problem by only considering closed grammars. However, this creates a second problem when taking the derivative of a fixed point expression $\mu x.\ g$; if we restrict ourselves to closed terms, then we cannot look structurally at g, because we may encounter the free variable x. However, fixed points can be unfolded, and so we might think that we can avoid this problem by unfolding the fixed point before taking the derivatives. Unfortunately, we now face the problem of *left recursion* – unfolding $\mu x.\ g_0 \vee x \cdot g_1$ gives us $g_0 \vee (\mu x.\ g_0 \vee x \cdot g_1) \cdot g_1$, leaving the very same fixed point at the front of an expression.

Luckily, this is just the kind of issue our type system exists to prevent. Our typing rules ensure that occurrences of a recursive variable are guarded, and hence unfolding a fixed point will always get us closer to something we can make progress on. As a result, the naive strategy does work for *typed* context-free expressions.

In Figure 6, we give the full definition of the derivative, along with some auxiliary functions used in its definition, and the following subsections explain the algorithm and its properties. We explain each of these functions below.

3.2.1 Nullability. The first function $\mathsf{null}_\Gamma(g)$, looks at the syntax of a grammar and returns true or false depending on whether or not g is nullable, and taking the unrestricted context Γ as an additional argument.

It computes this by an inductive examination of the structure of the expression. Some of the cases are straightforward:

$$\begin{array}{lcl} \mathsf{null}_\Gamma(\epsilon) & = & \mathsf{true} \\ \mathsf{null}_\Gamma(c) & = & \mathsf{false} \\ \mathsf{null}_\Gamma(\bot) & = & \mathsf{false} \\ \mathsf{null}_\Gamma(g \vee g') & = & \mathsf{null}_\Gamma(g) \parallel \mathsf{null}_\Gamma(g') \end{array}$$

The null expression ϵ denotes singleton language containing the empty string, so it returns true. Neither the empty expression $\bot$ and the singleton expression c contain the empty string, so they return false. The disjunction $g \vee g'$ denotes the union, so if either grammar is nullable the disjunction will be.

$$\begin{array}{lcl} \mathsf{null}_\Gamma(g \cdot g') & = & \mathsf{false} \\ \mathsf{null}_\Gamma(x) & = & \begin{cases} \tau.\textsc{Null} & \text{when } x : \tau \in \Gamma \\ \text{undefined} & \text{otherwise} \end{cases} \end{array}$$

The case for concatenations is simple, but for an interesting reason. Because the CoreCat requires g to be non-null, we know that a well-typed concatenation cannot contain the empty string. As a result, we can immediately return false. For variables x, we just look up the type variable in Γ. We do not need to consider any variables in Δ, since they will be to the right of a concatenation in well-typed terms, and we never even look inside concatenations. Similar reasoning applies to fixed points:

$$\mathsf{null}_\Gamma(\mu x.\ g) \quad = \quad \mathsf{null}_\Gamma(g)$$

Since the recursion variable is guarded, we can just check the nullability of the body, without needing to track the recursion variable at all.

It is then straightforward to prove that null checking works as expected.

Lemma 3.11. *(Soundness of Null Checking) If* $\Gamma; \Delta \vdash g : \tau$ *then* $\mathsf{null}_\Gamma(g) = \tau.\textsc{Null}$.

Proof. By induction on the typing derivation. □

However, soundness (Theorem 3.5) is not quite strong enough for what we will need. It promises that if the empty string is in the denotation of g, then $\tau.\textsc{Null}$ will be true. We want this to be an if-and-only-if, so we need to prove the converse as well:

Lemma 3.12. *(Correctness of Null Checking) If* $\cdot; \Delta \vdash g : \tau$ *and* $\delta \models \Delta$, *then* $\epsilon \in [\![g]\!]\ \delta \iff \mathsf{null}.(g) = \mathsf{true}$.

Proof. By induction on the typing derivation. □

Note that we only prove these theorems for grammars with no unrestricted free variables. Ultimately we will only take derivatives of completely closed grammars, but permitting guarded variables strengthens the induction hypothesis enough to make the proofs go through.

3.2.2 The Derivative of a Grammar. Next, we will examine the definition of the one-character derivative.

First, we define a few auxiliary functions $g \sqcup g'$, $g \circ g'$, and $\langle g \rangle$, which are "smart constructors" for union, concatenation and nonnullability. The union and concatenation operations check if either argument is $\bot$, and then simplify based on that. The $\langle g \rangle$ smart constructor likewise simplifies if g is either ϵ, or already obviously nonnullable. These are not strictly necessary for correctness, but make reading the constructed derivatives much easier. We also define the concatenation operator $g \star_\Gamma g'$, which concatenates g to g'. It is just the regular concatenation $g \cdot g'$ when g is nonnullable, but returns $([g] \cdot g') \vee g'$ if g is nullable. We use this definition to ensure the typability of concatenation.

Many of the cases of the derivative $\mathbb{D}_c(g)$ are straightforward.

$$\begin{array}{lcl}
\mathbb{D}_c(x) & = & \text{undefined} \\
\mathbb{D}_c(c') & = & \begin{cases} \epsilon & \text{when } c = c' \\ \bot & \text{otherwise} \end{cases} \\
\mathbb{D}_c(\epsilon) & = & \bot \\
\mathbb{D}_c(\bot) & = & \bot \\
\mathbb{D}_c(g \vee g') & = & \mathbb{D}_c(g) \sqcup \mathbb{D}_c(g') \\
\mathbb{D}_c([g]) & = & \mathbb{D}_c(g)
\end{array}$$

The variable case is undefined, following the intuition that we want to consider terms without any leading free variables. The cases for null ϵ, the singleton c, the empty grammar g, the union $g \vee g'$, and the concatenation $g \cdot g'$ all match the definition of the Brzozowski derivative. The derivative of the singleton is empty if it is c, and $\bot$ otherwise. The derivatives of $\bot$ and ϵ are both $\bot$, since they contain no nonempty strings at all. The derivative of the union is the derivative of the unions.

The derivative of $g \cdot g'$ is $\mathbb{D}_c(g) \star_\Gamma g'$, and is actually *simpler* than the original Brzozowski derivative – since typing ensures g is nonempty, we don't need to consider the case when g is null. The derivative of $[g]$ is exactly the same as the derivative of g, since it contains the same nonempty strings.

This leaves the fixed point:

$$\mathbb{D}_c(\mu x.\ g) \quad = \quad [\mu x.\ g/x]\mathbb{D}_c(g)$$

The definition of the fixed point is very straightforward: we take the derivative of the body, and then substitute the fixed point. This is exactly equivalent to taking the derivative of the unfolding of the fixed point, but makes the structural recursion more apparent.

We can now give the correctness properties.

Lemma 3.13. *(Well-Definedness of the Derivative) If* $\cdot; \Delta \vdash g : \tau$ *then:*

- $\mathbb{D}_c(g)$ *is well-defined.*
- *There is a* τ' *such that* $\Delta; \cdot \vdash \mathbb{D}_c(g) : \tau'$ *and* $\tau'.\textsc{Followlast} \subseteq \tau.\textsc{Followlast}$.
- *If* $c \notin \tau.\textsc{First}$, *then* $\mathbb{D}_c(g) = \bot$.

Proof. By induction on the derivation $\cdot; \Delta \vdash g : \tau$. □

This proves that the one-character derivative of a well-typed grammar g is itself well-typed.

As with nullability, we had to strengthen the induction hypothesis to begin with terms containing guarded free variables. However, note that the guarded variables in g had to be moved into the unrestricted context to type the derivative. This is because taking the derivative can strip off leading characters, making guarded variables unguarded. As a result, if we want to iterate the derivative we will need to begin with a closed context-free expression. This is the mathematical sense possessed by the intuition that we defined our derivative over closed grammars. Note also that all of our proofs about closed grammars were simplified by the ability to take a detour through open terms; this is a commonplace in semantics, but relatively uncommon in parsing theory.

We also prove a theorem asserting that that the interpretation of the derivative is exactly the semantic one-character derivative.

Lemma 3.14. *(Correctness of the Derivative) If* $\cdot; \Delta \vdash g : \tau$ *and* $\delta \models \Delta$*, then* $[\![\mathbb{D}_c(g)]\!]\ \delta = \{w \mid c \cdot w \in [\![g]\!]\ \delta\}$.

Proof. By induction on the derivation $\cdot; \Delta \vdash g : \tau$. □

3.2.3 Putting It Together. Now that we have defined the nullability operation and the one-character derivative, we can put them together to define a parsing function. To do so, we first lift the derivative operation to words:

$$\begin{array}{lcl} \mathbb{D}_\epsilon(g) & = & g \\ \mathbb{D}_{c \cdot w}(g) & = & \mathbb{D}_c(\mathbb{D}_w(g)) \end{array}$$

The derivative of the empty string is the identity, and the derivative of the string $c \cdot w$ is just the c-derivative of the w-derivative. To recognise a word, we can just check the nullability of the word derivative.

Theorem 3.15. *(Parsing with derivatives) If* $\cdot; \cdot \vdash g : \tau$ *and* $w \in \Sigma^*$*, then* $w \in [\![g]\!] \iff \mathsf{null}.(\mathbb{D}_w(g)) = \mathsf{true}$.

Proof. By induction on the length of w. □

Observe that checking nullability in the empty context is a purely syntactic, recursive operation on the grammar, and likewise for taking the word derivative.

3.3 Example

Consider the following simple grammar for fully parenthesised arithmetic expressions:

$$E \triangleq \mu x : \tau.\ n \vee v \vee \boxed{(} \cdot x \cdot \boxed{+} \cdot x \cdot \boxed{)}$$

where n denotes numeric literals and v denote variable identifiers. The type of this grammar is

$$\tau \triangleq \left\{ \begin{array}{lcl} \textsc{Null} & = & \mathsf{false} \\ \textsc{First} & = & \{n, (, v\} \\ \textsc{Followlast} & = & \{+\} \end{array} \right\}$$

If we take the string $((1 + 2) + a)$, it will produce the following sequence of derivatives for each of the substrings of $((1 + 2) + a)$:

$$\begin{array}{lcl}
\mathbb{D}_{\epsilon}(E) & = & E \\
\mathbb{D}_{(}(E) & = & E + E) \\
\mathbb{D}_{((}(E) & = & E + E) + E) \\
\mathbb{D}_{((1}(E) & = & +E) + E) \\
\mathbb{D}_{((1+}(E) & = & E) + E) \\
\mathbb{D}_{((1+2}(E) & = &) + E) \\
\mathbb{D}_{((1+2)}(E) & = & +E) \\
\mathbb{D}_{((1+2)+}(E) & = & E) \\
\mathbb{D}_{((1+2)+a}(E) & = &) \\
\mathbb{D}_{((1+2)+a)}(E) & = & \epsilon
\end{array}$$

Notice that the grammar expressions look exactly like the stack in an implementation of LL(1) parsing. Essentially, the derivative operation is encoding the parse stack *in the grammar itself.*

3.4 Implementation Considerations

Even in the case of regular expressions, direct implementations of derivative parsing can be very inefficient. The basic issue is that we need to add disjunctions to track the nondeterminism inherent in parsing concatenations. In fact, a naive implementation which does not carefully quotient by equivalences like $g \vee g \equiv g$ can face *unbounded* memory usage! The fundamental cause of this problem is the interaction between the definition of derivatives for concatenation and disjunction (inlining $\mathbb{D}_c(g) \star. g'$):

$$\begin{array}{lcl}
\mathbb{D}_c(g \cdot g') & = & \mathsf{if}\ \mathsf{null}.(g)\ \mathsf{then}\ g' \sqcup (\langle \mathbb{D}_c(g) \rangle \circ g')\ \mathsf{else}\ \mathbb{D}_c(g) \circ g' \\
\mathbb{D}_c(g \vee g') & = & \mathbb{D}_c(g) \sqcup \mathbb{D}_c(g')
\end{array}$$

As we can see, taking a character derivative of a concatenation potentially introduces a disjunction, and taking the derivative of a union can preserve it. So it seems like we face the potential issue of the number of disjunctions growing exponentially in the length of a word.

Luckily, our type system prevents this problem from occurring. The CoreVee rule requires the branches of an alternative to have *disjoint* First sets, and Lemma 3.13 ensures that the $c-$derivative of a typed grammar g is $\bot$, whenever c is not in the First set. And so the definition of $g \sqcup g'$ ensures that the derivative $\mathbb{D}_c(g \vee g')$ can either be $\mathbb{D}_c(g)$ or $\mathbb{D}_c(g')$. As a result, one of the main sources of inefficiency in derivative parsing is statically ruled out.

In the move from regular expressions to grammars, though, we add one additional source of inefficiency which is *not* immediately ruled out. Consider the derivative of a fixed point:

$$\mathbb{D}_c(\mu x.\ g) \quad = \quad [\mu x.\ g/x]\mathbb{D}_c(g)$$

Here, we have to substitute the whole definition of the fixed point for the variable x. As a result, if the variable occurs several times, the size of the expression can grow by a potentially large multiplicative factor. This issue with recursive definitions also occurs in the implementation of ordinary functional languages, and so we can adopt the same solution: when parsing, we can implement fixed points as *closures*. This permits fixed points to be expanded lazily, and so the main algorithmic sources of inefficiency seem to be accounted for. Ultimately, we would like to study how to derive a parsing automaton from the actions of the derivative, though we leave that for future work.

4 RELAXING THE RESTRICTIONS

The calculus we have presented so far is *expressive*, in that many languages can be parsed by recursive descent; but it can also be rather *inconvenient*, in that many grammars that a programmer might wish to write are ruled

$$
\begin{array}{lcl}
\mathsf{null}_\Gamma(\bot) & = & \mathsf{false} \\
\mathsf{null}_\Gamma(g \vee g') & = & \mathsf{null}_\Gamma(g) \parallel \mathsf{null}_\Gamma(g') \\
\mathsf{null}_\Gamma(\epsilon) & = & \mathsf{true} \\
\mathsf{null}_\Gamma(c) & = & \mathsf{false} \\
\mathsf{null}_\Gamma(g \cdot g') & = & \mathsf{false} \\
\mathsf{null}_\Gamma(\mu x.\, g) & = & \mathsf{null}_\Gamma(g) \\
\mathsf{null}_\Gamma(x) & = & \begin{cases} \tau.\textsc{Null} & \text{when } x : \tau \in \Gamma \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
g \sqcup \bot & = & g \\
\bot \sqcup g' & = & g' \\
g \sqcup g' & = & g \vee g'
\end{array}
$$

$$
\begin{array}{lcl}
g \circ \bot & = & \bot \\
\bot \circ g' & = & \bot \\
g \circ g' & = & g \cdot g'
\end{array}
$$

$$
\begin{array}{lcl}
\langle \bot \rangle & = & \bot \\
\langle \epsilon \rangle & = & \bot \\
\langle g \rangle & = & [g]
\end{array}
$$

$$
g \star_\Gamma g' = \text{if } \mathsf{null}_\Gamma(g) \text{ then } g' \sqcup (\langle g \rangle \circ g') \text{ else } g \circ g'
$$

$$
\begin{array}{lcl}
\mathbb{D}_c(c') & = & \begin{cases} \epsilon & \text{when } c = c' \\ \bot & \text{otherwise} \end{cases} \\
\mathbb{D}_c(\epsilon) & = & \bot \\
\mathbb{D}_c(g \cdot g') & = & \mathbb{D}_c(g) \star .\, g' \\
\mathbb{D}_c(\bot) & = & \bot \\
\mathbb{D}_c(g \vee g') & = & \mathbb{D}_c(g) \sqcup \mathbb{D}_c(g') \\
\mathbb{D}_c(\mu x.\, g) & = & [\mu x.\, g / x]\mathbb{D}_c(g) \\
\mathbb{D}_c([g]) & = & \mathbb{D}_c(g) \\
\mathbb{D}_c(x) & = & \text{undefined}
\end{array}
$$

Fig. 6. Derivatives of Context Free Expressions

out by typing. The two most annoying restrictions are (1) the requirement that context-free expressions be fully left-factored, and (2) forbidding left recursion. What makes these restrictions particularly inconvenient is that in many cases, the process for converting a grammar to adhere to these restrictions is a mechanical application of the equational theory of context-free expressions.

Luckily, what is a problem for a programmer is an opportunity for a programming language designer: since the transformation is mechanical, we can ask the machine to do it for us. Let us look at these two restrictions in turn, to develop an intuition for how we can encode a more permissive condition into our core type system.

4.0.1 Left Factoring. The CoreVee typing rule ensures that a disjunction is only permitted if it is *immediately* apparent that the two branches do not overlap:

$$\frac{\Gamma; \Delta \vdash g : \tau \qquad \Gamma; \Delta \vdash g' : \tau' \qquad \tau \# \tau'}{\Gamma; \Delta \vdash g \vee g' : \tau \vee \tau'} \textsc{CoreVee}$$

The apartness condition $\tau \# \tau'$ ensures that at most one language contains the empty string, and that the First sets of the two languages are disjoint. As a result, we can tell which branch to take with a single character of lookahead.

But the semiring equations for context free expressions tell us that:

$$g \cdot g' \vee g \cdot g'' = g \cdot (g' \vee g'')$$

That is, left-factoring is a semantics-preserving operation. So if our type system can deduce that g' and g'' are immediately disjoint, then appending g to the beginning of both of them should not alter typability.

4.0.2 Left Recursion. The CoreFix rule permits recursive definitions, only when they are guarded.

$$\frac{\Gamma; \Delta, x : \tau \vdash g : \tau'}{\Gamma; \Delta \vdash \mu x : \tau.\ g : \tau} \textsc{CoreFix}$$

A consequence of this rule is that g cannot in general be left-recursive, because a fixed point of the form $\mu x.\ g_0 \vee x \cdot g_1$ will have the problem that for any nontrivial g_0, its First set will overlap with the first set of x itself. However, we do know from the equational theory of context-free expressions that:

$$\mu x.\ g_0 \vee x \cdot g_1 = \mu x.\ g_0 \cdot g_1 *$$

So as long as $g_0 \cdot g_1 *$ is typeable in the core type system, the apparent left recursion is harmless.

4.1 The Elaborating Type System

In Figures 7, 8, 9, we give the definitions of a type system for a more relaxed set of rules. This type system is presented as a typed elaboration algorithm, which both identifies well-typed terms, and compiles them into terms well-typed in the core type system. There are two primary judgements, $\Gamma; \Delta \rhd g : \tau \rightsquigarrow g'$ and $\Gamma; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$, and two auxiliary judgements $x \rhd g \stackrel{\circ}{=} g_0 \parallel x \cdot g_1$ and $\vec{g} \stackrel{\circ}{=} g_0 \bullet \vec{g_1}$.

The $\Gamma; \Delta \rhd g : \tau \rightsquigarrow g'$ judgement is read as "under hypotheses Γ and guarded hypotheses Δ, the context-free expression g elaborates to a core expression g' of type τ." The $\Gamma; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$ judgement is read as "under hypotheses Γ and guarded hypotheses Δ, the disjunction of the list of context-free expressions $\vec{g}$ elaborates to the core expression g' of type τ."

The $x \rhd g \stackrel{\circ}{=} g_0 \parallel x \cdot g_1$ judgement is read as "the context-free expression g is equal to $g_0 \vee x \cdot g_1$." The purpose of this judgement is to separate g into the part g_1 which follows a left-recursive occurrence of the variable x, and the part g_0 which is not left-recursive. The $\vec{g} \stackrel{\circ}{=} g_0 \bullet \vec{g_1}$ judgement is read as "$\bigvee \vec{g}$ equals $g_0 \cdot \bigvee \vec{g_1}$", where $\vec{g}$ is a list of context-free expressions, and $\bigvee \vec{g}$ is the iterated disjunction of these expressions.

What makes these two judgements auxiliary is that they use the untyped equational theory of context-free expressions and do not exploit typing. Instead, they just break apart a context-free expression into pieces and hand them over to the elaborator to continue working on. In programming terms, they are not mutually recursive with the elaborator typing.

Now we will explain the rules of the $\Gamma; \Delta \rhd g : \tau \rightsquigarrow g'$ judgement. The EEps rule is the elaboration rule for the null grammar. Unsurprisingly, it elaborates to the null grammar. Similarly, the EChar rule says that the singleton grammar c elaborates to c, and the EVar rule says that variables elaborate to themselves. (The fact that we elaborate variables to themselves is the basic reason that substitution commutes with elaboration.) All of these rules also output types which match the corresponding rule in the core type system.

The ENonempty rule takes a grammar $[g]$, and elaborates it to $[g']$, where g' is the elaboration of g. As expected it also sends the type τ to $[\tau]$. Similarly, the ECat rule says to elaborate $g_0 \cdot g_1$ by elaborating g_0 to g_0' and g_1 to g_1', and outputting the concatenation $g_0' \cdot g_1'$ if the types are separable. The EVee rule and the EBot rule recursively invoke the left factoring judgement. The EVee rule invokes it with two arguments in the list (the left and right branches of the disjunction), and the EBot rule invokes it with zero arguments, corresponding to the fact that they are the binary and nullary disjunctions respectively.

There are two rules for elaborating fixed points. The EFixR case handles the case where left recursion is not used: it just checks that the body elaborates when the recursion variable is guarded, and then returns the fixed point of the elaborated body.

The EFixL handles the left recursive case. This rule disassembles the input g into $g_0 \vee x \cdot g_1$ using the auxiliary judgement $x \triangleright g \stackrel{\circ}{=} g_0 \parallel x \cdot g_1$, and then checks that g_0 elaborates to g_0' of type τ_0. It also checks that g_1 elaborates to g_1' of type τ_1, but permits using the variables of Δ in an unrestricted way. This is because the whole fixed point elaborates to $\mu x.\ g_0' \cdot g_1'*$, and since g_0' has to be nonempty, all of the variables in Δ are guarded enough to be used freely in g_1'. We check this with the separability conditions $\tau_0 \circledast \tau_1$ (which ensures it is unambiguous when g_0' has finished parsing), and that $\tau_1 \circledast \tau_1$ (which ensures that it is unambiguous how many times g_1' is encountered). We also check that the type annotation is correct (i.e., that the annotation $\tau = \tau_0 \cdot \tau_1*$).

The left recursion judgement $x \triangleright g \stackrel{\circ}{=} g_0 \parallel x \cdot g_1$ works by syntactically decomposing the structure of g, trying to separate the part that begins with x from the part that doesn't. The RVar and RSelf rules handle the case when g is a variable. The RSelf rule says that if g is x, then g_0 is $\bot$ and g_1 is ϵ, since $x = x \cdot \epsilon$. The RVar rule says that if g is some other y, then g_0 is y and g_1 is $\bot$ (note that $x \cdot \bot = \bot$). The RChar and REps rules work similarly to RVar. The RBot rule also works similarly, except that it returns both g_0 and g_1 as $\bot$. If g is a disjunction, then the RVee rule decomposes both branches recursively, and then reassembles them with disjunctions. The RCat rule works by just decomposing the first half g_0 of a concatenation $g_0 \cdot g_1$, and then gluing the second half g_1 to both results.

The RFix rule works by just returning a fixed point as g_0. This is safe, if conservative – it means that we do not permit left recursion inside of a nested fixed point. It is possible to relax this restriction by unfolding the nested fixed point. However, we refrain from doing this since it could potentially lead to explosions in the size of the elaborated grammar.

The RNonempty rule works by elaborating the body of $[g]$ into g_0 and $x \cdot g_1$, and then returning $[g_0]$ and g_1. Here, we exploit one of the conditions on the EFixL rule. Since we know that this rule is invoked only when x is given a non-null type, we come in knowing that $x \cdot g_1$ is already nonnull. Then we know by Lemma 3.7 that $[x \cdot g_1] = x \cdot g_1$.

Now we will discuss the rules of the $\Gamma; \Delta \triangleright \bigvee \vec{g} : \tau \rightsquigarrow g'$ judgement. As the notation suggests, we are trying to elaborate an n-ary disjunction $\bigvee \vec{g}$ into a single expression g'. The FEmpty rule says that the empty list elaborates to $\bot$, which is intuitively motivated by the idea that $\bot$ is a nullary disjunction. The FBot rule asserts that it is safe to discard $\bot$ from the list of clauses, which is motivated by the semiring equation that $g \vee \bot = g$. The FVee rule says that if $g_0 \vee g_1$ is in the list of clauses, it can be broken up and g_0 and g_1 added separately to the list of clauses. This rule is motivated by the associativity of the disjunction $\vee$.

There are two rules for handling concatenations. The first rule, FCat , is the rule that actually does left factoring. It is invoked when every expression in the sequence $\vec{g}$ begins with a common prefix g_0, which is found by using the syntactic left factoring judgement $\vec{g} \stackrel{\circ}{=} g_0 \bullet \vec{g_1}$. It then checks that g_0 elaborates to g_0', and then continues elaborating the disjunction $\vec{g_1}$, only now letting the variables in Δ be used in an unrestricted fashion. The FSplit rule handles the other case, when it is immediately apparent that one of the clauses is disjoint from all of the others. If we have a sequence $\vec{g}, g, \vec{g'}$, and g elaborates to g_0' at type τ_0 and $\vec{g}, \vec{g'}$ elaborates to g_1' at type τ_1, then if τ_0 and τ_1 satisfy the apartness condition, then we can elaborate the whole thing to $g_0' \vee g_1'$. Together, FCat and FSplit turn a list of expressions into a left-factored tree of concatenations alternating with disjunctions.

Finally, there are just two rules in the syntactic left factoring judgement $\vec{g} \stackrel{\circ}{=} g_0 \bullet \vec{g_1}$. The first rule is when $\vec{g}$ is a singleton of the form $g_0 \cdot g_1$, where it returns g_0 and the singleton list g_1. The second rule is when the vector is of the form $g_0 \cdot g, \vec{g}$, and the rest of the elements split with a leading g_0 as well. That is, we split off g_0 when every element of $\vec{g}$ is a concatenation starting with g_0.

It is worth noting that our elaboration algorithm is not the most aggressive possible design: neither left-recursion elimination nor left-factoring are as general as possible. For example, it is always possible to convert a context-free grammar to Greibach normal form (Greibach 1965), in which all nonempty productions begin with a terminal symbol. This in turn ensures that all recursive occurrences are intrinsically guarded. However, the transformation can become intricate, and so we decided that our elaboration algorithm should only do the standard left-recursion elimination transformation.

Similarly, the syntactic factoring judgement as given is not able to tell that $g_0 \cdot g_1$ and $(g_0 \cdot \epsilon) \cdot g_1$ share a common prefix, because g_0 and $g_0 \cdot \epsilon$ are not syntactically identical. This judgement could be made more powerful by making it more aggressive about reassociating concatenations and making more aggressive use of the distributivity of $\vee$ over $(\cdot)$. This would end up resembling the algorithms used to compile pattern matching to decision trees (Krishnaswami 2009; Le Fessant and Maranget 2001), but giving those rules in this paper would add extra rules without adding much extra insight.

4.2 Example

The natural grammar for arithmetic expressions with addition, subtraction, and negation (we choose these since negation and subtraction share an operator) typechecks, and elaborates as follows:

$$\left(\begin{array}{ll} \mu Exp : \tau. & n \\ \vee & \boxed{(} \cdot Exp \cdot \boxed{)} \\ \vee & \boxed{-} \cdot n \\ \vee & \boxed{-} \cdot \boxed{(} \cdot Exp \cdot \boxed{)} \\ \vee & Exp \cdot \boxed{-} \cdot Exp \\ \vee & Exp \cdot \boxed{+} \cdot Exp \end{array}\right) \quad \leadsto \quad \left(\begin{array}{l} \mu Exp : \tau. \ (n \vee \boxed{(} \cdot Exp \cdot \boxed{)} \vee \boxed{-} \cdot (n \vee \boxed{(} \cdot Exp \cdot \boxed{)})) \\ \qquad \cdot ((\boxed{+} \cdot Exp) \vee (\boxed{-} \cdot Exp))* \end{array}\right)$$

Here, the type τ is:

$$\tau \triangleq \{\textsc{Null} = \text{false};\ \textsc{First} = \{(, n, -\};\ \textsc{Followlast} = \{+, -\}\}$$

Essentially, the first four clauses are identified as non-left-recursive, and then are left-factored, and the tails of the left-recursive clauses are also obviously disjoint. So the type system accepts this definition without issue.[3]

4.3 Metatheory of the Elaboration

We now describe the correctness of the elaboration judgement. First, we prove a few basic properties of the syntactic left-factoring judgement and the left-recursion decomposition judgement. In each case we show that the outputs do not grow in size (which we use to establish termination of the elaborator), that they are stable under substitution, and that they are semantics-preserving.

Lemma 4.1. *(Properties of Syntactic Factoring) If $\vec{g} \stackrel{\circ}{=} g_0 \bullet \vec{g_1}$, then:*

(1) *Both g_0 and $\vec{g_1}$ are strictly smaller than $\vec{g}$.*
(2) $\overrightarrow{[\hat{g}/x]g} \stackrel{\circ}{=} [\hat{g}/x]g_0 \bullet \overrightarrow{[\hat{g}/x]g_1}$
(3) $\bigvee \vec{g} = g_0 \cdot \bigvee \vec{g_1}$

[3]Readers concerned about the possibility of left-recursion elimination unexpectedly altering the associativity of binary operators can rest easy: Thielecke (2012) shows how parser actions can be systematically transformed (using CPS) to preserve associativity while eliminating left recursion.

Proof. By induction on the derivation. □

Lemma 4.2. *(Properties of Left Recursion Decomposition) If* $x \rhd g \stackrel{\circ}{=} g_0 \parallel x \cdot g_1$, *then*

(1) *The size of* g_0 *and the size of* g_1 *are each less than or equal to the size of* g.
(2) *If* $x \neq y$ *and* $x \notin \mathrm{FV}(\hat{g})$, *then* $x \rhd [\hat{g}/y]g \stackrel{\circ}{=} [\hat{g}/y]g_0 \parallel x \cdot [\hat{g}/y]g_1$
(3) *If* $x : \tau \in \Delta$ *and* $\neg\tau.\textsc{Null}$ *and* $\mathrm{FV}(g) \in \mathrm{dom}(\Gamma, \Delta)$, *then* $\Gamma; \Delta \models g \equiv g_0 \vee (x \cdot g_1)$.

Proof. By induction on the derivation. □

We can now prove the properties of the elaboration judgement itself. First, we prove that the elaboration algorithm produces context-free expressions which are typeable under the core type system.

Lemma 4.3. *(Type Safety of the Translation)*

- *If* $\Gamma; \Delta \rhd g : \tau \rightsquigarrow g'$, *then* $\Gamma; \Delta \vdash g' : \tau$.
- *If* $\Gamma; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$, *then* $\Gamma; \Delta \vdash g' : \tau$.

Proof. By a mutual induction on the typing derivations. □

Next, we prove the soundness of substitution for the elaborated system. As with substitution for the core type system, we first have to prove two weakening and one transfer lemma. However, this time around we have to prove a version for each of the two elaboration judgements.

Lemma 4.4. *(Weakening and Transfer for the Translation)*

(1) (a) *If* $\Gamma; \Delta \rhd g : \tau \rightsquigarrow g'$, *then* $\Gamma, y : \tau'; \Delta \rhd g : \tau \rightsquigarrow g'$.
    (b) *If* $\Gamma; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$, *then* $\Gamma, y : \tau'; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$.

(2) (a) *If* $\Gamma; \Delta \rhd g : \tau \rightsquigarrow g'$, *then* $\Gamma; \Delta, y : \tau' \rhd g : \tau \rightsquigarrow g'$.
    (b) *If* $\Gamma; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$, *then* $\Gamma; \Delta, y : \tau' \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$.

(3) (a) *If* $\Gamma; \Delta, y : \tau' \rhd g : \tau \rightsquigarrow g'$, *then* $\Gamma, y : \tau'; \Delta \rhd g : \tau \rightsquigarrow g'$.
    (b) *If* $\Gamma; \Delta, y : \tau' \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$, *then* $\Gamma, y : \tau'; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$.

Proof. Just as with the core type system, we prove these lemmas sequentially. Now, however, each property comes in a pair that must be proved mutually for both judgements. □

Now, we can prove the properties of the elaboration algorithm we are actually interested in. First, we show the elaboration algorithm *respects substitution*:

Theorem 4.5. *(Substitution for the Translation)*

(1) *If* $\Gamma; \Delta \rhd g_1 : \tau_1 \rightsquigarrow g_1'$, *then:*
    - *If* $\Gamma, x : \tau_1; \Delta \rhd g : \tau \rightsquigarrow g'$, *then* $\Gamma; \Delta \rhd [g_1/x]g' : \tau \rightsquigarrow [g_1'/x]g'$.
    - *If* $\Gamma, x : \tau_1; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$, *then* $\Gamma; \Delta \rhd \bigvee [g_1'/x]\vec{g} : \tau \rightsquigarrow [g_1'/x]g'$.
(2) *If* $\Gamma, \Delta; \cdot \rhd g_1 : \tau_1 \rightsquigarrow g_1'$, *then:*
    - *If* $\Gamma; \Delta, x : \tau_1 \rhd g : \tau \rightsquigarrow g'$, *then* $\Gamma; \Delta \rhd [g_1/x]g' : \tau \rightsquigarrow [g_1'/x]g'$.
    - *If* $\Gamma; \Delta, x : \tau_1 \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$, *then* $\Gamma; \Delta \rhd \bigvee [g_1'/x]\vec{g} : \tau \rightsquigarrow [g_1'/x]g'$.

Proof. By induction on derivations, just as with the core type system. However, once again we need to prove it mutually inductively for the two elaboration judgements. □

Theorem 4.6. *(Soundness of the Translation) If* $\Gamma; \Delta \rhd g_1 : \tau_1 \rightsquigarrow g_1'$, *then:*

- *If* $\Gamma; \Delta \rhd g : \tau \rightsquigarrow g'$, *then* $\Gamma; \Delta \models g \equiv g'$.

$$\boxed{\Gamma;\Delta \rhd g : \tau \rightsquigarrow g'}$$

$$\frac{x : \tau \in \Gamma}{\Gamma;\Delta \rhd x : \tau \rightsquigarrow x}\ \textsc{EVar} \qquad \frac{}{\Gamma;\Delta \rhd c : \tau_c \rightsquigarrow c}\ \textsc{EChar} \qquad \frac{}{\Gamma;\Delta \rhd \epsilon : \tau_\epsilon \rightsquigarrow \epsilon}\ \textsc{EEps}$$

$$\frac{\Gamma;\Delta \rhd \bigvee g_0 : \tau_0 \rightsquigarrow g_0' \qquad \Gamma,\Delta;\cdot \rhd \bigvee g_1 : \tau_1 \rightsquigarrow g_1' \qquad \tau_0 \circledast \tau_1}{\Gamma;\Delta \rhd g_0 \cdot g_1 : \tau_0 \cdot \tau_1 \rightsquigarrow g_0' \cdot g_1'}\ \textsc{ECat} \qquad \frac{\Gamma;\Delta \rhd \bigvee \cdot : \tau \rightsquigarrow g'}{\Gamma;\Delta \rhd \bot : \tau \rightsquigarrow g'}\ \textsc{EBot}$$

$$\frac{\Gamma;\Delta \rhd \bigvee g_0, g_1 : \tau \rightsquigarrow g'}{\Gamma;\Delta \rhd g_0 \vee g_1 : \tau \rightsquigarrow g'}\ \textsc{EVee}$$

$$\frac{x \rhd g \stackrel{\circ}{=} g_0 \parallel x \cdot g_1 \quad \Gamma;\Delta, x : \tau \rhd g_0 : \tau_0 \rightsquigarrow g_0' \quad \Gamma,\Delta, x : \tau;\cdot \rhd g_1 : \tau_1 \rightsquigarrow g_1' \quad \begin{array}{l}\tau_0 \circledast \tau_1\\ \tau_1 \circledast \tau_1\\ \tau = \tau_0 \cdot \tau_1 *\end{array}}{\Gamma;\Delta \rhd \mu x : \tau.\ g : \tau \rightsquigarrow \mu x : \tau.\ (g_0' \cdot g_1' *)}\ \textsc{EFixL}$$

$$\frac{\Gamma;\Delta, x : \tau \rhd g : \tau \rightsquigarrow g'}{\Gamma;\Delta \rhd \mu x : \tau.\ g : \tau \rightsquigarrow \mu x : \tau.\ g'}\ \textsc{EFixR}$$

Fig. 7. Elaboration: Core Judgement

- *If* $\Gamma;\Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$, *then* $\Gamma;\Delta \models \bigvee \vec{g} \equiv g'$.

Proof. By mutual induction on derivations. □

Theorem 4.7. *(Decidability of Elaboration) For every* Γ, Δ *it is decidable if*

(1) *For every* g, *whether there is a* g' *and* τ *such that* $\Gamma;\Delta \rhd g : \tau \rightsquigarrow g'$.
(2) *For every* $\vec{g}$, *whether there is a* g' *and* τ *such that if* $\Gamma;\Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$.

Proof. This follows by induction on the size of g and $\vec{g}$. For each g and $\vec{g}$, at most one rule applies, and all premises are invoked on smaller arguments. We also make use of the facts that the syntactic factoring and left-recursion decomposition judgements return smaller arguments (and are manifestly structurally terminating themselves). □

As a result of this, we have an algorithmic (indeed, syntax-directed) elaboration algorithm. Furthermore, elaboration respects substitution, and so the more liberal system remains just as compositional as the kernel system. In particular, a programmer remains just as free to use scoped variables in the outer system as in the kernel.

5 FUTURE WORK AND EXTENSIONS

Intersection Grammars. It is well-known that context-free grammars are closed under unions, but not intersections. Surprisingly, there are no problems with adding intersections to the typed grammars of Section 3. We can a term $g \wedge g'$ to form the intersection of two typed expressions with the following interpretation:

$$[\![g \wedge g']\!]\ \gamma = [\![g]\!]\ \gamma \cap [\![g]\!]\ \gamma'$$

$$\boxed{\Gamma; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'}$$

$$\frac{}{\Gamma; \Delta \rhd \bigvee \cdot : \tau_\perp \rightsquigarrow \perp}\ \textsc{FEmpty} \qquad \frac{\Gamma; \Delta \rhd \bigvee \vec{g}, \vec{g'} : \tau \rightsquigarrow g'}{\Gamma; \Delta \rhd \bigvee \vec{g}, \perp, \vec{g'} : \tau \rightsquigarrow g'}\ \textsc{FBot} \qquad \frac{\Gamma; \Delta \rhd \bigvee \vec{g}, g_0, g_1, \vec{g'} : \tau \rightsquigarrow g'}{\Gamma; \Delta \rhd \bigvee \vec{g}, g_0 \vee g_1, \vec{g'} : \tau \rightsquigarrow g'}\ \textsc{FVee}$$

$$\frac{\vec{g} \doteq g_0 \bullet \vec{g_1} \qquad \Gamma; \Delta \rhd g_0 : \tau_0 \rightsquigarrow g_0' \qquad \Gamma, \Delta; \cdot \rhd \bigvee \vec{g_1} : \tau_1 \rightsquigarrow g_1' \qquad \tau_0 \circledast \tau_1}{\Gamma; \Delta \rhd \bigvee \vec{g} : \tau_0 \cdot \tau_1 \rightsquigarrow g_0 \cdot g_1}\ \textsc{FCat}$$

$$\frac{\Gamma; \Delta \rhd g : \tau_0 \rightsquigarrow g_0' \qquad \Gamma; \Delta \rhd \bigvee \vec{g}, \vec{g'} : \tau_1 \rightsquigarrow g_1' \qquad \tau_0 \# \tau_1}{\Gamma; \Delta \rhd \bigvee \vec{g}, g, \vec{g'} : \tau_0 \vee \tau_1 \rightsquigarrow g_0' \vee g_1'}\ \textsc{FSplit}$$

$$\boxed{\vec{g} \doteq g_0 \bullet \vec{g_1}}$$

$$\frac{\vec{g} \doteq g_0 \bullet \vec{g'}}{(g_0 \cdot g, \vec{g}) \doteq g_0 \bullet (g, \vec{g'})} \qquad \frac{}{(g_0 \cdot g) \doteq g_0 \bullet g}$$

Fig. 8. Elaboration: Left Factoring

$$\boxed{x \rhd g \doteq g_0 \parallel x \cdot g_1}$$

$$\frac{x \neq y}{x \rhd y \doteq y : \tau \parallel x \cdot \perp}\ \textsc{RVar} \qquad \frac{}{x \rhd x \doteq \perp \parallel x \cdot \epsilon}\ \textsc{RSelf} \qquad \frac{}{x \rhd \epsilon \doteq \epsilon \parallel x \cdot \perp}\ \textsc{REps}$$

$$\frac{x \rhd g \doteq g_0 \parallel x \cdot g_1}{x \rhd [g] \doteq [g_0] \parallel x \cdot g_1}\ \textsc{RNonempty} \qquad \frac{}{x \rhd c \doteq c \parallel x \cdot \perp}\ \textsc{RChar} \qquad \frac{}{x \rhd \perp \doteq \perp \parallel x \cdot \perp}\ \textsc{RBot}$$

$$\frac{x \rhd g_1 \doteq g_2 \parallel x \cdot g_3}{x \rhd g_1 \cdot g_0 \doteq g_2 \cdot g_0 \parallel x \cdot (g_3 \cdot g_0)}\ \textsc{RCat} \qquad \frac{}{x \rhd \mu y : \tau.\ g \doteq \mu y : \tau.\ g \parallel x \cdot \perp}\ \textsc{RFix}$$

$$\frac{x \rhd g \doteq g_0 \parallel x \cdot g_1 \qquad x \rhd \hat{g} \doteq \hat{g}_0 \parallel x \cdot \hat{g}_1}{x \rhd g \vee \hat{g} \doteq (g_1' \vee \hat{g}_1') \parallel x \cdot (g_0' \vee \hat{g}_0')}\ \textsc{RVee}$$

Fig. 9. Elaboration: Left Recursion

Then, we can give the

$$\frac{\Gamma; \Delta \vdash g : \tau \qquad \Gamma; \Delta \vdash g' : \tau'}{\Gamma; \Delta \vdash g \wedge g' : \tau \wedge \tau'}$$

where the definition of $\tau \wedge \tau'$ can be given as:

$$\tau \wedge \tau' \quad = \quad \left\{ \begin{array}{lcl} \textsc{Null} & = & \tau.\textsc{Null} \ \wedge \tau'.\textsc{Null} \\ \textsc{First} & = & \tau.\textsc{First} \ \cap \tau'.\textsc{First} \\ \textsc{Followlast} & = & \tau.\textsc{Followlast} \ \cap \tau'.\textsc{Followlast} \end{array} \right\}$$

Then, we can extend the definition of nullability and derivatives in the obvious way:

$$\begin{array}{lcl} \mathsf{null}_\Gamma(g \wedge g') & = & \mathsf{null}_\Gamma(g) \wedge \mathsf{null}_\Gamma(g') \\ \mathbb{D}_c(g \wedge g') & = & \mathbb{D}_c(g) \wedge \mathbb{D}_c(g') \end{array}$$

This innocuous-looking feature represents a very large increase in the expressiveness of our language. For example, consider the language:

$$L = \left\{ a^i \cdot b^i \cdot c^i \;\middle|\; i > 0 \right\}$$

This is one of the usual examples of a non-context-free language, but it *can* be expressed with the typed expression

$$\begin{array}{c} [\mu x.\ \epsilon \vee (a \cdot x \cdot b)] \cdot [c*] \\ \wedge \\ [a*] \cdot [\mu x.\ \epsilon \vee (b \cdot x \cdot c)] \end{array}$$

Surprisingly, all of the theorems in Section 3 continue to hold in the presence of intersections, despite the fact that they take us beyond the context-free languages. Because of this expressiveness gain, we left intersections out of the main development. In the future, we think relating the work of Okhotin (2013) on Boolean grammars is likely to be helpful in understanding what is going on, particularly his work on Boolean LL(k) parsing (Okhotin 2011).

Higher-Order Grammars. Since we have a compositional type system for grammars, it is very easy to add support for functions: we can just add function types and lambda-abstractions and applications. Since (a) Boolean rings have most general unifiers (Martin and Nipkow 1989), and (b) the semantic types we introduce for grammars are just a ternary product of boolean algebras, this suggests that an ML-style type discipline for higher-order grammars should be both feasible and effective. However, we leave this for future work.

LR Parsing. The type system in this paper can be seen as a identifying grammars which can be parsed by predictive, top-down methods. The other main style of parsing, the bottom-up LR(k) algorithm, has the attractive property that it recognises precisely the same languages as deterministic pushdown automata (Knuth 1965). It would be interesting to find a notion of type which can characterise LR(k) parsing.

The key issue seems to be the fact that LR parsers can delay parsing decisions for a potentially unbounded number of symbols. For example, the following context-free expression can be parsed by an LR(1) grammar but has no LL(k) grammar for any k:

$$\mu x.\ (a*) \vee a \cdot x \cdot b$$

This recognises the language $\left\{ a^{i+j} b^j \;\middle|\; i, j \in \mathbb{N} \right\}$. When parsing, we do not know how many as belong to the iteration $a*$ and how many belong to belong to the bracketing $a \cdot x \cdot b$ until we have seen all of the bs. As a result, a new notion of type will likely be necessary to characterise the LR style – a task we leave to future work.

6 RELATED WORK

The idea of generalising regular expressions to context-free expressions by adding a least fixed point operator is an idea sufficiently natural that it has been re-invented several times, but which has nevertheless escaped being studied in depth until surprisingly recently. The earliest reference we found is in Salomaa (1973), which describes a construction dubbed the "regular-like languages". This essentially constructs a fixed point by adjoining a fresh letter to the alphabet, and using it as a variable to do a fixed point construction.

Almost two decades later, Leiß (1991) extended Kleene algebra (Kozen 1990) with a least fixed point operator, resulting in essentially the same calculus as our untyped context-free expressions. He also noted that the context-free languages offered a model of his calculus. Subsequently, Ésik and Leiß (2005) showed that the equational theory was strong enough that the transformation of grammars to Greibach normal form could be carried out completely equationally using context-free expressions. More recently, Grathwohl et al. (2014) have given a complete axiomatization of the inequational theory of context free expressions.

All of this work is for untyped (or general) context-free expressions; the idea of restricting CFGs with a type system seems to be a novelty of our approach. Partly, this is because our typed grammars have left the pure theory of Kleene algebra with fixed points, since equations like $g = g \vee g$ do not hold in general, because $g \vee g$ will typically not be a typeable expression. This is reminiscent of how it is possible to give *bidirectional type systems* (Dunfield and Krishnaswami 2013) which directly classify the beta-normal, eta-long forms. This idea suggests that further restricting the core type system could be interesting: for example, to further restrict the alternation of $\vee$ and $\cdot$, or to restrict where ϵ can occur.

The notion of type used in this paper drew critical inspiration from the work of Brüggemann-Klein and Wood (1992). Their paper proves a Kleene-style theorem for the *deterministic regular languages*, which are those regular expressions which can be compiled to state machines without an exponential blowup in the NFA-to-DFA construction. As part of their characterisation, they defined the notion of a Followlast set. It was not remarked upon in that paper, but this represents a compositional alternative to the traditional follow set computation in LL parsing. Since type systems should be defined compositionally, this is the key idea that made it possible to characterise grammars with types.

As an alternative to a type-based approach, Winter et al. (2011) presented their own variant of the context-free expressions. Their formalism was similar to our own and that of Leiß (1991), with the key difference being that their fixed point operator was required to be *syntactically* guarded – every branch in a fixed point expression $\mu x.\ g$ had to begin with a leading alphabetic character. Their goal was the same as ours: to ensure that the Brzozowski derivative (Brzozowski 1964) could be extended to fixed point expressions. Our type system replaces this syntactical constraint with a type-based guardedness restriction on variables. This is a more flexible approach, a fact we take advantage of to elaborate away left-recursion.

Might et al. (2011) also give an algorithm for parsing context-free languages using derivatives. In their approach, they represent context-free grammars as cyclic graphs (aka the "rational trees") over the Kleene algebra of regular expressions, and then (1) implement nullability with a bottom-up dataflow analysis, and (2) implement the derivative using Brzozowski's original algorithm, but making it lazy and memoizing to handle cycles. This algorithm had an exponential worst case, but Adams et al. (2016) subsequently showed that a modification of this algorithm has cubic time complexity, the same as other general parsing algorithms.

Danielsson (2010) presents a dependently-typed variant of this approach in Agda. Instead of representing grammars as regular trees, they are instead represented with as fully coinductive trees. As a result, the expressiveness (and hence computational complexity) of this library is the same as Agda itself. Interestingly, the library uses nullability as a dependent index in essentially the same way that our type system uses nullability. This suggests that the parser combinators could be extend to use a richer set of dependent indices based on our types in order to ensure that parsing is deterministic.

REFERENCES

Michael D. Adams, Celeste Hollenbeck, and Matthew Might. 2016. On the Complexity and Performance of Parsing with Derivatives. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 224–236. DOI: http://dx.doi.org/10.1145/2908080.2908128

Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 197–208. DOI: http://dx.doi.org/10.1145/2500365.2500597

H. Bekič and C. B. Jones (Eds.). 1984. *Programming Languages and Their Definition: Selected Papers of H. Bekič*. LNCS, Vol. 177. Springer-Verlag. DOI: http://dx.doi.org/10.1007/BFb0048933

Anne Brüggemann-Klein and Derick Wood. 1992. Deterministic Regular Languages. In *9th Annual Symposium on Theoretical Aspects of Computer Science (STACS 92)*. 173–184. DOI: http://dx.doi.org/10.1007/3-540-55210-3_182

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. DOI: http://dx.doi.org/10.1145/321239.321249

Nils Anders Danielsson. 2010. Total Parser Combinators. *SIGPLAN Not.* 45, 9 (Sept. 2010), 285–296. DOI: http://dx.doi.org/10.1145/1932681.1863585

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 429–442. DOI: http://dx.doi.org/10.1145/2500365.2500582

Niels Bjørn Bugge Grathwohl, Fritz Henglein, and Dexter Kozen. 2014. Infinitary Axiomatization of the Equational Theory of Context-Free Languages. In *Fixed Points in Computer Science (FICS 2013)*, Vol. 126. 44–55.

Sheila A. Greibach. 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM* 12, 1 (Jan. 1965), 42–52. DOI: http://dx.doi.org/10.1145/321250.321254

Dick Grune and Ceriel J.H. Jacobs. 2007. *Parsing Techniques: A Practical Guide* (2 ed.). Springer Science, New York, NY 10013, USA.

Graham Hutton. 1992. Higher-order functions for parsing. *Journal of Functional Programming* 2, 3 (001 007 1992), 323–343. DOI: http://dx.doi.org/10.1017/S0956796800000411

Clinton L. Jeffery. 2003. Generating LR Syntax Error Messages from Examples. *ACM Trans. Program. Lang. Syst.* 25, 5 (Sept. 2003), 631–640. DOI: http://dx.doi.org/10.1145/937563.937566

Donald E Knuth. 1965. On the translation of languages from left to right. *Information and control* 8, 6 (1965), 607–639.

Dexter Kozen. 1990. On Kleene algebras and closed semirings. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 26–47.

Neelakantan R. Krishnaswami. 2009. Focusing on pattern matching. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*. ACM, New York, NY, USA, 366–378. DOI: http://dx.doi.org/10.1145/1480881.1480927

Neelakantan R. Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 221–232. DOI: http://dx.doi.org/10.1145/2500365.2500588

Fabrice Le Fessant and Luc Maranget. 2001. Optimizing Pattern-Matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press.

Hans Leiß. 1991. Towards Kleene Algebra with Recursion. In *Computer Science Logic (CSL)*. 242–256.

P. M. Lewis, II and R. E. Stearns. 1968. Syntax-Directed Transduction. *J. ACM* 15, 3 (July 1968), 465–488. DOI: http://dx.doi.org/10.1145/321466.321477

Ursula Martin and Tobias Nipkow. 1989. Boolean Unification — The Story So Far. *Journal of Symbolic Computation* 7 (1989), 275–293. Reprinted in C. Kirchner, *Unification*, Academic Press (1990), 437–455.

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 189–195. DOI: http://dx.doi.org/10.1145/2034773.2034801

Alexander Okhotin. 2011. Expressive power of LL(k) Boolean grammars. *Theoretical Computer Science* 412, 39 (2011), 5132–5155. DOI: http://dx.doi.org/10.1016/j.tcs.2011.05.013

Alexander Okhotin. 2013. Conjunctive and Boolean grammars: The true general case of the context-free grammars. *Computer Science Review* 9 (2013), 27–59. DOI: http://dx.doi.org/10.1016/j.cosrev.2013.06.001

François Pottier and Yann Régis-Gianas. 2017. *Menhir Reference Manual*. INRIA. http://gallium.inria.fr/~fpottier/menhir/

Arto Salomaa. 1973. *Formal Languages*. Academic Press.

Hayo Thielecke. 2012. Functional Semantics of Parsing Actions, and Left Recursion Elimination As Continuation Passing. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP '12)*. ACM, New York, NY, USA, 91–102. DOI: http://dx.doi.org/10.1145/2370776.2370789

Hayo Thielecke. 2014. On the semantics of parsing actions. *Science of Computer Programming* 84 (2014), 52 – 76. DOI: http://dx.doi.org/10.1016/j.scico.2013.04.010

Leslie G. Valiant. 1975. General context-free recognition in less than cubic time. *J. Comput. System Sci.* 10, 2 (1975), 308 – 315. DOI: http://dx.doi.org/10.1016/S0022-0000(75)80046-8

Joost Winter, Marcello M Bonsangue, and Jan Rutten. 2011. Context-free languages, coalgebraically. In *International Conference on Algebra and Coalgebra in Computer Science*. Springer, 359–376.

Zoltán Ésik and Hans Leiß. 2005. Algebraically complete semirings and Greibach normal form. *Annals of Pure and Applied Logic* 133 (1-3) (2005), 173–203.