A Typed, Algebraic Approach to Parsing

ANONYMOUS AUTHOR(S)

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 which gives a compositional criterion for identifying those context-free expressions which can be parsed unambiguously by predictive algorithms in the style of recursive descent or LL(1).

Next, we show how these typed grammar expressions can be used to derive a parser combinator library which both guarantees linear-time parsing with no backtracking and single-token lookahead, and which respects the natural denotational semantics of context-free expressions. Finally, we show how to exploit the type information to write a staged version of this library, which produces dramatic increases in performance, even outperforming code generated by the standard parser generator tool ocamlyacc.

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:
Anonymous Author(s). 2017. A Typed, Algebraic Approach to Parsing. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 40 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 seemingly a reasonable 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.

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

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 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 and 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 all well-typed grammars are unambiguous.

  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.
- Next, we describe a parser combinator library for OCaml based upon this type system. This library exposes basically the standard applicative-style API for parser combinators, but it uses higher-order abstract syntax to build a GADT representing well-typed, binding-safe grammars. Having a first-order representation available lets us analyze the grammar to *reject* grammars not typeable in our type system.

  When a parser function is built from this AST, the resulting parsers have extremely predictable performance: they are guaranteed to be linear-time and non-backtracking, using a single token of lookahead. In addition, our API has no purely-operational features to block backtracking, so the simple reading of disjunction as union of languages and sequential composition as concatenation of languages remains valid (unlike in other formalisms like packrat parsers). However, while the performance is predictable, it is also significantly worse than the performance of code generated by conventional parser generators such as ocamlyacc.
- Next, we build a staged version of this library using MetaOCaml. By using staging, we are able to entirely eliminate the abstraction overhead of parser combinator libraries. The grammar type system proves beneficial, as the types give very useful information in guiding the generation of staged code (indeed, the output of staging looks very much like handwritten recursive descent parser code). Our resulting parsers are very fast, typically performing even better than code generated by ocamlyacc. A table-driven parser can be viewed as an interpreter for a state machine, and staging lets us eliminate the interpretive overhead from parsing!

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.

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

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:

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

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

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

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:

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

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) $g \vee g = g$

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

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

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

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

(5) $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$
(6) $g \cdot \epsilon = g$
(7) $(g_1 \vee g_2) \cdot g = (g_1 \cdot g) \vee (g_2 \cdot g)$
(8) $g \cdot (g_1 \vee g_2) = (g \cdot g_1) \vee (g \cdot g_2)$
(9) $g \cdot \bot = \bot \cdot g = \bot$

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.) 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.

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Definition of Types

However, 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.

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:

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

$\textsc{Null}\ (L)$ 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:

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

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. Somewhat asymmetrically, the FOLLOWLAST set is the FOLLOWLAST set of τ_2, merged together with the FOLLOWLAST set of τ_1 and the FIRST set of τ_2 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* $\bigcup_{x \in X} L_x \models \tau$.

PROOF. Most of these properties are straightforward, except for (4), the satisfaction property for language concatenation. Even for this property, the only interesting case is the soundness of FOLLOWLAST , whose proof we sketch below.

Assume that we have languages L and M, types τ and τ', such that $L \models \tau$ and $M \models \tau'$ and $\tau \circledast \tau'$ holds. Now, note that by the definition of satisfaction, FIRST $(M) \subseteq \tau'$.FIRST and that FOLLOWLAST $(L) \subseteq \tau$.FOLLOWLAST , and that the empty string is not in L. Therefore, by the Unique Decomposition lemma, we know that for every word w in $L \cdot M$, there are unique $w_L \in L$ and $w_M \in M$ such that $w_L \cdot w_M = w$.

Now, we want to show that FOLLOWLAST $(L \cdot M) \subseteq (\tau \cdot \tau')$.FOLLOWLAST . To show this, assume that $c \in$ FOLLOWLAST $(L \cdot M)$. So there exists $w \in L \cdot M / \{\epsilon\}$ and $w' \in \Sigma^*$ such that $w \cdot c \cdot w' \in L \cdot M$. Since $w \in L \cdot M$, and so we know that w decomposes into $w_L \in L$ and $w_M \in M$ such that $w_L \cdot w_M = w$ and w_L is nonempty. So we know that $w_L \cdot w_m \cdot c \cdot w' \in L \cdot M$ with w_L nonempty. Now, consider whether w_M is the empty string or not.

If w_M is the empty string, then we know that τ'.NULL must be false. In addition, we know that $w_L \cdot c \cdot w' \in L \cdot M$. By unique decomposition, we know that there is a unique $w'_L \in L$ and $w'_M \in M$ such that $w'_L \cdot w'_M = w_L \cdot c \cdot w'$. Depending on whether w'_M is the empty string or not, we can conclude that either $w'_L = w_L \cdot c \cdot w'$ and $w'_M = \epsilon$, or that $w'_L = w_L$ and $c \cdot w' \in M$ (which follows since we know that c is not in FOLLOWLAST (L)). In the first case, $c \in$ FOLLOWLAST (L), and since τ'.NULL we know $(\tau \cdot \tau')$.FOLLOWLAST $= \tau'$.FOLLOWLAST $\cup\, \tau'$.FIRST $\cup$ τ.FOLLOWLAST . Hence $c \in (\tau \cdot \tau')$.FOLLOWLAST . In the second case, $c \in$ FIRST (M), and again we know $(\tau \cdot \tau')$.FOLLOWLAST $= \tau'$.FOLLOWLAST $\cup\, \tau'$.FIRST $\cup\, \tau$.FOLLOWLAST . So either way, $c \in (\tau \cdot \tau')$.FOLLOWLAST .

If w_M is nonempty, then we know that $w_L \cdot w_M \cdot c \cdot w' \in L \cdot M$, and by unique decomposition we have a $w'_L \in L$ and $w'_M \in M$ such that $w'_L \cdot w'_M = w_L \cdot w_M \cdot c \cdot w'$. We know that w'_L cannot be a prefix of w_L, because otherwise we would violate the unique decomposition property. We also know that w'_L cannot be longer than w_L, because otherwise the first character of w_M would be in FOLLOWLAST (L), which contradicts the property that FOLLOWLAST $(L) \cap$ FIRST $(M) = \emptyset$. Hence $w'_L = w_L$ and $w'_M = w_M \cdot cw'$. Hence $c \in$ FOLLOWLAST (M), which immediately means that $c \in (\tau \cdot \tau')$.FOLLOWLAST .

□

Example. Consider the example of s-expressions:

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

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

Fig. 3. Contexts and Substitutions

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 $\tau.\textsc{Null} = \text{false}$. 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.

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 $\tau_\perp$ 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:*

[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.

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

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

$$\frac{x : \tau \in \Gamma}{\Gamma;\Delta \vdash x : \tau}\ \textsc{CoreVar} \qquad \frac{\Gamma;\Delta, x : \tau \vdash g : \tau'}{\Gamma;\Delta \vdash \mu x : \tau.\ g : \tau}\ \textsc{CoreFix} \qquad \frac{\Gamma;\Delta \vdash g : \tau \qquad \Gamma,\Delta;\cdot \vdash g' : \tau' \qquad \tau \circledast \tau'}{\Gamma;\Delta \vdash g \cdot g' : \tau \cdot \tau'}\ \textsc{CoreCat}$$

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

Fig. 4. Typing for Context-free Expressions

(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. □

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 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.

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

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

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.

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

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

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

Fig. 5. Inference Rules for Language Membership

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, $g \Rightarrow w$ (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 $g \Rightarrow w$ and $g' \Rightarrow w'$, then $g \cdot g' \Rightarrow w \cdot w'$. 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, $c \vee c \Rightarrow c$ 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 $g \Rightarrow w$ 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:

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

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

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* $\mathrm{rank}(g) = \mathrm{rank}([g'/x]g)$.

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, we take a more direct approach.

3.2 Recursive Descent Parsing for Typed Grammars

Since the type system essentially enforces the constraints necessary for predictive parsing, it is possible to read off a parsing algorithm from the structure of a typed context-free expression. In Figure 6, we give a simple algorithm to generate a parser from a typing derivation. This algorithm defines a parsing function $\mathcal{P}\,(\Gamma;\Delta \vdash g : \tau)$ by recursion over the structure of the typing derivation. We define a parser (really, a recognizer) as a partial function on infinite streams of characters ($\Sigma^\omega \rightharpoonup \Sigma^\omega$). (By using infinite streams, we can streamline the correctness proof by avoiding having to give separate cases for empty and nonempty inputs. Note that a finite input sequence can be modelled as a the input string plus an infinite stream of end-of-file EOF characters.)

Since the algorithm is entirely compositional, it can be understood as a species of combinator parsing (indeed, this is how we implement it in OCaml).

3.2.1 *Soundness.*

We write $p \triangleright L$ to indicate that a parser p is *sound* for a language L, which means:

For all w, w'', if $p(w) = w''$ then there is a $w' \in L$ such that $w = w' \cdot w''$.

This means that if a parser takes a stream w as an input, and it returns a stream w'', then the stream w can be divided into a prefix w' and a suffix w'', and that w' is a word in L. We write $p \triangleright_n L$ to constrain this to the subset of L consisting of strings of length n or less.

This lifts to environments in the obvious way. Given an environment $\hat{\gamma}$ of recognizers $(p_1/x_1, \ldots, p_n/x_n)$ and a substitution γ of languages $(L_1/x_1, \ldots, L_n/x_n)$, we can write $\hat{\gamma} \triangleright \gamma$ when $p_i \triangleright L_i$ for each $i \in \{1 \ldots n\}$. The constrained variant $p_i \triangleright_n L_i$ is defined similarly.

Theorem 3.11. *(Soundness of Parsing) For all* n, *derivations* $\Gamma;\Delta \vdash g : \tau$, $\gamma \models \Gamma$, $\delta \models \Delta$, *and that* $\hat{\gamma} \triangleright_n \gamma$ *and* $\hat{\delta} \triangleright_{n-1} \delta$. *Then we have that*

$$\mathcal{P}\,(\Gamma;\Delta \vdash g : \tau)\ \ \hat{\gamma}\ \hat{\delta} \triangleright_n [\![\Gamma;\Delta \vdash g : \tau]\!]\ \ \gamma\ \delta$$

Proof. The proof is by a lexicographic induction on the size of n and the structure of the derivation of $\Gamma;\Delta \vdash g : \tau$. The two most interesting cases are the fixed point $\mu x.\, g$ and the sequential composition $g_1 \cdot g_2$. The fixed point case relies upon the fact that the induction hypothesis tells us that the recursive parser is sound for strings strictly smaller than n to put it into δ. The sequential composition rule relies upon the fact that g_1 recognizes only non-empty strings (i.e., of length greater than 0) to justify combining $\hat{\delta}$ and $\hat{\gamma}$ as the typing rule requires.

□

As a result of this proof, we know that if a parser returns, it has recognized a string in the language of the grammar.

$$\mathcal{P}\,(\Gamma;\Delta \vdash g:\tau) \in \mathrm{Env}(\Gamma) \to \mathrm{Env}(\Delta) \to \Sigma^* \rightharpoonup \Sigma^*$$
$$\mathcal{P}\,(\Gamma;\Delta \vdash \bot:\tau_\bot)\ \hat{\gamma}\ \hat{\delta}\ s = \mathrm{fail}$$
$$\mathcal{P}\,(\Gamma;\Delta \vdash g \vee g':\tau \vee \tau')\ \hat{\gamma}\ \hat{\delta}\ ((c :: \_) \text{ as } s) =$$
$$\begin{cases} \mathcal{P}\,(\Gamma;\Delta \vdash g:\tau)\ \hat{\gamma}\ \hat{\delta}\ s & \text{when } c \in \tau.\textsc{First} \\ & \text{or } \tau.\textsc{Null} \wedge c \notin (\tau \vee \tau').\textsc{First} \\ \mathcal{P}\,(\Gamma;\Delta \vdash g':\tau')\ \hat{\gamma}\ \hat{\delta}\ s & \text{when } c \in \tau'.\textsc{First} \\ & \text{or } \tau'.\textsc{Null} \wedge c \notin (\tau \vee \tau').\textsc{First} \\ \mathrm{fail} & \text{otherwise} \end{cases}$$
$$\mathcal{P}\,(\Gamma;\Delta \vdash c:\tau_c)\ \hat{\gamma}\ \hat{\delta}\ (c' :: s) = \text{if } c = c' \text{ then } s \text{ else fail}$$
$$\mathcal{P}\,(\Gamma;\Delta \vdash \epsilon:\tau_\epsilon)\ \hat{\gamma}\ \hat{\delta}\ s = s$$
$$\mathcal{P}\,(\Gamma;\Delta \vdash g \cdot g':\tau \cdot \tau')\ \hat{\gamma}\ \hat{\delta}\ s =$$
$$\quad \mathcal{P}\,(\Gamma;\Delta \vdash g':\tau')\ \hat{\gamma}\ \hat{\delta}\ (\mathcal{P}\,(\Gamma;\Delta \vdash g:\tau)\ \hat{\gamma}\ \hat{\delta}\ s)$$
$$\mathcal{P}\,(\Gamma;\Delta \vdash x:\tau)\ \hat{\gamma}\ \hat{\delta}\ s = \hat{\gamma}(x)\ s$$
$$\mathcal{P}\,(\Gamma;\Delta \vdash \mu x:\tau.\ g:\tau)\ \hat{\gamma}\ \hat{\delta}\ s =$$
$$\quad \mathit{fix}(\lambda F.\ \mathcal{P}\,(\Gamma;\Delta, x:\tau \vdash g:\tau)\ (\hat{\gamma}\ \hat{\delta}, F/x))\ s$$
$$\mathcal{P}\,(\Gamma;\Delta \vdash [g]:[\tau])\ \hat{\gamma}\ \hat{\delta}\ s =$$
$$\quad \text{let } s' = \mathcal{P}\,(\Gamma;\Delta \vdash g:\tau)\ \hat{\gamma}\ \hat{\delta}\ s \text{ in}$$
$$\quad \text{if } s = s' \text{ then fail else } s'$$

Fig. 6. Parsing Algorithm

3.2.2 Completeness. We write $p \blacktriangleright L$ to indicate that a parser p is *complete* for a language L, which means:

> For all $w \in L$, $c \in \Sigma$ and $w'' \in \Sigma^\omega$ such that $c \notin \textsc{Followlast}\,(L)$ and also $c \notin \textsc{First}\,(L)$ when $\varepsilon \in L$, we have $p(w \cdot c \cdot w'') = c \cdot w''$.

We write $p \blacktriangleright_n L$ to constrain this to the subset of L consisting of strings of length n or less.

Just as with soundness, this lifts to environments in the obvious way. Given an environment $\hat{\gamma}$ of recognizers $(p_1/x_1, \ldots, p_n/x_n)$ recognizes a substitution γ of languages $(L_1/x_1, \ldots, L_n/x_n)$ when $p_i \blacktriangleright L_i$. The constrained variant $p_i \blacktriangleright_n L_i$ is defined similarly.

Theorem 3.12. *(Completeness of Parsing) For all n, derivations* $\Gamma;\Delta \vdash g:\tau$, $\gamma \models \Gamma$, $\delta \models \Delta$, *and that* $\hat{\gamma} \blacktriangleright_n \gamma$ *and* $\hat{\delta} \blacktriangleright_{n-1} \delta$. *Then we have that*

$$\mathcal{P}\,(\Gamma;\Delta \vdash g:\tau)\ \hat{\gamma}\ \hat{\delta} \blacktriangleright_n [\![\Gamma;\Delta \vdash g:\tau]\!]\ \gamma\ \delta$$

Proof. The proof is by a lexicographic induction on the size of n and the structure of the derivation of $\Gamma;\Delta \vdash g:\tau$. As with soundness, sequential composition $g_1 \cdot g_2$ is the interesting case, and requires considering whether g_2 is nullable or not. In this case, we need to argue that that $c \notin \textsc{First}\,([\![g_2]\!]\ \gamma\ \delta)$, which follows because the $\textsc{Followlast}$ set of $[\![g_1 \cdot g_2]\!]\ \gamma\ \delta$ includes the $\textsc{First}$ set of $[\![g_2]\!]\ \gamma\ \delta$ when g_2 is nullable. □

As a result of soundness proof, we know that if the parser succeeds, then it consumed a word of the language. As a result of the completeness proof, we know that if we supply a parser a stream prefixed with a word of the language and a delimiter, then it will consume precisely that word, stopping at the delimiter.

4 IMPLEMENTATION OF THE BASIC COMBINATORS

The theory described in the previous sections describes language *recognizers*, which is just an algorithm for testing language membership. However, what we really want are *parsers*, which also construct a value (such as an abstract syntax tree). In this section, we describe how we can implement these combinators in OCaml.

The main design issue is that our combinators are only guaranteed to be well-behaved if they operate on a grammar which is well-typed with respect to the type system defined in Section 3, and typechecking is easiest to implement on a first-order representation. However, we also want to guarantee that the parsers we generate are also well-typed in the sense that they we know what type of OCaml values they produce.

4.1 Representing Grammars

The first requirement suggests that we use a first-order syntax tree to represent grammars, variables and fixed points, so that we can write a typechecker as a standard tree-traversal. However, matters are complicated by the

need to attach OCaml type information to these abstract syntax trees. First, tree types need to be additionally indexed by the type of the OCaml value the parser will produce. Second, since our language of grammars has binding structure, we need a representation of variables which also remembers the type information each parser is obliged to produce. Below, we give the grammar datatype:

```
type ('ctx, 'a) t =
  | Eps : 'a -> ('ctx, 'a) t
  | Seq : ('ctx, 'a) t * ('ctx, 'b) t -> ('ctx, 'a * 'b) t
  | Chr : char -> ('ctx, char) t
  | Bot : ('ctx, 'a) t
  | Alt : ('ctx, 'a) t * ('ctx, 'a) t -> ('ctx, 'a) t
  | Map : ('a -> 'b) * ('ctx, 'a) t -> ('ctx, 'b) t
  | Fix : ('a * 'ctx, 'a) t -> ('ctx, 'a) t
  | Var : ('ctx, 'a) var -> ('ctx, 'a) t

type ('ctx, 'a) var = Z : ('a * 'ctx, 'a) var | S : ('rest, 'a) var -> ('b * 'rest, 'a) var
```

This datatype is *almost* a plain algebraic datatype, but it is not quite as boring as one might hope: it is a *generalized* algebraic datatype (GADT). The type `('ctx,'a) t` is indexed by a type parameter `'a` which indicates the return type of the parser, and also indexed by a parameter `'ctx`, which is a type denoting the context.

Variables are represented in a basically de Bruijn fashion. Our datatype `('ctx, 'a) var` is basically a dependent Peano number, which says that the *n*-th element of a context of type `'ctx` is a hypothetical parser returning elements of type `'a`. Each of the constructors of this datatype corresponds closely to the productions of the grammar for context-free expressions. The main addition is the `Map` constructor, which wraps a function of type `'a -> 'b` to apply to the inner grammar, thereby taking a grammar of type `('ctx, 'a) t` to one of type `('ctx, 'b) t`. Morally, this is just a representation of a parser action. Binding structure starts to come into play with the `Var` constructor, which says that if the `n`-th component of the context type `'ctx` is the type `'a`, then `Var n` has the type `('ctx, 'a) t`. The fixed point construction `Fix` takes a single argument of type `('a * 'ctx, 'a) t`, with the first component of the context the recursive binding.

Manually building grammars with this representation API is possible, but not especially convenient. The use of an intrinsic de Bruijn representation means that grammars have to make uses of weakening explicit. For example, to implement the Kleene star, one would need to write the following:

```
val weaken : ('ctx, 'a) t -> ('b * 'ctx, 'a) t (* Definition omitted; Sec. 4.4 has a better alternative *)

let rec star : ('a, 'b) t -> ('a, 'b list) t = fun p ->
  let cons (x, xs) = x :: xs in
  Fix(Alt(Eps [],
          Map(cons, Seq(weaken p, Var Z))))
```

Note that explicit call `weaken p` so that the bindings of `p` do not interfere with binding introduced for the recursive fixed point. Furthermore, naive use of combinators to build grammars frequently runs afoul of the value restriction, causing a loss of polymorphism. For example, consider the s-expression building code below:

```
type sexp = Atom of string | Seq of sexp list

let any gs = List.fold_let (fun g g' -> Alt(g,g')) Bot gs
let charset cs = any (List.map (fun c -> Chr c) cs)
let id () = Map(toString, star (charset ['a'; 'b'; 'c'; ...]))
let atom() = Map((fun s -> Atom s), id())

let sexp () : unit -> ('ctx, sexp) t =
  Fix (Alt (atom(),
            Map ((fun ((_,y),_) -> Seq y),
                 Seq (Seq (Chr '(', star (Var Z)), Chr ')'))))
```

In addition to the `star` combinator, the `id` combinator for parsing identifiers uses an n-ary disjunction `any` and a character set combinator `charset`. Because of this, to get enough polymorphism, `id` has to be thunked, and this forces all dependent definitions (such as the `sexp` combinator) to also be thunked.

As we will see in section 4.4, we can use a higher-order abstract syntax representation to ensure the *client* never has to see this complexity.

4.2 Typechecking

The GADT constraints on the grammar datatype ensure that we will have a coherent ML typing for the parsers, but it makes no effort to check that these grammars will satisfy the type discipline described in Section 3. To enforce that, we can write our own typechecker, exploiting the fact that grammars are "just" data.

To implement the typechecker, we represent the types of our type system as an OCaml datatype:

```
type t = { first : CharSet.t; follow : CharSet.t; null : bool; guarded : bool }
```

This is almost the same as the types defined in Section 2. The only difference is the extra field `guarded`, which tracks whether a variable is in the Γ and Δ context. This saves us from having to manage two contexts as GADT indices, which would be rather clunky.

Next, we can define the various operations on types defined in Section 2. The following code is illustrative:

```
let disjoint t1 t2 = not (t1.null && t2.null) && nonoverlapping t1.first t2.first
let check b msg = if b then () else failwith msg

let alt t1 t2 = check (disjoint t1 t2) "disjointness failure";
  { first = CharSet.union t1.first t2.first;
    follow = CharSet.union t1.follow t2.follow;
    null = t1.null || t2.null;
    guarded = t1.guarded && t2.guarded }
```

The `disjoint` function corresponds to the $\tau \# \tau'$ operation which ensures that two languages have no words in common as elements. The `check` function converts a boolean test into either a unit or exception, and the `alt` function checks for disjointness and computes the union $\tau \vee \tau'$. (Our API raises exceptions to avoid having to track errors monadically. Since none of our code ever *catches* an exception, the translation in either direction is completely mechanical.) All the other operations on types are similar, and we can (nearly) define the typechecking algorithm as a homomorphism on the structure of the grammar. To start with, we define a type of contexts, as well as a lookup function:

```
type 'ctx tp_env = []   : unit tp_env
                 | (::) : Tp.t * 'ctx tp_env -> ('a * 'ctx) tp_env

let rec lookup : type ctx a. ctx tp_env -> (ctx, a) var -> tp = fun env x ->
    match env, x with
    | (tp :: rest), Z   -> tp
    | (_ :: rest),  S y -> lookup rest y
```

The `'ctx tp_env` type is just a list of types, and the `lookup` function deconstructs a de Bruijn variable of type `('ctx, 'a) var` until it finds the correct entry. The use of GADTs ensures that the lookup will succeed, but is not essential. However, we use the same technique for generating parsers, and here, the use of a heterogenous list is essential, since each parser in the environment may have a different type. With context lookups and type operations in place, typechecking a grammar is easy:

```
let rec typeof : type ctx a d. ctx tp_env -> (ctx, a) Grammar.t -> Tp.t  = fun env g ->
  match g with
  | Eps v        -> Tp.eps
  | Seq (g1, g2) -> Tp.seq (typeof env g1) (typeof env g2)
  | Chr c        -> Tp.chr c
```

```
  | Bot          -> Tp.bot
  | Alt (g1, g2) -> Tp.alt (typeof env g1) (typeof env g2)
  | Map(f, g')   -> typeof env g'
  | Fix g'       -> let tp = Tp.fix (fun tp -> typeof (tp :: env) g') in
                    Tp.check tp.guarded "guardedness failure";
                    tp
  | Var n        -> lookup env n
```

The `typeof` function just walks over the structure of the grammar, invoking the appropriate type combinators at each step. Since `Tp.seq` and `Tp.alt` check whether it is safe to combine the types and raise an error otherwise, we do not need to do that checking in this traversal. The one exception is in the case for `Fix`. The formal system in Section 3 has a type annotation for fixed points, but we do not require that in our implementation. Since types form a complete lattice, we can perform a fixed point iteration from the least type to infer a type for a fixed point. Once the fixed point is computed, we can then check to see if the resulting type is guarded (as it will be the same as the type of the binding). We should note that our actual implementation is slightly more complicated. Later on, it more convenient if `typeof` *annotates* the subterms of its input with types, rather than just returning the type.

4.3 From Grammars to Parsers

The type we use for parsers is very simple:

```
type 'a t = (char Stream.t -> 'a)
```

This takes a stream of characters, and either returns a value or fails with an exception. The key fact about this type is that it *does not support backtracking*; since we take an imperative stream, we can only step past each character once. This makes the implementation of the basic combinators very easy:

```
let eps v s = v

let seq p1 p2 s = let a = p1 s in
                  let b = p2 s in (a, b)

let chr c s = match Stream.peek s with
              | Some c' when c = c' -> Stream.junk s; Chr c
              | _ -> failwith "Unexpected character"
```

There is hardly anything here; the only marginally interesting case is `chr` combinator, which does a 1-token lookahead to decide whether to consume its input or not. More interesting is the `alt` combinator:

```
let alt tp1 p1 tp2 p2 s = match Stream.peek s with
  | None   -> if tp1.null then p1 s else
              if tp2.null then p2 s else
              error "Unexpected end of stream"
  | Some c -> if      CharSet.mem c tp1.first then p1 s
              else if CharSet.mem c tp2.first then p2 s
              else if tp1.null                then p1 s
              else if tp2.null                then p2 s
              else failwith "No progress possible"
```

This function peeks at the next token, and uses that information together with the statically-known First and Null information to decide whether to invoke `p1` or `p2`. This is slightly more complex than the algorithm in Section 3.2, since OCaml streams are not infinite – they can be empty. We can go from a well-typed grammar to a parser by walking over a (well-typed) grammar and composing combinators. As with type checking, we introduce a type of parse environments, which are heterogenous lists containing parsers. The `lookup` function for `'ctx parse_env` contexts returns a parser of the appropriate type.

```
type 'ctx parse_env = [] : unit parse_env
                    | (::) : 'a t * 'ctx parse_env -> ('a * 'ctx) parse_env
```

```
let rec lookup : type ctx a. ctx parse_env -> (ctx, a) var -> a t = fun env x ->
    match env, x with
    | (p :: rest), Z   -> p
    | (_ :: rest), S y -> lookup rest y
```

Unlike with type checking, the GADT is essential in this case. Every parser in the environment can have a different type, and so the ability of GADTs to represent heterogenous lists is essential. With this machinery in hand, writing the function to build a parser from a grammar is easy:

```
let rec parse : type ctx a d. (ctx, a) Grammar.t -> ctx parse_env -> a t = fun (d, g) penv ->
  match g with
  | Eps v        -> eps v
  | Seq (g1, g2) -> seq (parse g1 penv) (parse g2 penv)
  | Chr c        -> chr c
  | Bot          -> bot ()
  | Alt (g1, g2) -> let p1 = parse g1 penv in
                    let p2 = parse g2 penv in
                    alt (getType g1) p1 (getType g2) p2
  | Map(f, g')   -> let p = parse g' penv in map f p
  | Var n        -> lookup penv n
  | Fix g'       -> let rec p s = parse g' (p :: penv) s in p
```

The `Var` case just looks up a parser in the environment, and the `Fix` case builds a recursive parser with recursion. As defined here, the `Fix` combinator builds a parser which re-traverses the grammar tree each time it is called. This could be avoided by (for example) implementing recursion imperatively using Landin's knot, but in the sequel we will use staging to eliminate this overhead instead. We also use a `getType` function to get the type of a grammar in the `Alt` case; this is why it turns out to be so convenient to work with type-annotated grammars.

4.4 A Higher-Order Interface

While the first-order data type of Section 4 is convenient for typechecking and other analyses, it is less convenient as a programming interface. In order to avoid the need for programmers to deal with the `'ctx` type parameter and the low-level variable implementation we introduce a more conventional high-level parser combinator interface:

```
type 'a t
val eps : 'a -> 'a t
val ( >>> ) : 'a t -> 'b t -> ('a * 'b) t
val chr : char -> 'a t
val bot : 'a t
val ( <|> ) : 'a t -> 'a t -> 'a t
val ( $ ) : 'a t -> ('a -> 'b) -> 'b t
val fix : ('b t -> 'b t) -> 'b t
```

Unlike the data type of the previous section, this interface does not expose an explicit representation of variables. Instead, following the conventions of *higher-order abstract syntax*, the binding operation `fix` accepts a function whose parameter serves as the bound variable. Here is the `sexp` parser once again, written using the higher-order interface; the bound variable `self` names the recursively-defined parser within the definition:

```
let sexp = fix (fun self ->
              atom
          <|> (chr '(' >>> star self >>> chr ')' $ fun ((_,sexps),_) -> Seq sexps))
```

4.5 From higher-order to first-order

While higher-order interfaces are more convenient for programming, first-order representations are more convenient for analyses such as type checking. Following Atkey et al. (2009), we combine the advantages of both

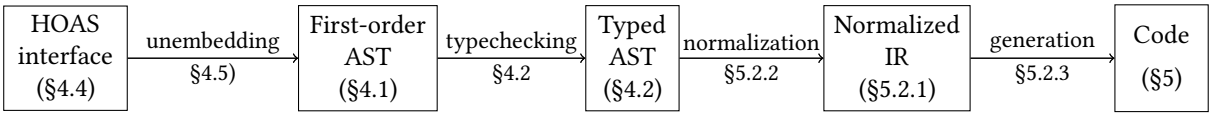

Fig. 7. From combinators to specialized code

approaches by translating the higher-order combinators into the first-order representation. The implementation is a straightforward application of their technique; we implement the interface term `'a t` as a function from a context to a value of the first-order data type of Section 4. In OCaml higher-rank polymorphism, needed here to quantify over the context type `'ctx`, can be introduced using a record type with a polymorphic field:

```
type 'a t = { tdb : 'ctx. 'ctx Ctx.t -> ('ctx, 'a) Grammar.t }
```

The implementation of the non-binding combinators is straightforwardly homomorphic, passing the context argument through to child nodes. For example, here is the conversion from `<|>` in the interface to `Alt` in the representation:

```
let (<|>) f g = { tdb = fun i -> Alt (f.tdb i, g.tdb i) }
```

The only non-trivial case is for `fix`, where the context must be adjusted in the `Var` term passed to the argument:

```
let fix f = let tdb i = Fix ((f {tdb = fun j -> Var (tshift j (S i))}).tdb (S i)) in { tdb }
```

5 ADDING STAGING FOR PERFORMANCE

The typed combinators of Section 4.4 enjoy a number of advantages over both parser combinators and parser generators: unlike most combinator parsers they are guaranteed to run in linear time with no ambiguities; unlike yacc-style parser generators they support binding and substitution. However, they also suffer a notable drawback: they are significantly slower than parsers generated by ocamlyacc. In the benchmarks of Section 6 our combinators perform between 45 and 255 times slower than ocamlyacc.

In this section we address this shortcoming using *multi-stage programming*, a technique for improving the performance of programs by adding annotations that cause their evaluation to take place over several stages. A staged program also receives its input in several stages, and at each stage generates a new program specialized to the available input; in this way the performance penalty of abstraction over arbitrary inputs can be avoided.

The modified version of our combinators described in this section evaluate in two stages. The first stage takes the grammatical structure as input and generates a parser specialized to that grammar that is evaluated in the second stage, when the input string is supplied. In our benchmarks parsers constructed in this way are significantly more efficient than equivalent parsers written using the unstaged combinators of Section 4.4, and even outperform the code generated by ocamlyacc.

Our staged combinators are written in the multi-stage programming language MetaOCaml (Kiselyov 2014), which extends OCaml with two quasiquotation constructs for controlling evaluation: *brackets* `<<e>>` around an expression `e` block its evaluation, while *escaping* a sub-expression `.~e` within a bracketed expression indicates that the sub-expression `e` should be evaluated and the result inserted into the enclosing bracketed expression. Quoted expressions of type `t` have type `t code`, and the typing rules additionally track *levels* (writtten as a superscript over the turnstile) to prevent safety violations such as evaluation of open code:

$$\frac{\Gamma \vdash^{n+} \texttt{e} : \tau}{\Gamma \vdash^{n} \texttt{<<e>>} : \tau\ \texttt{code}}\ \textsc{TBracket} \qquad \frac{\Gamma \vdash^{n} \texttt{e} : \tau\ \texttt{code}}{\Gamma \vdash^{n+} \texttt{.~e} : \tau}\ \textsc{TEscape}$$

5.1 Staging the parser combinator interface

The changes involved in staging the parser combinators are almost entirely internal. Two changes to the interface are needed: the `eps` and `$` combinators that inject and transform the OCaml values returned from the parser now instead work with quoted `code` expressions:

```
val eps : 'a code -> 'a t
val ( $ ) : 'a t -> ('a code -> 'b code) -> 'b t
```

The argument types of `Eps` and `Map` in the first-order representation need similar adjustment.

5.2 Staging the parser combinator implementation

In principle, staging a program is straightforward. After deciding which program inputs are available to each stage, one simply annotates the program accordingly: *dynamic* expressions that depend on inputs that are only available at a later stage must be bracketed, while *static* sub-expressions that only use inputs available already can be escaped or (if at top-level) left unbracketed (Taha 1999, p87). In practice, this naive approach tends to produce poor results, and some additional transformations to the input program are typically needed.

5.2.1 An intermediate representation. Our staged parser combinators make use of an intermediate representation to improve upon the results of naive staging. The intermediate representation has several properties that facilitate generation of efficient code: it distinguishes values from effectful computations, names each sub-term, uses an *n*-ary branching operator that can compile to efficient OCaml pattern-matching code, and ensures that `let`-binding is not nested. The interface to the intermediate form is as follows:

```
type 'a comp
val return : 'a code -> 'a comp
val (>>=) : 'a comp -> ('a code -> 'b comp) -> 'b comp
val fail : string -> 'a comp
val junk : unit comp
val peek : CharSet.t -> ([`Yes | `No | `Eof] -> 'b comp) -> 'b comp
val fix : ('b comp -> 'b comp) -> 'b comp
```

There are six constructors, for injecting a value, sequencing, failure, discarding a token, reading and examining a token, and introducing fixed points.

The monadic sequencing operator `>>=` is a convenience, with no counterpart in the representation of computations; internally the representation uses two binding constructs. The first, `Peek`, examines the next character in the input, and calls one of three possible continuations according to whether the character is a member of the `CharSet.t` argument or whether it is EOF. The second, `Call`, makes a call to another function that is generated from the parser combinators:

```
type 'a comp = Call : 'a reccall * ('a code -> 'b comp) -> 'b comp
             | Junk : 'b comp -> 'b comp
             | Peek : CharSet.t * ([`Eof | `Yes | `No] -> 'b comp) -> 'b comp
             | Fail : string -> 'a comp
             | Return : 'a code -> 'a comp
```

The `>>=` operator then simply combines expressions of the `comp` type, turning a computation `m` in normal form and a computation `k` into a new normal form expression. Here are the cases for `Peek` and `Return`, which respectively correspond to a commuting conversion of `let` and `case` and the beta rule for `let`:

```
let rec (>>=) m k = match m with
  ...
  | Peek (s, k') -> Peek (s, fun v -> k' v >>= k)  (* let x = (case peek of i -> k'[i]) in k[x]
                                                      ~> (case peek of i -> let x = k'[i] in k[x]) *)
  | Return v -> k v                                (* let x = v in k[x] ~> k[v] *)
```

5.2.2 Staging with the IR. The combinators can then be staged by rewriting the parsing code for each operator using the monadic `comp` interface. For example, here is the parsing code for `>>>`, with the unstaged implementation on the left and the staged version on the right:

```
let seq p1 p2 s =
  let a = p1 s in
  let b = p2 s in
  (a, b)
```

```
let seq p1 p2 =
  p1 >>= fun a ->
  p2 >>= fun b ->
  return <<(.~a, .~b)>>
```

5.2.3 From normalized IR to code. The final step in staging is to convert the `comp` representation into `code`. The conversion makes use of a context argument that maintains information known about the various values in the generated program. Here is an excerpt, with fields that maintain information about the next token and the current position in the input stream:

```
type context = { next: [`EOF|`Tok of CharCode.t * char code|`Unknown];
                 index: (int * int code);
                 ... }
```

As the conversion function processes each node in the representation, it transforms the context to accumulate information, and examines the context to improve the generated code. For example, the conversion of a **`Peek`** node to code generates a branch that checks whether a character lies within a set:

```
<< if 'a' <= c && c <= 'z' then ... else ... >>
```

There are two possible interactions with the context in the generation of branches. First, the context is updated before generating the code for the branches: for the **`then`** branch the context records the fact that the character `c` lies *within* the range `'a'..'z'`, and for the **`else`** branch the context records the fact that the character lies *outside* that range. Secondly, the code that generates the condition examines the context to determine whether the condition can be simplified or elided. For example, if `c` is already known not to lie within the range `'a'..'z'` then the whole **`if`**-expression can be replaced i the generated code with the code for the **`else`** branch.

Some calls to `peek` originate from the `chr` combinator, which parses a single character. However, the staged `alt` combinator also calls `peek`, using the computed `first` sets for each operand in the condition, and the computed `null` property to select the continuation. (It is instructive to compare this definition with the unstaged `alt` in Section 4.3; besides the switch to continuation-passing style, a significant difference is that the first sets are now passed to `peek` rather than simply used to test the result.)

```
let alt tp1 p1 tp2 p2 =
  peek tp1.first
    (function `Eof -> if tp.null then p1 else p2
            | `Yes -> p1
            | `No  -> peek tp2.first (function `Eof -> if tp1.null then p1 else p2
                                             | `Yes -> p2
                                             | `No  -> if tp1.null then p1
                                                       else if tp2.null then p2
                                                       else fail "No progress possible!"))
```

In this way the grammar type information passed down into `peek` leads to simplifications in the generated code.

The constraints of the intermediate representation greatly ease analyses of this sort. For example, if **`Peek`** were allowed to appear on the right-hand side of a **`let`** binding then the flow of information through the context would be impeded, since the information obtained from the condition only flows down into the branches, not outwards from the whole **`if`**-expression.

The `index` field in the context enjoys a similar interaction with the normal form; it is often the case during code generation that the current index is known, and need not be recomputed in the generated code; this would rarely be the case without constraints on the form of represented computations.

5.2.4 Recursion. Both the higher-order combinators (Section 4.4) and the interface to the normalized representation (Section 5.2.1) are based on a conventional fixed-point operator, `fix`. In the unstaged parser combinators a call to `fix` constructs a single recursive function, and nested calls to `fix` result in nested recursive functions. A naive staging would produce a similar nested structure in the generated code, but it is preferable for legibility and for compilation to efficient code to flatten the recursive functions into a top-level mutually-recursive group. Our implementation makes use of a recent extension to MetaOCaml that performs precisely that flattening (Yallop and Kiselyov 2018). Section 5.2.5 gives an example of the generated code.

5.2.5 Generated code. While the performance of parser combinators is typically poor, the performance of code generated by our staged combinators is significantly better, as we demonstrate in Section 6. The following generated code, for a minimal s-expression grammar, reveals the reason for the improved performance:

```
let rec f1 i n s = if i >= n then failwith "Expected chr"
                   else let c1 = s.[i] in
                        if 'a' = c1 then ((), (1 + i))
                        else if '(' = c1 then let (x,j) = f2 (1 + i) n s in
                                              if j >= n then failwith "Expected chr"
                                              else let c2 = s.[j] in
                                                   match c2 with ')' -> ((), (1 + j))
                                                               | _ -> failwith "wrong token"
                        else failwith "No progress possible!"
and f2 i n s = if i >= n then ([], i)
               else let c = s.[i] in
                    if ('(' = c) || ('a' = c) then let (x1,j1) = f1 i n s in
                                                   let (x2,j2) = f2 j1 n s in
                                                    ((x1 :: x2), j2)
                    else ([], i) in
fun index s -> let n = String.length s in let (x,j) = f1 index n s in (x, j)
```

The output closely resembles the kind of low-level code one might write by hand, without abstraction overhead, back-tracking, unnecessary tests, or intermediate data structures. There are two mutually-recursive functions, corresponding to the two fixed points in the input grammar, for Kleene star and s-expression nesting. Each function accepts three arguments: the current index, the input length, and the input string. The body of each function examines the input character-by-character, branching immediately and proceeding deterministically. Although the example is minimal, for space reasons, the style of generated code is representative of the much larger examples in Section 6.

5.3 Pipeline summary

Figure 7 summarizes the pipeline that turns higher-order combinators into specialized code. First, the combinators build the first-order representation using Atkey et al.'s unembedding technique. Second, this first-order representation is then typechecked using the type system for context-free expressions. Third, the typechecked representation is converted to normalized form. Finally, the normalized representation is converted to code using MetaOCaml's quasiquotation facilities, and making use of the types constructed in the second phase.

We emphasize that, although the diagram in Figure 7 resembles the structure of a compiler, everything described here is written in user code, as a MetaOCaml library. Typechecking, normalization and other transformations are expressed as OCaml functions operating on the data types used to represent grammars and parsers.

This resemblance to compiler organization is typical of multi-stage programs, where the first stage acts as a compiler, generating specialized code that is executed in a subsequent stage. The analogy between compiler passes (such as typechecking) and staged computation has been drawn several times: Shields et al. (1998) divides type inference into compile-time and run-time phases, much as our GADT representation of grammars is first typechecked by the OCaml compiler and then passed to the grammar typechecker function to check for

ambiguities (Section 4.2). Similarly, Chang et al. (2017) show that macros can be used to build an elaborate type system in user code by running typechecking during macro expansion.

As Figure 7 indicates, the various steps in our library are strictly ordered, without interleaving. In particular, typechecking occurs entirely before code generation. This ordering relies on the fact that the entire grammatical structure is available in advance, which is not the case in traditional monadic parser combinators, where the bind operator allows the structure of the parser to depend on the structure of the input (Hutton and Meijer 1998).

5.4 Generalizing to non-character tokens

We have described the operation of the library on characters. In our implementation, the library is parameterized over the token type, so that it can build parsers for streams of arbitrary tokens. Section 6 describes the use of this parameterization in constructing a parsing program from separate lexer and parser components, each written using our combinators, but instantiated with different token types.

Similarly, although we have described the operation of the library to construct parsers that accept strings, the implementation is parameterized over the input type, and the library can work with a variety of input sources.

6 PERFORMANCE MEASUREMENTS

We illustrate the performance of the staged combinators using parsers for a variety of languages. Each benchmark involves two parsers, one that fills the role of a lexer, turning a sequence of characters into a stream of tokens, and a second that parses the token stream produced by the first. While it would be possible to structure the parser as a single entity, handling lexical structure separately makes it easier to discard whitespace, which may appear at many different locations in the input. Both lexer and parser use the same high-level combinators of Section 5, using the parameterization described earlier: tokens passed to the lexer are characters, while tokens passed to the parser are represented in a way specific to the parsed language. Similarly, the parser operates on OCaml streams, while the lexer can operate on a variety of character sources, including I/O channels, strings, streams and arrays. The benchmarks described here use strings throughout.

In each benchmark the tokens produced by the lexer are represented as a pair of a tag and a value. A tag is a nullary GADT constructor whose index represents the type of the value. For example, here are the tag and token types for s-expressions:

```
type _ tag = ATOM   : string tag
           | LPAREN : unit tag                    type tok = Tok : 'a tag * 'a -> tok
           | RPAREN : unit tag
```

The token type `tok` ensures that the type parameter `'a` of the tag matches the type of the value. The parameter type is existentially hidden in the definition `tok`, not exposed as a parameter, so that tokens with tags of different types can be produced from the same lexer.

The `tag` and `tok` types mediate between the lexer and the parser. Here is the lexer for s-expressions, where each branch produces a value of type `tok option`, with cases for atoms, left and right parentheses, whitespace (which is skipped) and end-of-input:

```
let lex = fix (fun self ->
              (atom
              <|> (chr '(' $ fun _ -> << Some (Tok (LPAREN, ())) >>)
              <|> (chr ')' $ fun _ -> << Some (Tok (RPAREN, ())) >>)
              <|> (whitespace >>> self $ fun x -> << snd .~x >>)
              <|> eps <<None>>)
```

The `atom` and `whitespace` components are built up from simpler components. The `any` combinator provides *n*-ary alternation; it is used to construct parsers `whitespace`, `digit`, `lower`, and `upper`:

```
let any gs = fold_left (<|>) bot gs
let digit = any ['0';'1';'2'; '3'; '4'; '5'; '6'; '7'; '8'; '9']
```

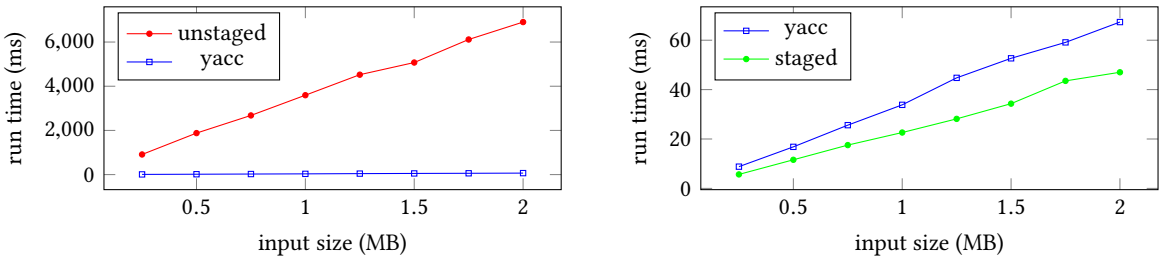

Fig. 8. Parsing S-expressions. Unstaged run-time: ≈100× yacc; staged run-time: ≈0.7× yacc

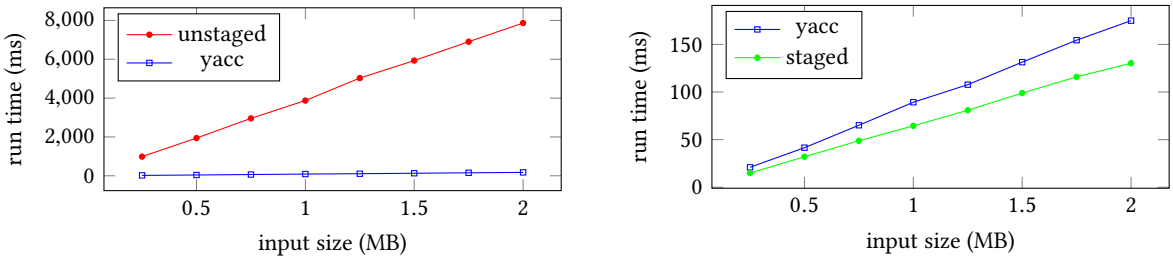

Fig. 9. Parsing integer expressions. Unstaged run-time: ≈45× yacc; staged run-time: ≈0.75× yacc

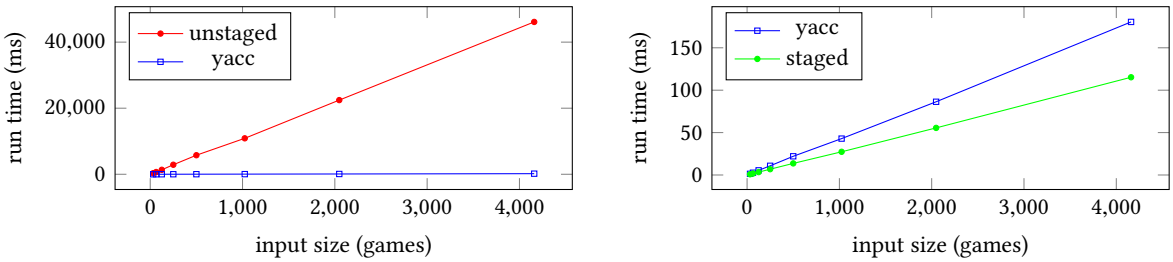

Fig. 10. Parsing PGN chess game description files. Unstaged run-time: ≈255× yacc; staged run-time: ≈0.63× yacc

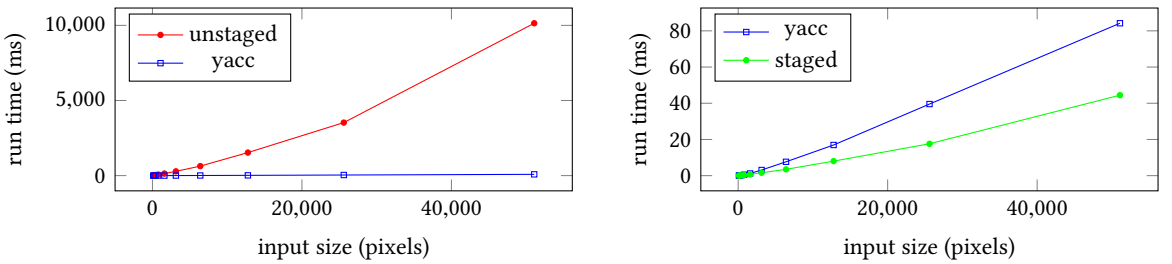

Fig. 11. Parsing PPM images. Unstaged run-time: ≈117× yacc; staged run-time: ≈0.52× yacc

Then `atom` is built from these components using sequencing, alternation, and repetition:

```
let atom = any [lower; upper] >>> star (any [lower; upper; digit])
           $ fun x -> << Some (Tok (ATOM, toString (fst .~x :: snd .~x))) >>)
```

The parser then uses the tokens produced by the lexer in place of input characters:

```
let parse = fix (fun sexp ->
                 (tok LPAREN >>> star sexp >>> tok RPAREN $ fun p -> <<Sexp (snd (fst .~p))>>)
                  <|> (tok ATOM $ fun s -> <<Atom s>>))
```

Figures 8–11 show the running times for staged and unstaged parsers for four languages. The left column compares the running time of an unstaged parser against a parser generated by ocamlyacc; the right column compares the same ocamlyacc parser with a parser generated by the staged combinators described in Section 5. While all the parsers have running time that is linear in the length of the input, there are large performance differences between the different implementations; parsers built from the unstaged combinators are between 45 and 400 times slower than the other two implementations.

For measurement purposes, we avoid the overhead of allocating AST nodes in the benchmarks; instead, our parsers each directly compute some function of the input, such as a count of certain tokens, or a check that some non-syntactic property holds.

The languages are as follows:

(1) S-expressions with alphanumeric atoms (Figure 8). The lexer uses 256 constructors, the parser uses 8 constructors, and there are 3 non-EOF terminals (tokens). The semantic actions of the parser count the number of atoms in each s-expression.
(2) Integer expression language (Figure 9): a miniature programming language with arithmetic, comparison, parenthesization, let-binding and branching. The lexer uses 269 constructors, the parser uses 122 constructors, and there are 13 non-EOF terminals (tokens). The semantic actions of the parser evaluate the expression.
(3) Chess game descriptions in Portable Game Notation format[3] (Figure 10). The lexer uses 842 constructors, the parser uses 78 constructors, and there are 11 non-EOF terminals (tokens). The semantic actions extract the result of each game. We use a corpus of 6759 Grand Master chess games as input.
(4) Image files in PPM format[4] (Figure 11). The lexer uses 580 constructors, the parser uses 10 constructors, and there are 2 non-EOF terminals (tokens). Generates code that validates the non-syntactic constraints of the format, such as colour range and pixel count. We use the POPL 2019 Twitter avatar, scaled to various sizes, as input.

The benchmarks were compiled with BER MetaOCaml N104 and run on an AMD FX 8320 machine with 16GB of memory running Debian Linux. The measurements were taken with the `Core_bench` micro-benchmarking library (Hardin and James 2013).

7 RELATED WORK

The earliest reference we found to the idea of generalizing regular expressions by adding a least fixed point operator 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

[3]https://en.wikipedia.org/wiki/Portable_Game_Notation
[4]https://en.wikipedia.org/wiki/Netpbm_format

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 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 approach both permits parsing with Brzozowski derivatives, and elaborating away left-recursion (see appendix).

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.

Staging and parsers share a long history. In one of the earliest papers on multi-stage programming, Davies and Pfenning (1996) present a staged regular expression matcher as a motivating example. Sperber and Thiemann (2000) apply the closely-related techniques of partial evaluation to build LR parsers from functional specifications. More recently, Jonnalagedda et al. (2014) present an implementation of parser combinators written with the Scala Lightweight Modular Staging framework. The present work shares high-level goals with Jonnalagedda et al's work, notably eliminating the abstraction overhead in standard parser combinator implementations. However, while we focus on deterministic parsers with a type system that guides staging, they focus on ambiguous and input-dependent grammars that are not amenable to the techniques studied here.

Swierstra & Duponcheel (Swierstra and Duponcheel 1996) also gave a parser combinator which statically calculated first sets and nullability to control which branch of an alternative to take. Since they used a higher-order representation of parsers, they were unable to calculate follow sets ahead of time (moreover, follow sets are noncompositional), and so they had to calculate those dynamically, as the parser consumed data. By using GADTs, we are able to give a fully-analyzable first-order representation, which enables us to give us much stronger guarantees about runtime.

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, Sam Lindley, and Jeremy Yallop. 2009. Unembedding Domain-specific Languages. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 37–48. DOI: http://dx.doi.org/10.1145/1596638.1596644

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

Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type Systems As Macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 694–705. DOI: http://dx.doi.org/10.1145/3009837.3009886

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

Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 258–270. DOI: http://dx.doi.org/10.1145/237721.237788

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.

Christopher S. Hardin and Roshan P. James. 2013. Core_bench: micro-benchmarking for OCaml. OCaml Workshop. (September 2013).

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

Graham Hutton and Erik Meijer. 1998. Monadic Parsing in Haskell. *Journal of Functional Programming* 8, 4 (July 1998), 437–444.

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

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 637–653. DOI: http://dx.doi.org/10.1145/2660193.2660241

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

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.

Mark Shields, Tim Sheard, and Simon Peyton Jones. 1998. Dynamic Typing As Staged Type Inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. ACM, New York, NY, USA, 289–302. DOI: http://dx.doi.org/10.1145/268946.268970

Michael Sperber and Peter Thiemann. 2000. Generation of LR Parsers by Partial Evaluation. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 224–264. DOI: http://dx.doi.org/10.1145/349214.349219

S. Doaitse Swierstra and Luc Duponcheel. 1996. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text*. 184–207. DOI: http://dx.doi.org/10.1007/3-540-61628-4_7

Walid Mohamed Taha. 1999. *Multistage Programming: Its Theory and Applications*. Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology. AAI9949870.

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.

Jeremy Yallop and Oleg Kiselyov. 2018. Generating Mutually Recursive Definitions. ACM SIGPLAN Workshop on ML. (September 2018).

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.

A APPENDIX

In these appendices, we show how parsing with derivatives can be extended to well-typed μ-regular expressions, and how to relax our typing rules to permit non-left-factored and left-recursive grammars, while retaining the ability to parse efficiently.

B 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 12, 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.

B.0.1 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.

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

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 B.1. *(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 B.2. *(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.

B.1 The Derivative of a Grammar

First, we will give 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 typeability 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 B.3. *(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 B.4. *(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 $\cdot; \Delta \vdash g : \tau$. □

B.1.1 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 B.5. *(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.

B.2 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

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

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.*

B.3 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 B.3 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

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

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

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

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

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

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

Fig. 12. Derivatives of Context-Free Expressions

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.

C 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 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.

C.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 typeability.

C.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.

C.1 The Elaborating Type System

In Figures 13, 14, 15, 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 $x \rhd g \doteq g_0 \parallel x \cdot g_1$ and $\vec{g} \doteq g_0 \bullet \vec{g_1}$.

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 $x \rhd g \doteq g_0 \parallel x \cdot g_1$ 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 $\vec{g} \doteq g_0 \bullet \vec{g_1}$ 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 $x \rhd g \doteq g_0 \parallel x \cdot g_1$, 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 $x \rhd g \doteq g_0 \parallel x \cdot g_1$ 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 $\Gamma; \Delta \rhd \bigvee \vec{g} : \tau \rightsquigarrow g'$ 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 $\vec{g} \triangleq g_0 \bullet \vec{g_1}$. 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 $\vec{g} \triangleq g_0 \bullet \vec{g_1}$. 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.

C.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:

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

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.[5]

[5]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.

C.3 Allowing Non-Left Factored and Left Recursive Grammars

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 C.1. *(Properties of Syntactic Factoring) If* $\vec{g} \triangleq g_0 \bullet \vec{g_1}$*, then:*

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

Proof. By induction on the derivation. □

Lemma C.2. *(Properties of Left Recursion Decomposition) If* $x \rhd g \triangleq g_0 \parallel x \cdot g_1$*, 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* $x \rhd [\hat{g}/y]g \triangleq [\hat{g}/y]g_0 \parallel x \cdot [\hat{g}/y]g_1$
(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 C.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 C.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 C.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 C.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* $\Gamma; \Delta \models g \equiv g'$.
- *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 C.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.

D 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:

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

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[6]:

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

With a functional representation of streams, this is also very easy to parse:

$$\begin{array}{rcl} \mathcal{P}\ (\Gamma; \Delta \vdash g_1 \wedge g_2 : \tau_1 \wedge \tau_2)\ \hat{\gamma}\ \hat{\delta}\ s & = & \mathsf{let}\ s_1 = \mathcal{P}\ (\Gamma; \Delta \vdash g_1 : \tau_1)\ \hat{\gamma}\ \hat{\delta}\ s\ \mathsf{in} \\ & & \mathsf{let}\ s_2 = \mathcal{P}\ (\Gamma; \Delta \vdash g_2 : \tau_2)\ \hat{\gamma}\ \hat{\delta}\ s\ \mathsf{in} \\ & & \mathsf{if}\ s_1 = s_2\ \mathsf{then}\ s_1\ \mathsf{else}\ \bot \end{array}$$

[6]The Followlast set is indeed an overapproximation, but necessary for greedy combinator parsing to work.

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

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

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

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

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

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

Fig. 13. Elaboration: Core Judgement

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

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

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

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

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

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

Fig. 14. Elaboration: Left Factoring

$$\boxed{x \rhd g \mathrel{\dot{=}} g_0 \parallel x \cdot g_1}$$

$$\frac{x \neq y}{x \rhd y \mathrel{\dot{=}} y : \tau \parallel x \cdot \bot}\ \textsc{RVar} \qquad \frac{}{x \rhd x \mathrel{\dot{=}} \bot \parallel x \cdot \epsilon}\ \textsc{RSelf} \qquad \frac{}{x \rhd \epsilon \mathrel{\dot{=}} \epsilon \parallel x \cdot \bot}\ \textsc{REps}$$

$$\frac{x \rhd g \mathrel{\dot{=}} g_0 \parallel x \cdot g_1}{x \rhd [g] \mathrel{\dot{=}} [g_0] \parallel x \cdot g_1}\ \textsc{RNonempty} \qquad \frac{}{x \rhd c \mathrel{\dot{=}} c \parallel x \cdot \bot}\ \textsc{RChar} \qquad \frac{}{x \rhd \bot \mathrel{\dot{=}} \bot \parallel x \cdot \bot}\ \textsc{RBot}$$

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

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

Fig. 15. Elaboration: Left Recursion

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

$$[\mu x.\ \epsilon \vee (a \cdot x \cdot b)] \cdot [c*]$$
$$\wedge$$
$$[a*] \cdot [\mu x.\ \epsilon \vee (b \cdot x \cdot c)]$$

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.