A Typed, Algebraic Approach to Parsing

Neel Krishnaswami
University of Cambridge
nk480@cl.cam.ac.uk

Jeremy Yallop
University of Cambridge
jeremy.yallop@cl.cam.ac.uk

Abstract

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.

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 linear-time algorithms for parsing unambiguous grammars efficiently [Knuth 1965; Lewis and Stearns 1968] and with excellent error reporting for bad input strings [Jeffery 2003].

However, in languages with good support for higher-order functions (such as ML, Haskell, and Scala) it is very popular to use *parser combinators* [Hutton 1992] instead. These libraries typically build backtracking recursive descent parsers by composing higher-order functions. This allows building up parsers with the ordinary abstraction features of the programming language (variables, data structures and functions), which is sufficiently beneficial that these libraries are popular in spite of the lack of a clear declarative reading of the accepted language and bad worst-case performance (often exponential in the length of the input).

Naturally, we want to have our cake and eat it too: we want to combine the benefits of parser generators (efficiency and a clear declarative reading) with the benefits of parser combinators (ease of building high-level abstractions). The two main difficulties are that the binding structure of BNF (a collection of globally-visible, mutually-recursive nonterminals) is at odds with the structure of programming languages (variables are lexically scoped), and that generating efficient parsers requires static analysis of the input grammar.

This paper begins by recalling the old observation that the context-free languages can be understood as the extension of regular expressions with variables and a least-fixed point operator (aka the "μ-regular expressions" [Leiß 1991]). These features replace the concept of nonterminal from BNF, and so facilitate embedding context-free grammars into programming languages. Our contributions are then:

- First, we extend the framework of μ-regular expressions by giving them a *semantic* notion of type, building on Brüggemann-Klein and Wood's characterization of unambiguous regular expressions [1992]. 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 prove that this type system is well-behaved (i.e., syntactic substitution preserves types, and sound with respect to the denotational semantics), and that all well-typed grammars are unambiguous.
- Next, we describe a parser combinator library for OCaml based upon this type system. This library exposes basically the standard applicative-style API with a higher-order fixed point operator but internally it builds a first-order representation of well-typed, binding-safe grammars. Having a first-order representation available lets us *reject* untypeable grammars.
  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

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314625

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 remains worse than the performance of code generated by conventional parser generators such as ocamlyacc.
- Finally, we build a staged version of this library using MetaOCaml. Staging lets us 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 outperforming even ocamlyacc. Table-driven parsers can be viewed as interpreters for a state machine, and staging lets us eliminate this interpretive overhead!

2 API Overview and Examples

Before getting into the technical details, we show the user-level API in our OCaml implementation, and give both examples and non-examples of its use.

2.1 The High Level Interface

From a user perspective, we offer a very conventional combinator parsing API. We have an abstract type `'a t` representing parsers which read a stream of tokens and produce a value of type `'a`. Elements of this abstract type can be constructed with the following set of constants and functions:

```
type 'a t
val eps : unit t
val chr : char -> char t
val seq : 'a t -> 'b t -> ('a * 'b) t
val bot : 'a t
val alt : 'a t -> 'a t -> 'a t
val fix : ('b t -> 'b t) -> 'b t
val map : ('a -> 'b) -> 'a t -> 'b t
```

Here `eps` is a parser matching the empty string; `chr c` is a parser matching the single character `c`[1]; and `seq l r` is a parser that parses strings with a prefix parsed by `l` and a suffix parsed by `r`. The value `bot` is a parser that never succeeds, and `alt l r` is a parser that parses strings parsed either by `l` or by `r`. Finally we have a fixed point operator `fix` for constructing recursive grammars[2]. The API is also functorial: the expression `map f p` applies a user-supplied function `f` to the result of the parser `p`.

This API is basically the same as the one introduced by Swierstra and Duponcheel [1996], with the main difference that our API is in the "monoidal" style (we give sequencing the type `seq : 'a t -> 'b t -> ('a * 'b) t`), while theirs is in the "applicative" style (i.e., `seq : ('a -> 'b) t -> 'a t -> 'b t`). However, as McBride and Paterson [2008] observe, the two styles are equivalent.

These combinators build an abstract grammar, which can be converted into an actual parser (i.e., a function from streams to values, raising an exception on strings not in the grammar) using the `parser` function.

```
exception TypeError of string
val parser : 'a t -> (char Stream.t -> 'a)
```

Critically, however, the `parser` function *does not* succeed on all abstract grammars: instead, it will raise a `TypeError` exception if the grammar is ill-formed (i.e., ambiguous or left-recursive). As a result, `parser` is able to provide a stronger guarantee when it does succeed: if it returns a parsing function, that function is *guaranteed* to parse in linear time with single-token lookahead.

2.2 Examples

General Utilities One benefit of combinator parsing is that it permits programmers to build up a library of abstractions over the basic parsing primitives. Before showing how this works, we will define a few utility functions and operators to make the examples more readable. The expression `always x` is a constant function that always returns `x`:

```
let always x = fun _ -> x
```

We define `(++)` as an infix version of the `seq` function, and `(==>)` as an infix `map` operator to resemble semantic actions in traditional parser generators:

```
let (++) = seq
let (==>) p f = map f p
```

We also define `any`, an n-ary version of the binary alternative `alt`, using a list fold:

```
val any : 'a t list -> 'a list t
let any gs = List.fold_left alt bot gs
```

The parser `option r` parses either the empty string or alternatively anything that `r` parses:

```
val option : 'a t -> 'a option t
let option r =  any [eps ==> always None;
                     r   ==> (fun x -> Some x)]
```

The operations can be combined with the fixed point operator to define the Kleene star function. The parser `star g` parses a list of the input parsed by `g`: either the empty list, or an instance of `g` (the head of the list) and `star g` (the tail) again recursively. Naturally, we can also define the pure transitive closure `plus` as well.

```
val star : 'a t -> 'a list t
let star g = fix (fun rest -> any [
                   eps       ==> always [];
                   g ++ rest ==> (fun (x, xs) -> x :: xs) ])

val plus : 'a t -> 'a list t
let plus g = g ++ star g ==> (fun (x, xs) -> x :: xs)
```

[1] The API in this paper is slightly simplified, as our real implementation is parameterized over token types, rather than baking in the `char` type.

[2] We are consdering extending `fix` in the style of Yallop and Kiselyov [2019] to make defining mutually-recursive grammars more convenient.

Since this gives us the API of regular expressions, we can use these operations to define, for example, lexers and tokenizers:

```
type token = SYMBOL of string | LPAREN | RPAREN

val charset : string -> char t
let charset s = any (List.map chr (list_of_string s))

let lower = charset "abcdefghijklmnopqrstuvwxyz"
let upper = charset "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
let word = upper ++ star lower
             ==> (fun (c,cs) -> string_of_list (c ::cs))
let symbol = word    ==> (fun s -> SYMBOL s)
let lparen = chr '(' ==> always LPAREN
let rparen = chr ')' ==> always RPAREN

let token = any [symbol; lparen; rparen]
```

Here `charset s` is a parser that accepts and returns any of the characters in the string `s`, `upper` and `lower` parse any single upper- or lower-case letter respectively, and `word` an upper-case letter followed by any number of lowercase letters. The `symbol`, `lparen` and `rparen` parsers respectively accept a word, a left parenthesis and a right parenthesis, and each returns a corresponding value of type `token`.

Recognizing S-Expressions However, the main interest of parser combinators lies in their ability to handle context-free languages such as s-expressions. Below, we give a simple combinator grammar parsing s-expressions, in which an s-expression is parsed as either a symbol, or a list of s-expressions between a left and right parenthesis.

```
type sexp = Sym | Seq of sexp list

let paren p = lparen ++ p ++ rparen ==> fun ((_, x), _) -> x

let sexp = fix (fun sexp -> any [
                  symbol            ==> always Sym;
                  paren (star sexp) ==> (fun s -> Seq s) ])
```

The `symbol` case is wrapped in a `Sym` constructor, and the `paren` case ignores the data for the left and right parenthesis and wraps the list `s` of s-expressions in a `Seq` constructor.

Infix Expressions One common criticism of techniques based on recursive descent is that they make it difficult to handle naturally left-recursive grammars such as arithmetic expressions without awkward grammar transformations. However, with combinators it is possible to package up these transformations once-for-all.

The `infixr` combinator takes a parser recognizing infix operators (returning an operator function of type `'a -> 'a -> 'a`) and a base parser (returning elements of type `'a`), and produces a right-associative parser for that operator class:

```
val infixr : ('a -> 'a -> 'a) t -> 'a t -> 'a t
let infixr op base =
  fix (fun g ->
    base ++ option (op ++ g) ==> function
    | (e, None) -> e
    | (e, Some(f, e')) -> f e e')
```

This works by parsing a base phrase, and then checking to see if an operator function and subexpression follows. If it does not, then the base expression is returned and otherwise the operator function `f` is applied to combine the two subexpressions. Likewise, it is possible to define a combinator `infixl` to generate parsers for left-asssociative operators.

```
val infixl : ('a -> 'a -> 'a) t -> 'a t -> 'a t
let infixl op base =
  let reassociate (e, oes) =
    List.fold_left (fun e (f, e') -> f e e') e oes
  in base ++ star (op ++ base) ==> reassociate
```

This combinator uses one of the classical techniques for handling left-associative operators in recursive descent—it parses an expression `e1 ^ e2 ^ e3 ^ e4` into a head `e1` and a list of pairs `[(^, e2); (^, e3); (^, e4);]`, and then uses a left fold to associate the subterms as `(((e1 ^ e2) ^ e3) ^ e4)`.

These two functions can be used to build a function `infix`, which takes a list of associativities and operator parsers ordered by precedence, and returns a parser for the whole grammar.

```
type assoc = Left | Right

val infix : (assoc * ('a -> 'a -> 'a) t) list -> 'a t -> 'a t
let infix aos base =
  let make_level base = function
    | Left,  op -> infixl op base
    | Right, op -> infixr op base
  in List.fold_left make_level base aos
```

Below, we give a simple expression parser which defines arithmetic expressions. The base expressions are either numeric literals, or fully-parenthesized expressions, and these are fed to the `infix` operator along with a list of associativities and operator parsers.

```
val num : float t (* definition omitted for space *)

let arith = fix (fun arith ->
  infix [(Left,  chr '+' ==> always Float.add);
         (Left,  chr '*' ==> always Float.mul);
         (Right, chr '^' ==> always Float.pow)]
    (any [num; paren arith]))
```

Non-Examples A key feature of our parser combinator library is that it *rejects* certain expressions. The following expressions are all elements of `'a t`, which our `parse` function will reject with a `TypeError` exception.

```
let bad1 = any [chr 'a' ==> always 1;
                chr 'a' ==> always 2]
```

The `bad1` definition is rejected because it suffers from *disjunctive non-determinism*. Given the string `"a"`, it could legitimately return either 1 or 2, depending on whether the first or second alternative is taken.

```
let bad2 = (option (chr 'a')) ++ (option (chr 'a'))
```

The `bad2` definition is rejected because it has *sequential non-determinism*. It is ambiguous whether the parser should return `(Some 'a', None)` or `(None, Some 'a')` when parsing the string `"a"`.

```
let bad3 p = fix (fun left -> any [
      eps          ==> always [];
      (left ++ p) ==> (fun (xs, x) -> List.append xs [x]) ])
```

The function `bad3` is an alternative definition of the Kleene star that is *left-recursive*. This definition would make recursive descent parsers loop indefinitely, so our library rejects it.

```
let bad4 = any [chr 'a' ++ chr 'b' ==> always 1;
                chr 'a' ++ chr 'c' ==> always 2]
```

The `bad4` definition is not ambiguous, but it is also not *left-factored*. It is consequently unclear whether to take the left or the right branch with only a single token of lookahead.

3 Context-Free Expressions

We begin the formal development by defining the grammar of *context-free expressions* in Figure 1. Given a finite set of characters Σ and a countably infinite set of variables V, the context-free expressions are $\bot$, denoting the empty language; the expression $g \vee g'$, denoting the language which is the union of the languages denoted by g and g'; the expression 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'; 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. 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 $g*$ since it is definable by means of a fixed point: $g* \triangleq \mu x.\, \epsilon \vee g \cdot x$ (as indeed we saw in the examples).

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 the singleton set containing the one-character string c drawn from the alphabet Σ; and the meaning of $g \cdot g'$ is those strings formed from a prefix drawn from g and a suffix drawn from g'. 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 3.1. *The context-free expressions satisfy the equations of an idempotent semiring with $(\vee, \bot)$ as addition and its unit, and $(\cdot, \epsilon)$ as the multiplication. In addition, fixed points satisfy the following equations:*

$$\mu x.\, g = [\mu x.\, g/x]g \qquad \mu x.\, g_0 \vee x \cdot g_1 = \mu x.\, g_0 \cdot g_1*$$

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

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

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

Figure 1. Syntax and semantics of context-free expressions

The semiring equations are all standard. The first fixed point equation is the standard unrolling equation for fixed points. The second equation is familiar to anyone who has implemented a recursive descent parser: it is the rule for left-recursion elimination. Leiß [1991] proves 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) as well. Languages form a lattice with a partial order given by set inclusion, and this lifts to environments in the expected way: if $\gamma = (L_1/x_1, \ldots, L_n/x_n)$ and $\gamma' = (L'_1/x_1, \ldots, L'_n/x_n)$, we can write $\gamma \subseteq \gamma'$ if $L_i \subseteq L'_i$ for all i in $1 \ldots n$. This lets us show that the interpretation of grammars is also monotone in its free variables.

Proposition 3.2. *(Monotonicity of μ-regular expressions) If $\gamma \subseteq \gamma'$ then $[\![g]\!]\,\gamma \subseteq [\![g]\!]\,\gamma'$.*

The context-free expressions are equivalent in expressive power to Backus-Naur form. Their main difference 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. 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 algorithms for general context-free grammars (e.g., Earley or CYK) have superlinear worst-case complexity. Furthermore parsing may be *ambiguous*, with multiple possible parse trees for the same string. However, it is well-known that grammars falling into more restrictive classes, such as the LL(k) and LR(k) classes, can be parsed efficiently in linear time. In this paper, we give a type system for grammars parseable by recursive descent.

Types for Languages There are two main 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

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

$$\begin{array}{lcl}
\tau_1 \circledast \tau_2 & \triangleq & \tau_1.\textsc{FLast} \cap \tau_2.\textsc{First} = \emptyset \wedge \neg\tau_1.\textsc{Null} \\
\tau_1 \# \tau_2 & \triangleq & (\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset) \wedge \neg(\tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null}) \\
b \Rightarrow S & \triangleq & \text{if } b \text{ then } S \text{ else } \emptyset
\end{array}$$

$$\tau_\bot \quad = \quad \{\textsc{Null} = \text{false};\ \textsc{First} = \emptyset;\ \textsc{FLast} = \emptyset\}$$

$$\tau_1 \vee \tau_2 \quad = \quad \begin{cases} \textsc{Null} & = & \tau_1.\textsc{Null} \vee \tau_2.\textsc{Null} \\ \textsc{First} & = & \tau_1.\textsc{First} \cup \tau_2.\textsc{First} \\ \textsc{FLast} & = & \tau_1.\textsc{FLast} \cup \tau_2.\textsc{FLast} \end{cases}$$

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

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

$$\begin{array}{lcl}
\tau_\bot \cdot \tau & = & \tau_\bot \\
\tau \cdot \tau_\bot & = & \tau_\bot
\end{array}$$

$$\tau_1 \cdot \tau_2 \quad = \quad \begin{cases} \textsc{Null} & = & \tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null} \\ \textsc{First} & = & \begin{pmatrix} \tau_1.\textsc{First} \cup \\ \tau_1.\textsc{Null} \Rightarrow \tau_2.\textsc{First} \end{pmatrix} \\ \textsc{FLast} & = & \begin{pmatrix} \tau_2.\textsc{FLast} \cup \\ \tau_2.\textsc{Null} \Rightarrow (\tau_2.\textsc{First} \cup \tau_1.\textsc{FLast}) \end{pmatrix} \end{cases}$$

$$\tau* \quad = \quad \begin{cases} \textsc{Null} & = & \text{true} \\ \textsc{First} & = & \tau.\textsc{First} \\ \textsc{FLast} & = & \tau.\textsc{FLast} \cup \tau.\textsc{First} \end{cases}$$

Figure 2. Definition of Types

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 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:

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

$\textsc{Null}(L)$ 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 $\textsc{FLast}(L)$ set are the set of characters which can follow the last character of a string in L. (The FLast set is used as an alternative to the Follow sets traditionally used in LL(1) parsing.)

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

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

This ensures the type τ is an overapproximation of the properties of L. What makes this notion of type useful are the following two lemmas.

Lemma 3.3. *(Disjunctive unique decomposition) Suppose L and M are languages. If $\textsc{First}(L) \cap \textsc{First}(M) = \emptyset$ and $\neg(\textsc{Null}(L) \wedge \textsc{Null}(M))$, then $L \cap M = \emptyset$.*

Lemma 3.4. *(Sequential unique decomposition) Suppose L and M are languages. If $\textsc{FLast}(L) \cap \textsc{First}(M) = \emptyset$ and $\neg\textsc{Null}(L)$ and $w \in L \cdot M$, then there are 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. So we can 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 $L \cdot M$).

Practical use of language types requires being able to easily calculate types from smaller types. Nullability and first sets have this property immediately, but it is not obvious for FLast sets. Indeed, the Follow sets of traditional LL(1) parsing lack this property! Follow is the set of characters that can follow a particular nonterminal, and so are a property of a grammar rather than being a property of languages.

FLast sets were originally introduced by Brüggemann-Klein and Wood [1992]. In that paper, they prove 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 FLast set. It was not explicitly remarked upon in that paper, but their definition, which both rules out sequential non-determinism and is defined in a grammar-independent way, offers a compositional alternative to the traditional follow set computation in LL parsing.

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 $\tau_\bot$ to be the type of the empty language, and so it is not nullable and has empty First and FLast sets. We define τ_ϵ to be the type of the language containing just the empty string, and it *is* nullable but has empty First and FLast 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 FLast set. 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 FLast sets.

The $\tau_1 \cdot \tau_2$ operation calculates a type for a language concatenation by first taking the conjunction of the Null fields. The First set is the First set of τ_1, unioned together with the First set of τ_2 when τ_1 is nullable. The FLast set is the FLast set of τ_2, merged together with the FLast set of τ_1 and the First set of τ_2 when τ_2 is nullable. Additionally, if either of the types are the type of the empty language, we do an optimization that returns the type of the empty language, since the language concatenation of any language with the empty language is the empty language.

One fact of particular note is that these definitions of First and FLast are *not correct* in general. That is, if $L \models \tau_1$ and $M \models \tau_2$, then in general $L \cdot M \not\models \tau_1 \cdot \tau_2$. It only holds when L and M are *separable*. That is, they must meet the preconditions of the decomposition lemma (Lemma 3.4). We define a separability predicate $\tau_1 \circledast \tau_2$ to indicate this, which holds when τ_1.FLast and τ_2.First are disjoint, and τ_1.Null is false. Similarly, we must also define *apartness* $\tau_1 \# \tau_2$ for non-overlapping languages, by checking that at most one of τ_1.Null and τ_2.Null hold, and that the First sets are disjoint (Lemma 3.3). This lets us prove the following properties of the type operators and language satisfaction:

Lemma 3.5. *(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* F *is a monotone function on languages such that for all* L, *if* $L \models \tau$ *implies* $F(L) \models \tau$, *then* $\mu F \models \tau$.

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 FLast, 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, $\textsc{First}(M) \subseteq \tau'.\textsc{First}$ and that $\textsc{FLast}(L) \subseteq \tau.\textsc{FLast}$, 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 $\textsc{FLast}(L \cdot M) \subseteq (\tau \cdot \tau').\textsc{FLast}$. To show this, assume that $c \in \textsc{FLast}(L \cdot M)$. So there exists $w \in L \cdot M \setminus \{\epsilon\}$ 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 $\tau'.\textsc{Null}$ must be true. 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 $\textsc{FLast}(L)$). In the first case, $c \in \textsc{FLast}(L)$, and since $\tau'.\textsc{Null}$ we know $(\tau \cdot \tau').\textsc{FLast} = \tau'.\textsc{FLast} \cup \tau'.\textsc{First} \cup \tau.\textsc{FLast}$. Hence $c \in (\tau \cdot \tau').\textsc{FLast}$. In the second case, $c \in \textsc{First}(M)$, and again we know $(\tau \cdot \tau').\textsc{FLast} = \tau'.\textsc{FLast} \cup \tau'.\textsc{First} \cup \tau.\textsc{FLast}$. So either way, $c \in (\tau \cdot \tau').\textsc{FLast}$.

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 $\textsc{FLast}(L)$, which contradicts the property that $\textsc{FLast}(L) \cap \textsc{First}(M) = \emptyset$. Hence $w'_L = w_L$ and $w'_M = w_M \cdot c \cdot w'$. Hence $c \in \textsc{FLast}(M)$, which immediately means that $c \in (\tau \cdot \tau').\textsc{FLast}$. □

4 Typing μ-regular Expressions

We now use the semantic types and type operators defined in the previous section 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."

This judgement has *two* contexts for variables, one for ordinary variables and one for the "guarded" variables, which are variables which must occur to the right of a nonempty string not containing that variable. So if $x : \tau$ is guarded, the grammar x is considered ill-typed, but the grammar $\boxed{c} \cdot x$ is permitted. The TVar 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 TEps rule says that the empty string has the empty string type τ_ϵ, and similarly the rules for the other constants TChar and TBot return types τ_c and $\tau_\bot$ for the singleton string and empty grammar constants.

The TCat 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

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

Figure 3. Contexts and Substitutions

$$\frac{}{\Gamma; \Delta \vdash \epsilon : \tau_\epsilon}\ \textsc{TEps} \qquad \frac{}{\Gamma; \Delta \vdash c : \tau_c}\ \textsc{TChar}$$

$$\frac{}{\Gamma; \Delta \vdash \bot : \tau_\bot}\ \textsc{TBot} \qquad \frac{x : \tau \in \Gamma}{\Gamma; \Delta \vdash x : \tau}\ \textsc{TVar}$$

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

$$\frac{\Gamma; \Delta \vdash g : \tau \qquad \Gamma, \Delta; \bullet \vdash g' : \tau' \qquad \tau \circledast \tau'}{\Gamma; \Delta \vdash g \cdot g' : \tau \cdot \tau'}\ \textsc{TCat}$$

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

Figure 4. Typing for Context-free Expressions

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 TVee 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 put to use in the TFix rule. It says that a context-free expression[3] $\mu x : \tau.\, g$ 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 similar to the typing rule for guarded recursion in Atkey and McBride [2013]; Krishnaswami [2013].)

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

Lemma 4.1. *(Weakening and Transfer) We have that:*

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

[3]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 expressions. Our implementation (Sections 5–6) does not require annotations; since grammar types form a lattice with a bottom element, we perform a fixed point computation to find the minimal type which works as an annotation.

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 lets us move variables from the guarded to 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. We use these properties to prove syntactic substitution.

Lemma 4.2. *(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* $\Gamma, \Delta; \bullet \vdash g : \tau$, *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 has the typing $\Gamma, \Delta; \bullet \vdash g : \tau$; 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. We first extend satisfaction to contexts, and then show well-typed terms are sound:

Definition 1. *(Context Satisfaction) We define context satisfaction* $\gamma \models \Gamma$ *as the recursive definition:*

$$\begin{array}{lllll} \bullet & \models & \bullet & \triangleq & \textit{always} \\ (\gamma, L/x) & \models & (\Gamma, x : \tau) & \triangleq & \gamma \models \Gamma \textit{ and } L \models \tau \end{array}$$

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. We can now prove soundness:

Theorem 4.3. *(Semantic Soundness) If* $\Gamma; \Delta \vdash g : \tau$ *and* $\gamma \models \Gamma$ *and* $\delta \models \Delta$ *then* $[\![g]\!]\ (\gamma, \delta) \models \tau$

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:

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

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 $F = \lambda X.\ [\![g]\!]\ (\gamma, \delta, X/x)$. Now, note that F is monotone by Proposition 3.2. Hence by Lemma 3.5, $\mu F \models \tau$, which is the conclusion we sought.

□

$$\frac{}{\epsilon \Rightarrow \epsilon}\ \textsc{DEps} \qquad \frac{}{c \Rightarrow c}\ \textsc{DChar}$$

$$\frac{g \Rightarrow w \qquad g' \Rightarrow w'}{g \cdot g' \Rightarrow w \cdot w'}\ \textsc{DCat} \qquad \frac{g_1 \Rightarrow w}{g_1 \vee g_2 \Rightarrow w}\ \textsc{D}\vee\textsc{L}$$

$$\frac{g_2 \Rightarrow w}{g_1 \vee g_2 \Rightarrow w}\ \textsc{D}\vee\textsc{R} \qquad \frac{[\mu x : \tau.\, g/x]g \Rightarrow w}{\mu x : \tau.\, g \Rightarrow w}\ \textsc{DFix}$$

Figure 5. Inference Rules for Language Membership

4.1 Typed Expressions are Unambiguous

We will now show there is at most one way to parse a typed context-free expression. 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; and the DFix rule asserts that a fixed point grammar $\mu x : \tau.\, g$ can generate a word if its unfolding can. There is no rule for the empty grammar $\bot$ (it denotes the empty language), or for variables (since we only consider closed expressions).

Generally, 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 an expression as follows:

Definition 2. *(Rank of a context-free expression)*

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

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

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

Theorem 4.5. *(Unambiguous Parse Derivations) If* $\bullet; \bullet \vdash g : \tau$ *then* $w \in [\![g]\!]\ \bullet$ *iff 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$\vee$L or D$\vee$R can apply. □

4.2 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 $\mathcal{P}\,(-)$, which takes as arguments a typing derivation $\Gamma; \Delta \vdash g : \tau$, as well as environments $\hat{\gamma}$ and $\hat{\delta}$ which give parsers for each of the free variables in g. This function defines a parser by recursion over the structure of the typing derivation. We define a parser (really, a recognizer) as a partial function on strings of characters ($\Sigma^* \rightharpoonup \Sigma^*$). 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).

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

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

So if a parser takes w as an input, and returns 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 $\hat{\gamma} \triangleright_n \gamma$ is defined similarly.

Theorem 4.6. *(Soundness of Parsing) For all* n, *assume* $\Gamma; \Delta \vdash g : \tau$, *and* $\gamma \models \Gamma$, *and* $\delta \models \Delta$, *and* $\hat{\gamma} \triangleright_n \gamma$ *and* $\hat{\delta} \triangleright_{n-1} \delta$. *Then we have that* $\mathcal{P}\,(\Gamma; \Delta \vdash g : \tau)\ \hat{\gamma}\ \hat{\delta} \triangleright_n [\![g]\!]\ (\gamma, \delta)$

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. □

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 $w'' \in \Sigma^*$ such that $c \notin \textsc{FLast}(L)$ 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. 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 $\hat{\gamma} \blacktriangleright_n \gamma$ is defined similarly.

Theorem 4.7. *(Completeness of Parsing) For all n, assume $\Gamma; \Delta \vdash g : \tau$ and $\gamma \models \Gamma$ and $\delta \models \Delta$ and $\hat{\gamma} \blacktriangleright_n \gamma$ and $\hat{\delta} \blacktriangleright_{n-1} \delta$. Then $\mathcal{P}(\Gamma; \Delta \vdash g : \tau)\ \hat{\gamma}\ \hat{\delta} \blacktriangleright_n [\![g]\!]\ (\gamma, \delta)$*

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 FLast set of $[\![g_1 \cdot g_2]\!]\ \gamma\ \delta$ includes the First set of $[\![g_2]\!]\ \gamma\ \delta$ when g_2 is nullable. □

Soundness tells us that if the parser succeeds, then it consumed a word of the language. Completeness tells us 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.

5 Unstaged Implementation

The theory described in the previous sections is for *recognizers*, which test 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 §3, and typechecking is easiest to implement on a first-order representation. In addition, we want to ensure that the parsers we generate are also well-typed in the sense that we know what type of OCaml values they produce.

In other words, our goal is to provide the usual surface API for parser combinators (see §2), but to give the *implementation* a first-order representation. To achieve this, we begin with a first-order representation, to which we offer an interface based on higher-order abstract syntax to turn user-written higher-order code into an analyzable first-order representation (see §5.3).

5.1 Representing Grammars

To write a typechecker as a standard tree-traversal, we need to use a first-order syntax tree to represent grammars, variables and fixed points. However, matters are complicated by the need to attach binding and OCaml type information to our 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 : ('ctx, unit) 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 a parameter `'ctx`, which denotes 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 comes 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 inconvenient, since it is an intrinsic de Bruijn representation. This is why our surface API (§2.1) uses HOAS to hide this from client programs.

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

Figure 6. Parsing Algorithm

5.2 Typechecking and Parsing

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 §4. 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, and then we can implement a typechecker as a recursive function.

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

This is almost the same as the types defined in §3. The only difference is the extra field `guarded`, which tracks whether a variable is in the Γ or Δ context. This saves us from having to manage two contexts as GADT indices.

Typechecking is done with the `typeof` function, which just walks over the structure of the grammar, doing type checking at each step. It takes a representation of the type context, and a grammar, and returns a grammatical type, raising an exception if typechecking fails.

```
val typeof : type ctx a d.
  ctx tp_env -> (ctx, a) Grammar.t -> Tp.t
```

The formal system in §4 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).

The type we use for parsers is very simple:

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

This takes a stream of characters, and either returns a value or fails with an exception. This type *does not support backtracking*; since we take an imperative stream, we can only step past each character once. We implement one combinator for each of the data constructors, all of which have obvious implementations. The most 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 error "No progress possible"
```

This function peeks at the next token, and uses that information together with the static type information (i.e., First and Null sets) to decide whether to invoke `p1` or `p2`. This way backtracking can be avoided. Then we can write a function

```
val parse : type ctx a d.
  (ctx, a) Grammar.t -> ctx parse_env -> a parser
```

to traverse the tree, invoking combinators to build a parser.

5.3 From Higher-Order to First-Order

While first-order representations are convenient for analyses such as type checking, higher-order interfaces are more convenient for programming. Following Atkey et al. [2009], we combine the advantages of both approaches by translating the higher-order combinators into the first-order representation. We implement the interface term `'a t` of §2.1 as a function from a context to a value of the first-order data type of §5. 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 interesting case is `fix`, which must adjust the context 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 }
```

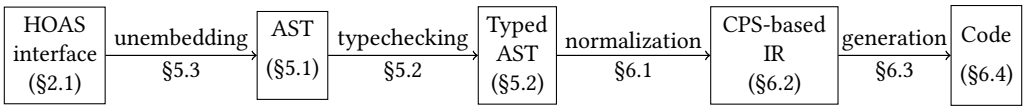

Figure 7. From combinators to specialized code

6 Adding Staging for Performance

The typed combinators of §2.1 enjoy a number of advantages over both parser combinators and parser generators: unlike most combinator parsers they run in linear time with no ambiguities; unlike yacc-style parser generators they support binding and substitution. However, they are much slower than parsers generated by ocamlyacc. In the benchmarks of §7 they are between 4.5 and 125 times slower than ocamlyacc.

We address this shortcoming using *multi-stage programming*, a technique for annotating programs to cause evaluation to take place over several stages. A staged program receives its input in several stages, and each stage generates a new program specialized to the available input, avoiding the performance penalty of abstraction over arbitrary inputs. Improving the performance of parsing using multi-stage programming has a long history, which we discuss further in §8. Our parser combinators build on that history: we make use of several existing techniques from the literature, but also add a new idea to the panoply, showing how the types computed for our grammars improve the generated code.

The modified version of our combinators described in this section evaluates 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 more efficient than equivalent parsers written using the unstaged combinators of §2.1, 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 MetaOCaml's type system prevents safety violations such as evaluation of open code.

The changes involved in staging the combinators are mostly internal. Only one change to the interface is needed: the `map` combinator should now work with quoted `code` expressions:

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

The type of `Map` in the GADT needs similar adjustment.

6.1 Binding-time improvements

In principle, staging a program is straightforward. First, the programmer identifies each *input* of the program as *static* (available to the first stage) or *dynamic* (available only to a later stage). Next, *expressions* are annotated: those that make use of dynamic inputs are bracketed, while other subexpressions can be escaped or, if at top-level, left unbracketed [Taha 1999, p87]. For parsers there are three inputs: the grammar and the parsing environment are static, and the input stream is dynamic. Any expressions depending on the input stream must be bracketed, delaying their evaluation to the second stage. In practice, further changes—so-called *binding-time improvements*, that change the program to reduce the number of expressions that depend on dynamic inputs—are often needed for optimal results.

CPS-based improvements Here is a typical example of a binding-time improvement. In the following code, from §5.2, the unstaged `alt` combinator interleaves access to the dynamic input `s` and the static type representation `tp1`:

```
let alt tp1 p1 tp2 p2 s = match Stream.peek s with
    ...
  | Some c -> if CharSet.mem c tp1.first then p1 s
```

Naively following the staging process classifies the whole expression to the right of the arrow as dynamic, since it depends on `c`, which depends on `s`. This is undesirable, since some parts of the expression (e.g. `tp1`) are statically known.

One well-known technique for disentangling the static and dynamic dependencies in an expression is to perform CPS conversion [Bondorf 1992; Nielsen and Sørensen 1995]. We follow this approach, introducing a new version of `peek` that passes its result to a continuation function; additionally, since the next step is always to determine whether the character belongs to some set, we extend `peek` to accept the set as an argument:

```
let alt tp1 p1 tp2 p2 = peek tp1.first (fun c -> ...
```

(Note that `alt`'s argument `s` has gone too; `peek` returns a dynamic function that accepts `s` as argument.) Now the call to `peek` does not use any dynamic inputs: both the character set `tp1.first` and the continuation argument are statically known.

The Trick It is possible to improve `peek` still further, using an ubiquitous technique from the partial evaluation community known simply as The Trick [Danvy et al. 1996]. The idea is as follows: we do not statically know the value of the dynamic variable `c`, but we know its type and so know that it will eventually be bound to one of a fixed set of values. We can use this fact by replacing any expression `e` that depends on `c` with a dynamic test that branches on `c`'s value:

```
.< match c with 'a' -> e | 'b' -> e | ... >.
```

After this change, we know the value of `c` in each branch `e` and can further specialize the program.

In practice, branching on every possible value of `c` generates unwieldy code; it is sufficient to branch on whether `c` belongs to the set passed to `peek`. We consequently change `peek` one last time to return an indication of whether the next character in the stream belongs to the set:

```
let alt tp1 p1 tp2 p2 =
  peek tp1.first (function
    | `Eof -> if tp1.null then p1 else p2
    | `Yes -> p1
    ...
```

Now the expression to the right of the arrow has no dynamic dependencies and `tp1` is no longer needed in the second stage.

6.2 An intermediate representation

While CPS translation is a binding-time improvement, the higher-order code that results is difficult to further analyse directly. Consequently, we once again reduce our higher-order code to a datatype representation [Ager et al. 2003], just as we did with the grammar interface (§5.3). The `peek` function constructs a value of an IR which can be traversed to generate efficient code.

6.3 Generating code

The final step after the above transformations is to generate code from the IR. Code generation is largely a straightforward traversal of the IR, generating code for each node and composing the results. However, two points deserve note.

Branch pruning One source of inefficiency in the unstaged combinators is that they may examine the first element in the input stream more than once. For example, when the parser `chr 'a' <|> chr 'b'` reads a character `'a'` from the input, there are two calls to `Stream.peek`—first in `alt` to determine which branch to take, and then in `chr` to determine whether the character matches. In the staged program this redundancy can be avoided by taking into account the *context* in which `chr` is called. Our type system gives the left-hand branch `chr 'a'` a type whose first set is simply `{'a'}`: there is therefore no need to call `Stream.peek` a second time in the code generated by this call to `chr`. Once again, therefore, the information from the type checking phase leads to improved code. (Branch pruning itself is an essential aspect of several existing multi-stage programs [Inoue 2014; Rompf et al. 2013]; however using a type system to provide the information needed to prune branches is novel.)

Recursion One of the combinators in our interface (§2.1) is a fixed-point operator, `fix`. In the unstaged implementation, a call to `fix` constructs a single recursive function, and nested calls to `fix` result in nested recursive functions. Once again, naively staging the implementation of `fix` yields sub-optimal results—i.e. it leads to similar nested structure in the generated code. It is better (both for legibility and efficient compilation) to flatten the code into a top-level mutually-recursive group. Our staged combinators generate one mutually-recursive group for each complete grammar using a recent extension to MetaOCaml for generating mutually-recursive functions [Yallop and Kiselyov 2019].

6.4 Generated code

Our staged combinators perform better than traditional combinators, as we demonstrate in §7. The following generated code (for s-expressions) shows why:

```
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 -> f1 index (String.length s) n)
```

The output resembles low-level code one might write by hand, without abstractions, 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. Each function examines the input character-by-character, branching immediately and proceeding deterministically. Although the example is small, this code is representative of the larger examples in §7.

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, the GADT is typechecked using the type system for context-free expressions. Third, the typechecked term is converted to a first-order CPS-based intermediate representation, making use of the types built in the second phase. Finally, the IR is converted to efficient code using MetaOCaml's quasiquotation facilities, again making use of the output of the typechecking phase. We emphasize that although the diagram in Figure 7 resembles the structure of a compiler, our implementation is a pure library, written entirely in user code.

Generalizing to non-character tokens Our description of the library has focused on characters, but our implementation is parameterized over the token type, and can build

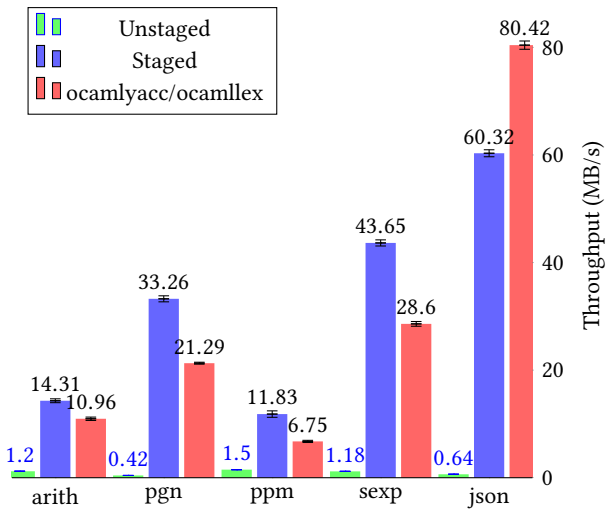

Figure 8. Parser throughput: unstaged & staged combinators and `ocamlyacc`/`ocamllex`. Error bars show 95% CIs.

parsers for streams of arbitrary tokens. §7 describes the use of this parameterization to construct 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, including input channels and streams that produce tokens one by one rather than materializing a complete list of tokens.

7 Performance Measurements

Parsing non-trivial languages typically involves a distinct scanning phase that turns a sequence of characters into a sequence of tokens, to be consumed by the parser proper. These two phases are often implemented using different tools. For example, OCaml programs often use `ocamllex` to generate scanners and `ocamlyacc` to generate parsers.

Using our library both the scanner and the parser can be written using the same combinators. First, the combinators are used to describe a scanner that builds a token stream from a character stream. The same combinators are then used to implement a parser that consumes the output of the scanner. As the measurements in this section show, the performance of parsers written in this way compares favourably with the code generated by `ocamllex` and `ocamlyacc`.

Figure 8 compares the throughput of three families of parsers. The *unstaged* bars show the throughput of the combinators described in §5. Each parser is written using two applications of the combinators, to build a scanner and a parser. The *staged* bars show the throughput of parsers written using the staged combinators (§6). These parsers resemble the unstaged implementations, but they are annotated with staging annotations (brackets and escapes), and so generate code specialized to each grammar. Finally, the `ocamlyacc`/`ocamllex` bars show the throughput of parsers written with the standard OCaml tools, following the same grammatical structure as the combinator-based implementations.

Our measurements avoid the overhead of allocating AST nodes in the benchmarks; instead, each parser directly computes some function of the input—e.g. a count of certain tokens, or a check that some non-syntactic property holds. The languages are as follows:

1. (arith) A mini programming language with arithmetic, comparison, let-binding and branching. The semantic actions evaluate the parsed expression.
2. (pgn) Chess game descriptions in PGN format[4]. The semantic actions extract each game's result. The input is a corpus of 6759 Grand Master games.
3. (ppm) Image files in PPM format[5]. The semantic actions validate the non-syntactic constraints of the format, such as colour range and pixel count.
4. (sexp) S-expressions with alphanumeric atoms The semantic actions of the parser count the number of atoms in each s-expression.
5. (json) A parser for JavaScript Object Notation (JSON). The semantic actions count the number of objects represented in the input. We use the simple JSON grammar given by Jonnalagedda et al. [2014].

The benchmarks were compiled with BER MetaOCaml N107 and run on an AMD FX 8320 machine with 16GB of memory running Debian Linux, using the `Core_bench` micro-benchmarking library [Hardin and James 2013]. The error bars represent 95% confidence intervals.

Even though our type system ensures that parsing has linear-time performance, the abstraction overhead involved in parser combinators makes the performance of the unstaged version uncompetitive—at best (for PPM) around 22% of the throughput of the corresponding `ocamlyacc`/`ocamllex` implementation, and at worst (for JSON) less than 1% of the throughput. However, staging both eliminates this overhead and avoids the interpretive overhead involved in the table-based `ocamlyacc` implementation. For every language except JSON, the throughput of the staged combinator implementation exceeds the throughput of the corresponding `ocamlyacc` version, often significantly (e.g. by 75% for PPM).

Linear-time performance Figure 9 provides empirical support for the claim that our parser combinators, like `yacc`-based systems, offer linear-time performance. Each graph plots running time against input size. (For reasons of space, we omit the similar results for the unstaged version.)

Further performance improvements There are several opportunities to further improve combinator performance.

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

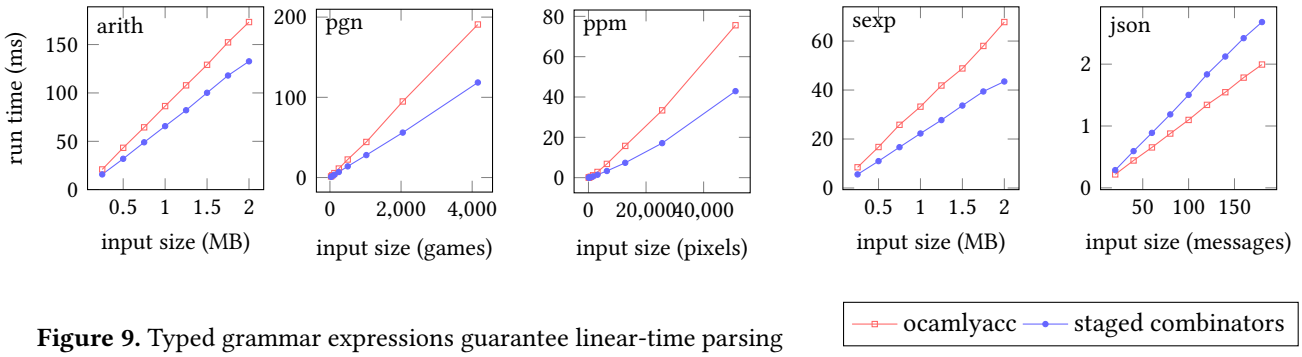

Figure 9. Typed grammar expressions guarantee linear-time parsing

The handling of tokens carrying strings is inefficient (and accounts for the relatively poor performance in the JSON benchmark), since the minimal combinator API necessitates reading strings as `char` lists before conversion to a flat representation; we plan to remedy this by extending the API. More ambitiously, we plan to combine the code generated for the scanning and parsing phases into a single unit to enable further optimizations, e.g. avoiding token materialization. Preliminary experiments suggest that throughput improvements of 2× upwards may be available in this way.

8 Related Work

Adding fixed points to regular expressions was described in Salomaa [1973], in a construction called the "regular-like languages". Later, Leiß [1991] extended Kleene algebra [Kozen 1990] with least fixed points, giving the untyped μ-regular expressions we use, and also noted that context-free languages offer a model. Ésik and Leiß [2005] showed that the equational theory sufficed to translate grammars to Greibach normal form. More recently, Grathwohl et al. [2014] have completely axiomatized the *in*equational theory. This work is for untyped μ-regular expressions; restricting CFGs with a type system seems to be a novelty of our approach.

Our notion of type—especially the FLast set—drew critical inspiration from the work of Brüggemann-Klein and Wood [1992]. They prove a Kleene-style theorem for the *deterministic regular languages*, which can be compiled to state machines without exponentially larger state sets. Johnstone and Scott [1998] independently invented the FLast property (which they named "follow-determinism") while studying generalised recursive descent parsing. They prove that for nonempty, left-factored grammars, using follow-determinism is equivalent to the traditional Follow set computation. Thus, our type system can be understood as the observation that LL-class parsing arises from adding guarded recursion to the deterministic regular languages.

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 syntactic constraint with a type-based guardedness restriction on variables.

Staging and parsers share a long history. In an early work on regular expression matching, Thompson [1968] takes a staged approach, dynamically generating machine code to recognize user-supplied regular expressions, and 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, they focus on ambiguous and input-dependent grammars that may require backtracking while our type system ensures that our parsers are deterministic and guarantees linear-time performance.

Swierstra and Duponcheel [1996] also gave parser combinators 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, 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 much stronger guarantees about runtime.

Acknowledgements We thank Matthew Pickering, Isaac Elliott, Nada Amin and the anonymous reviewers for comments on earlier drafts.

References

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. *From Interpreter to Compiler and Virtual Machine: a Functional Derivation.* Technical Report BRICS RS-03-14. Aarhus, Denmark.

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. https://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. https://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. https://doi.org/10.1007/BFb0048933

Anders Bondorf. 1992. Improving Binding Times Without Explicit CPS-conversion. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/141471.141483

Anne Brüggemann-Klein and Derick Wood. 1992. Deterministic Regular Languages. In *9th Annual Symposium on Theoretical Aspects of Computer Science (STACS 92)*. 173–184. https://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. https://doi.org/10.1145/321239.321249

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-Expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (1996), 730–751. https://doi.org/10.1145/236114.236119

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. https://doi.org/10.1145/237721.237788

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.

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.

Graham Hutton. 1992. Higher-Order Functions for Parsing. *Journal of Functional Programming* 2, 3 (001 007 1992), 323–343. https://doi.org/10.1017/S0956796800000411

Jun Inoue. 2014. Supercompiling with Staging. In *Fourth International Valentin Turchin Workshop on Metacomputation*.

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

Adrian Johnstone and Elizabeth Scott. 1998. Generalised Recursive Descent parsing and Fellow-Determinism. In *CC (Lecture Notes in Computer Science)*, Vol. 1383. Springer, 16–30.

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. https://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. 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. https://doi.org/10.1145/2500365.2500588

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. https://doi.org/10.1145/321466.321477

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13.

Kristian Nielsen and Morten Heine Sørensen. 1995. Call-By-Name CPS-Translation As a Binding-Time Improvement. In *Proceedings of the Second International Symposium on Static Analysis (SAS '95)*. Springer-Verlag, London, UK, UK, 296–313. http://dl.acm.org/citation.cfm?id=647163.717677

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

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers Based on Staging. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 497–510. https://doi.org/10.1145/2429069.2429128

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

Michael Sperber and Peter Thiemann. 2000. Generation of LR Parsers by Partial Evaluation. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 224–264. https://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. https://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.

Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. https://doi.org/10.1145/363347.363387

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. 2019. Generating Mutually Recursive Definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2019)*. ACM, New York, NY, USA, 75–81. https://doi.org/10.1145/3294032.3294078

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.