Certified Parsing of Dependent Regular Grammars

John Sarracino
Cornell University
jsarracino@cornell.edu

Gang Tan
The Pennsylvania State University
gtan@psu.edu

Greg Morrisett
Cornell Tech
greg.morrisett@cornell.edu

***Abstract*—Parsers are ubiquitous, but formal reasoning about the behavior of a parser is challenging. One key challenge is parsing dependent formats, which are difficult for traditional techniques to handle because parse values can influence future parsing behavior. We present dependent regular grammars, which extend regular languages with data-dependency by generalizing concatenation to monadic bind. Even this small tweak adds significant expressive power; for example, conditional parsing and dependent repetition are both implementable using monadic bind.**

However, it is not obvious how to actually parse dependent regular grammars. We implement a Brzozowski derivative based matching algorithm, and we show how many popular parser combinator functions can be implemented in our library while retaining the same simplicity as traditional parser combinators.

We implement and formalize these grammars in Coq, as well as a derivative-based matching algorithm. We prove soundness and completeness of the derivative operator in the standard way. We also implement a variety of popular parser combinator functions and give formal specifications to them. Finally, we implement as a case study a verified netstring parser, and prove functional correctness of the parser.

***Index Terms*—Monadic parser combinators, certified parsing, Coq, data-dependent parsing, parsing with derivatives.**

I. Introduction

Parsers convert input data (such as networking packets) into a structured format (such as packet headers). In modern systems, parsers are the intake component in larger application pipelines. Downstream applications make use of the structured output of parsers, and generally assume that the parser is trustworthy and that the structured output is correct. As a result parsers are the first line of defense for modern systems. It is critical that they have strong termination guarantees and that they are robust to adversarial input.

Despite the ubiquity of parsers, building a correct parser is a difficult and error-prone task. Formal reasoning about realistic parsers is challenging for both humans and algorithms. One common challenge is reasoning about data-dependency, in which one part of the parse depends on the *value* of a previous parse result.

In this work, we extend regular languages with a data-dependent *bind* operator, inspired by monadic parser combinators and written as $\gg\!=$. This operator generalizes the traditional language concatenation operator and allows previously parsed values to influence future parse results. Consider as a motivating example the challenge of *filtering*, in which a parser conditionally accepts or rejects if the input does not match a validation filter function. As we will see in section III, a filter combinator can be directly written using monadic bind.

The apparently small extension of monadic bind adds significant expressive power. With monadic bind, parsers are now able to directly use a parse value to determine future parsing behavior. As a consequence dependent regular grammars can serve as a formal model for many kinds of highly optimized low-level parsers, which do not fit into traditional formalisms due to their data-dependent nature.

In this work, we mechanize dependent grammars in Coq. To actually compute parse values, we implement a matching algorithm based on Brzozowski derivatives [1]. It surprisingly retains the simplicity of traditional Brzozowski derivatives, despite the extra expressiveness of bind. We formalize the correctness of the derivative and show that it is both sound and complete with respect to the parsing semantics of dependent grammars. All proofs in this work are mechanized in Coq unless otherwise stated.

Contributions: This paper makes the following contributions. We extend regular languages with data-dependent bind in section III, and we adapt Brzozowski's derivative to dependent regular grammars in section III. We show how to implement various popular parsing combinator functions using dependent grammars in Coq in section IV. Finally we implement a certified parser for netstrings in section V.

II. Related work

Our work extends regular languages with a monadic bind operator, develops a Brzozowski derivative operator for the extended languages, and mechanizes the effort in a proof assistant (Coq).

A. Monadic parser combinators

Monadic parser combinators are a foundational idea drawn from the functional programming literature [2]–[4]. The key idea is that parsers are fundamentally stateful functions from strings to partial parse results, and monads can be used to implement the stateful nature of parsing. This idea is very influential; modern parser combinator libraries are around for most popular languages including Haskell [5], OCaml [6], C [7], and Rust [8]. Most recently research has focused on applying classical type-system ideas to parser combinator libraries, to aid developers in constructing correct parsers [9]. This work extends regular languages with a primitive for monadic bind inspired by monadic parser combinators.

B. Dependent parsing

Data-dependent formats have posed a long-standing challenge to declarative parser generators. Recent work has focused on extending generalized parsing algorithms [10], the Earley algorithm [11], and various other parser-generator approaches [12] with data dependency, by tweaking the declarative algorithm's underlying data-structure with support for dependent parsing. Our work also adds data-dependency to a declarative language (namely regular languages) by adding monadic bind to regular languages. As a result our work is strictly less expressive than data-dependent parser generators, as regular languages are strictly smaller than context-free languages. However, one drawback to data-dependent parser generators is that their performance is extremely unintuitive (indeed, ill-formed grammars can cause them to have nonterminating behavior during parsing). Our work does not have this drawback; since we implement the Brzozowski derivative as a total function in a proof assistant, it is guaranteed to terminate. Moreover, to the best of our knowledge, data-dependent parser generators have *not* been formalized in a proof assistant. Our work could serve as such a formalism if it were generalized to context-free languages by e.g. adding a fixpoint/recursion grammar primitive.

C. Certified parsing

A long line of work has focused on reasoning about the output of a parser in a proof assistant.

1) Parsing with derivatives: Derivative-based parsing is a modern application of Brzozowski derivatives [1], rediscovered in the modern era by [13], [14] and proved efficient by [15]. While neither of these efforts were done in a proof assistant, in fact, they were very influential in the theorem-proving community, because reasoning about the derivative operator is relatively easy, e.g. [16]–[19].

2) Total parser combinators: Most closely related to our work is that on total parser combinators within a proof assistant. There are several unpublished works: a workshop presentation by McBride and McKinna [20], as well as a blog post by a user named "Muad'Dib." [21]. There are also two recently published results which we discuss in turn.

Certified parser combinators: Danielsson [22] develops a derivative-based approach for total monadic parsers in Agda. The key idea is to allow left-recursion for *productive* corecursive parsers. To accomodate this, Danielsson develops a mixed inductive-coinductive grammar for parser primitives, as well as tracking the epsilon-interpretations of parsers in their type indices. Our approach is less expressive (e.g. does not support left-recursion) but also significantly simpler. One way to view our work is that we demonstrate that the complexity of Danielsson's derivative operator is entirely due to supporting left-recursion, and not due to data-dependency (i.e. monadic bind).

Productive parser combinators: In Agda (and ported to Coq), Allais develops a type system and combinators for enforcing that shallowly embedded parser combinator programs in Gallina always make forward progress [23]. In doing so, this work avoids the complexity of coinduction, while still allowing many forms of recursion. However, because this effort lives entirely within shallowly-embedded functions, it does not develop a derivative-based approach. By contrast we develop a derivative-based approach using a deep embedding of parsers. It would be interesting to compare the relative verification effort of using Allais' shallow combinators vs our own parsing relation.

3) RockSalt: Our work is directly inspired by RockSalt [24], which uses derivatives to build a verified lexer for x86 instructions. In particular, we use RockSalt's formalism as a starting point for regular languages and extend it by adding monadic bind.

$$p ::= \emptyset \mid \texttt{.} \mid p|p \mid \texttt{pure}\ v \mid p \gg\!= f \mid p^*$$

Grammar syntax.

$$\textsc{PAny}\ \frac{}{(c :: \epsilon, \texttt{.}) \rightarrowtail c} \qquad \textsc{PPure}\ \frac{(s, p_l) \rightarrowtail v}{(s, p_l|p_r) \rightarrowtail v} \qquad \textsc{PAltL}\ \frac{(s, p_r) \rightarrowtail v}{(s, p_l|p_r) \rightarrowtail v}$$

$$\textsc{PAltR}\ \frac{}{(\epsilon, \texttt{pure}\ v) \rightarrowtail v} \qquad \textsc{PStar0}\ \frac{}{(\epsilon, p^*) \rightarrowtail []}$$

$$\textsc{PStarIter}\ \frac{s \neq \epsilon \qquad (s, p) \rightarrowtail x \qquad (s', p^*) \rightarrowtail xs}{(s +\!\!+ s', p^*) \rightarrowtail x :: xs}$$

$$\textsc{PBind}\ \frac{(s_l, p) \rightarrowtail x \qquad (s_r, f\ x) \rightarrowtail y}{(s_l +\!\!+ s_r, p \gg\!= f) \rightarrowtail y}$$

Grammar semantics.

Fig. 1. Syntax and semantics for dependent grammars.

III. Dependent grammars

In this section, we present the technical core of this work: dependent regular grammars and Brzozowski derivatives for dependent regular grammars.

A. Syntax and semantics

We give the syntax and semantics of dependent grammars in Figure 1.

We formalize the parsing semantics as a relation between strings (lists of characters), grammars, and values. We write ϵ as the empty string, or equivalently, the empty list $[]$; and we write $l +\!\!+ r$ to mean the list concatenation of two lists l and r. For a string s, a parser p, and a value v, we write $(s, p) \rightarrowtail v$ to relate s and p to v (i.e. that p parses s to v).

Parsing relation primer: Our parsing relation inference rules should be read bottom-up: the bottom relation follows from the preconditions above the line. For example, one of the rules for alternation is $\frac{(s, p_l) \rightarrowtail v}{(s, p_l|p_r) \rightarrowtail v}$. In plain English,

if a grammar p_l parses a string s to a value v, then it can be combined with any other parser p_r using | to also parse s to v. Rules that do not have entries above the line are axioms and have no preconditions. For example the rule for $\texttt{pure}\ v$ is $\frac{}{(\epsilon, \texttt{pure}\ v) \rightarrowtail v}$. In plain English, the parser $\texttt{pure}\ v$ parses the empty string ϵ to the value v for all values v.

Overview of grammars: We discuss each of the grammar syntax forms and their corresponding parsing rules in turn.

Failure: We represent a parser in failure state as $\emptyset$. Because this parser does not parse any values, it does not have any rules in the parsing relation.

Characters: We represent a grammar for a single character as $\texttt{.}$. The parsing relation rule for $\texttt{.}$, PAny, relates a string with just one character c to the value c.

Alternation: We represent grammar alternation (traditional union) between two parsers p_l and p_r as $p_l | p_r$. There are two parsing relation rules for alternation PAltL and PAltR, one for each side of the alternation. The left-hand side PAltL says that if p_l parses a string s to a value v, then $p_l \mid p_r$ parses s to v as well. The right-hand side PAltR is analogously defined for p_r.

Iteration: We represent iteration using Kleene star and we write p^* to represent the Kleene iteration of a grammar p. There are two parsing rules for iteration, for zero and non-zero amounts of repetition, PStar0 and PStarIter. The first is standard and relates the empty string ϵ to an empty list for all parsers p. The second rule, PStarIter, is a bit strange at face value, as it requires that the interior parser p to be *productive* and consume a nonempty amount of input. This restriction is inspired by the classic functional pearl on regex matching by Harper [25], and Harper also showed that the restriction does not have an impact on expressiveness for regular languages as unproductive regexes can be normalized to an equivalent productive regex. It takes some care to adapt Harper's normalization algorithm to dependent regular grammars, which we do not cover in this work[1]. In practice useful grammars are productive and so it does not hinder expressiveness. Moreover it turns out that this productivity restriction is important for our formalisms later.

Dependent bind: This rule replaces traditional concatenation with monadic bind. Given a parser p returning values of type A and a function f from A to parsers returning values of type B, we write bind as $p \gg\!= f$, which is a parser that produces values of type B.[2] Because f has access to the result of p, the grammar it returns can be built from the parse result.

The parsing semantics of $p \gg\!= f$ are the same as traditional monadic parser combinators. If p parses a prefix s_l to a value x, and the parser produced by $f\ x$ parses a suffix s_r to a value y, then $p \gg\!= f$ parses the concatenation $s_l +\!\!+ s_r$ to the value y as well.

$$\begin{aligned} p\ @\ f &= p \gg\!= (\lambda x \Rightarrow \texttt{pure}\ (f\ x)) \\ p_l\ \$\ p_r &= p_l \gg\!= (\lambda x \Rightarrow p_r \gg\!= (\lambda y \Rightarrow \texttt{pure}(x, y))) \end{aligned}$$

Fig. 2. Two helper functions for epsilon-interpretation and the derivative, which respectively define function application onto grammars and traditional concatenation.

Examples: Suppose `dig` is a parser for a single numeric digit, and `lower` parses a lower-case English letter. The grammar $\texttt{.}$ parses the string "1" to the character '1' by the PAny rule. The grammar $\texttt{dig} \mid \texttt{lower}$ parses "a" to the character 'a' by the PAltR rule. The grammar $\texttt{dig}^*$ parses "123" to the list ['1'; '2'; '3'] by three applications of PStarIter and one of PStar0.

Dependent filtering: In the introduction, we briefly discussed a motivating example for dependent parsing: conditional filtering. Now that we have a syntax in hand, we can define filter in terms of primitive grammars as a monadic bind followed by either a return or parse failure[3]:

$$\texttt{filter}\ p\ f = p \gg\!= \lambda x \Rightarrow \texttt{if}\ fx\ \texttt{then pure}\ x\ \texttt{else}\ \emptyset$$

Filter is useful for rich semantic validation. For example, the grammar $\texttt{filter}\ dig\ (\lambda x \Rightarrow x >?\ 0)$ only parses non-zero digits. In particular, it parses "1" to 1 by the PBind rule, but by contrast on the input "0", the parsing relation becomes stuck because it requires the derivation of $(\epsilon, \emptyset) \rightarrowtail v$ for some result v. This is not possible because there are no rules for $\emptyset$ and so the overall parser does not parse "0".

Now that we have a syntax and semantics for dependent grammars, we turn to actually evaluating the parsing relation. We take an approach inspired by parsing-with-derivatives.

B. Dependent grammar derivatives

There are two key ideas with derivative-based parsing. First, we inductively define a parser's evaluation of the empty string (the parsing analogue of nullability). We term this epsilon-interpretation and write $\|p\|_\epsilon$ to mean the set of values v (calculated as a list) such that $(\epsilon, p) \rightarrowtail v$. Second, we define a derivative operator over grammars, which takes a grammar p and a character c, and returns a grammar that parses s to v for all strings s and values v such that p parses $c :: s$ to v[4]. We write this operator as $\mathrm{D}_c\ p$, which is intentionally designed to *not* change the parse results of a grammar (except by discarding results in case of a parse failure).

We define $\|p\|_\epsilon$ as a recursive function over the structure of p, given in Figure 3. Most of the rules are straightforward and relatively standard. $\emptyset$ and $\texttt{.}$ do not parse the empty string, so their interpretation is []. Alternation is the list concatenation of its arguments, and $\texttt{pure}\ v$ produces just one value v.

[1]Harper's normalization algorithm is defined inductively over regular expressions. In many aspects dependent bind behaves similar to traditional concatenation, but since the inner function requires an argument, algorithms need to generate intermediate parse results in a sound way in order to unlock the right-hand-side of a dependent bind. Later on, we will show how to do this for a derivative operator by definition of *epsilon-interpretation*.

[2]Notice that if we write $parser\ A$ to mean a parser that returns values of type A, then f has the classic monadic type $f : A \rightarrow parser\ B$, and bind has the type of monadic bind: $\gg\!=$: $parser\ A \rightarrow (A \rightarrow parser\ B) \rightarrow parser\ B$.

[3]For a Gallina definition of filter, see Figure 10

[4]Intuitively, the derivative operator is "running" p on just the first character c of $c :: s$.

$$
\begin{array}{rl}
\|\emptyset\|_\epsilon & = [] \\
\|\,.\,\|_\epsilon & = [] \\
\|p_l|p_r\|_\epsilon & = \|p_l\|_\epsilon +\!\!+ \|p_r\|_\epsilon \\
\|\texttt{pure}\ v\|_\epsilon & = [v] \\
\|p^*\|_\epsilon & = [[]] \\
\|p \gg\!= f\|_\epsilon & = \texttt{concat\_map}\ (\lambda x \Rightarrow \|f\ x\|_\epsilon)\ \|p\|_\epsilon
\end{array}
$$

Fig. 3. Grammar interpretations of empty strings. We use lists (potentially with duplicates) of parse results to represent values in the parsing relation.

$$
\begin{array}{rl}
\mathrm{D}_c\ \emptyset & = \emptyset \\
\mathrm{D}_c\ . & = \texttt{pure}\ c \\
\mathrm{D}_c\ p_l|p_r & = \mathrm{D}_c\ p_l|\mathrm{D}_c\ p_r \\
\mathrm{D}_c\ \texttt{pure}\ v & = \emptyset \\
\mathrm{D}_c\ p^* & = (\mathrm{D}_c\ p)\ \$\ p^* \\
\mathrm{D}_c\ p \gg\!= f & = (\mathrm{D}_c\ p \gg\!= f)|\bigvee_{x \in \|p\|_\epsilon} \mathrm{D}_c\ (f\ x)
\end{array}
$$

Fig. 4. Brzozowski derivative of dependent grammars.

Finite epsilon-interpretation of iteration: The epsilon-interpretation of Kleene iteration is interesting because traditional Kleene iteration has a potentially infinite epsilon-interpretation. For example, the grammar $(\texttt{pure}\ 1)^*$ parses ϵ to 1, $\epsilon\epsilon$ to 1, $\epsilon\epsilon\epsilon$ to 1, etc. This poses a challenge both for our Coq implementation (which requires the definition of epsilon-interpretation to be total), as well as for implementing a finite derivative function.

This is where our restriction on the productivity of Kleene iteration comes into play. Because we force iteration to be productive, in fact, the only possible epsilon-interpretation of p^* is []. This is best understood by the following identity (which we have mechanized in Coq):

Lemma 3.1:

$$
\begin{array}{l}
\forall s\ p\ v, \\
\qquad (\forall s\ v, (s,p) \rightarrowtail v \implies s \neq \epsilon) \implies \\
\qquad ((s,p^*) \rightarrowtail v \iff (s, (\texttt{pure}[])|(p\ \$\ p^*)) \rightarrowtail v),
\end{array}
$$

where \$ is traditional concatenation (defined in Figure 2). In plain English, if a parser p is productive, then p^* is equivalent to either parsing ϵ to [] or parsing p and then p^*.

As a consequence, if the input string is empty, we can collapse the identity to

$$
\begin{array}{l}
\forall p\ v, \\
\qquad (\forall s\ v, (s,p) \rightarrowtail v \implies s \neq \epsilon) \implies \\
\qquad ((\epsilon,p^*) \rightarrowtail v \iff (\epsilon, \texttt{pure}[]) \rightarrowtail v).
\end{array}
$$

As a result the only possible epsilon-interpretation of p^* is [].

Epsilon-interpretation of Bind: While bind looks a bit complicated at its face, in fact, it is exactly the rule for epsilon-interpretation of concatenation, but specialized to data-dependent bind.

With a definition of epsilon-interpretation in hand, we can precisely state its specification:

Lemma 3.2:

$$
\forall p\ v, (\epsilon,p) \rightarrowtail v \iff v \in \|p\|_\epsilon .
$$

This is a straightforward proof by induction on the grammar p and inspection of the parse. For example, consider the case of alternation $p_l|p_r$; we have two grammars p_l and p_r, and from the inductive hypothesis, we know that $\forall v, (\epsilon,p_l) \rightarrowtail v \iff v \in \|p_l\|_\epsilon$ and $\forall v, (\epsilon,p_r) \rightarrowtail v \iff v \in \|p_r\|_\epsilon$. Consider a parse value v. For the forward direction, suppose $(\epsilon, p_l|p_r) \rightarrowtail v$. By the definition of the semantics of alternation, it must be the case that either $(\epsilon,p_l) \rightarrowtail v$ or $(\epsilon,p_r) \rightarrowtail v$. In either case, we know that v must be in $\|p_l|p_r\|_\epsilon = \|p_l\|_\epsilon +\!\!+ \|p_r\|_\epsilon$ from the inductive hypothesis; so the forward direction is proved. For the reverse direction, suppose that v is in $\|p_l|p_r\|_\epsilon = \|p_l\|_\epsilon + + \|p_r\|_\epsilon$. From the properties of list append, we know that v is in either $\|p_l\|_\epsilon$ or $\|p_r\|_\epsilon$. [5] In either case, we know that either $(\epsilon,p_l) \rightarrowtail v$ or $(\epsilon,p_r) \rightarrowtail v$ from the inductive hypothesis, so we can use either PAltL or PAltR to conclude that $(\epsilon, p_l|p_r) \rightarrowtail v$, and the reverse direction is also proved. Since we have proved both directions, we can conclude the bidirectional if-and-only-if: $\forall v, (\epsilon, p_l|p_r) \rightarrowtail v \iff v \in \|p_l|p_r\|_\epsilon$.

Brzozowski derivatives: We next define a derivative operator with respect to a character c, written as $\mathrm{D}_c\ p$ for an input grammar p. This is also defined as a syntax-directed recursive function over the structure of p, given in Figure 4. The derivative rules for $\emptyset$, ., pure, and alternation are equivalent to the derivative for traditional regular languages.

Derivative of iteration: In our formulation, the derivative of Kleene iteration is a bit simpler due again to the productivity restriction. This is best understood by again appealing to our iteration identity Lemma 3.1. First, since the parsing semantics only give meaning to productive interiors of Kleene iteration, we can assume that the interior parser p of iteration p^* is productive. From the definition of Lemma 3.1, we know the following:

$$
\begin{array}{l}
\forall s\ v, \\
\qquad (s,p^*) \rightarrowtail v \iff (s, (\texttt{pure}[])|(p\ \$\ p^*)) \rightarrowtail v.
\end{array}
$$

Next we consider the derivative of p^* with respect to a character c. By the above logic, we know that:

$$
\begin{array}{l}
\forall s\ v, \\
\qquad (s, \mathrm{D}_c\ p^*) \rightarrowtail v \iff \quad (s, \mathrm{D}_c\ (\texttt{pure}[])|(p\ \$\ p^*)) \rightarrowtail v.
\end{array}
$$

Since $\texttt{pure}\ v$ cannot accept input character c, we can further reduce the right-hand-side of the iff to just $p\ \$\ p^*$:

$$
\begin{array}{l}
\forall s\ v, \\
\qquad (s, \mathrm{D}_c\ p^*) \rightarrowtail v \quad \iff (s, \mathrm{D}_c\ (\texttt{pure}[])|(p\ \$\ p^*)) \rightarrowtail v \\
\qquad\qquad\qquad\qquad\ \ \iff (s, \mathrm{D}_c\ (p\ \$\ p^*)) \rightarrowtail v.
\end{array}
$$

Since p must be productive, we further know that the derivative must apply only to p and not to p^*:

$$
\begin{array}{l}
\forall s\ v, \\
\qquad (s, \mathrm{D}_c\ p^*) \rightarrowtail v \quad \iff (s, \mathrm{D}_c\ (p\ \$\ p^*)) \rightarrowtail v \\
\qquad\qquad\qquad\qquad\ \ \iff (\mathrm{D}_c\ p)\ \$\ p^*.
\end{array}
$$

[5] Indeed v could be in *both* lists but the reasoning does not change.

This is the definition we adopt for the derivative of p^* (i.e. we define $\mathrm{D}_c\ p^* = (\mathrm{D}_c\ p)\ \$\ p^*$).

Derivative of bind: The derivative of dependent bind $p \gg\!= f$ is a bit trickier than the other cases. If p parses the empty string ϵ to some value x, then the derivative must be applied to the resulting parser $f\ x$. In this case the resulting parser is $\mathrm{D}_c\ (f\ x)$. We lift this reasoning to all possible epsilon-interpretations of p by leveraging the definition of epsilon-interpretation and taking the overall alternation over all $x \in \|p\|_\epsilon$. Since $\|p\|_\epsilon$ returns a finite list, we know that this alternation is also finite. However, it's also possible that p parses the character, in which case the derivative must be applied to p. In this case the resulting parser is $\mathrm{D}_c\ p \gg\!= f$; in other words, p parses c and will eventually pass the result to f in the future. Both of these possibilities are valid, so we combine them with alternation.

Now that we have a definition of the derivative operator, we can formally state the correctness of the derivative:

Lemma 3.3:

$$\forall c\ s\ p\ v, (c :: s, p) \rightarrowtail v \iff (s, \mathrm{D}_c\ p) \rightarrowtail v.$$

This is a classic formulation, equivalent to traditional derivatives. The proof is a bit tricker but is also done by induction over the grammar p and inspection of the parse. Key to the proof is the correctness of epsilon-interpretation, which is necessary for the correctness of the bind case.

C. From derivatives to parsing

Now that we have machinery for Brzozowski derivatives, we can inductivly lift the derivative operator from a single character to a list of characters in the standard way. We define a parse function for a grammar p and a string s by applying the derivative with respect to s and returning the epsilon-interpretations of the resulting grammar:

$$\begin{aligned} \texttt{derivs}\ p\ (c :: s) &= \texttt{derivs}\ (\mathrm{D}_c\ p)\ s \\ \texttt{derivs}\ p\ \epsilon &= p \\ \texttt{parse}\ p\ s &= \|\texttt{derivs}\ p\ s\|_\epsilon \end{aligned}$$

The soundness and completeness of this parsing definition are straightforwardly defined:

Theorem 3.4:

$$\forall s\ p\ v, (s, p) \rightarrowtail v \iff v \in \texttt{parse}\ p\ s.$$

While weighty and impactful, this proof follows trivially from the correctness of the derivative operator and epsilon-interpretation.

IV. Dependent Grammars in Coq

We next detail our mechanization and implement a variety of popular parser combinator functions using dependent grammars. We first give a primer on Coq's functional language Gallina; readers familiar with Coq can skip forward to subsection IV-B.

```
Inductive nat : Type :=
  | Zero : nat
  | Succ : nat → nat.

Fixpoint add (l: nat) (r: nat) : nat :=
  match l with
  | Zero ⇒ r
  | Succ l' ⇒ Succ (add l' r)
  end.

Inductive list : Type → Type :=
  | Nil : forall {A: Type}, list A
  | Cons : forall {A: Type},
    A → list A → list A.

Fixpoint concat {A: Type} (xs: list A)
  : list A → list A :=
  match xs with
  | Nil ⇒
    fun r ⇒ r
  | Cons x xs' ⇒
    fun r ⇒ Cons x (concat xs' r)
  end.

Lemma concat_nil :
  forall A (xs: list A),
    concat xs Nil = xs.
Proof.
  induction xs.
  - trivial.
  - simpl.
    erewrite IHxs.
    trivial.
Qed.

Inductive vector : Type → nat → Type :=
  | VNil : forall {A: Type}, vector A Zero
  | VCons : forall
    {A: Type} {n: nat}
    (v: A) (vs: vector A n),
    vector A (Succ n).

Definition head_safe {A: Type} {n: nat}
  (vs: vector A (Succ n)) : A :=
  match vs with
  | VCons v _ ⇒ v
  end.
```

Fig. 5. Unary natural numbers, polymorphic lists, and length-indexed vectors in Coq. These definitions are all part of the standard library and we provide them for illustration.

A. Gallina basics

Coq [26] is an interactive theorem prover, which mixes a dependently-typed functional language (called Gallina) with support for interactive proofs. We mainly will focus on Gallina, which is similar to the functional subset of OCaml with two main differences. First, Gallina is both pure and total, so functions must always terminate and may not have implicit side effects, such as I/O or mutable references. Second, Gallina has rich *dependent types*, in which the return type of a function can depend on the *value* of its input argument.

We present some core features of Coq and Gallina by

example. Consider Figure 5, which defines Peano-style unary natural numbers, polymorphic lists, and length-indexed vectors. These definitions are all part of Coq's standard library and we redefine them for illustrative purposes.

a) Inductive types and recursion: Inductive types are a core building block of Gallina, analogous to data types in OCaml, with which the programmer can define a new type family and constructors for the type. For example, lines 1-3 define a new inductive type `nat` for natural numbers as well as two constructors `Zero` and `Succ` for the zero and successor cases. `Zero` takes no arguments and so has type `nat`, while `Succ` takes a single `nat` argument and so has type `nat -> nat`.

A key element of Gallina is that it supports pattern matching on inductive types, as well as a *fixpoint* primitive for recursion. For example, lines 5-9 define Peano addition of two `nats` by structural induction on the first argument. Because Gallina functions are total and must terminate, Coq uses a structural guardedness check to ensure that recursion is productive. The essense of this check is that recursive function calls always use structural elements of the input arguments. [6] In addition, because Gallina functions are total, pattern matches must consider all possible cases.

b) Indexed types and dependent types: Types in Gallina can also take arguments; these arguments are known as *indices* and the resulting type families are known as *indexed types*. Indexed types are a powerful and flexible mechanism, and in this work, we will use them for both generic programming and also to encode parser information at the type level. For example, line 11 defines a new type constructor `list` of type `Type -> Type`, in which the type index is used to determine the type of elements of the list. [7] Lines 12-14 define two inductive constructors for lists, `Nil` and `Cons`.

Both of these constructors make use of dependent types. In the case of `Nil`, line 12 gives `Nil` the *dependent type* of `forall {A: Type}, list A`, which is a dependent function from indices `A` to the list type `list A`. In other words, `Nil` can be applied to any type index `A` to produce a new list of `A`, namely the empty list. Second, because the declaration of `A` uses braces (i.e. `forall {A: Type}`), we are declaring that the index `A` is implicit, and asking Coq to try to infer the index from context. [8]

Because the type index `A` is declared implicit, we can use both `Nil` and `Cons` without explicitly providing the type index. For example, lines 16-23 define list concatenation of two lists by structural recursion on the first argument, and none of the uses of `Nil` or `Cons` need to specify the element type `A`.

c) Proofs and inversion: Coq also provides support for proving properties about Gallina functions. Lines 25-34 give a proof (`concat_nil`) for the fact that concatenation of `Nil` produces the input list. The proof uses *tactics*, such as `induction`, `simpl`, and `erewrite` to construct a proof term. We do not discuss tactics in detail because they are not part of our contribution; an interested reader can peruse [27]. [9]

Our tour of Coq concludes with a implementation of length-indexed vectors and a safe access function. Lines 36-41 define an inductive type for length-indexed vectors, in which the type `vector A n` is indexed by an element type `A` and a size `n`. Similar to lists, vectors are inductively defined by constructors for the empty and cons cases, but by contrast each case tracks the length of the vector as a type index. These constraints are useful because they allow for *dependent pattern matching*, in which the type indices for a particular element of a type constrain the possible constructors used to make the element. For example, lines 43-47 implement a type-safe operation for retrieving the head of a vector, provided that the vector is nonempty. This is done by constraining the input vector `vs` to have length `Succ n` for some value `n`, by setting the type of `vs` to `vector A (Succ n)` on line 44. Because of this constrained type, Coq infers that the only possible constructor for `vs` is `VCons`, since `VNil` requires the length index to be `Zero`. As a result the pattern match on lines 45-47 only needs to consider the `VCons` case; Coq statically analyzes the pattern match and infers that the omitted `VNil` case is dead code and so does not need to be included.

In interactive proofs, Coq also provides an *inversion* tactic, which performs analogous reasoning in the context of constructing a proof term. For example, in an interactive setting, suppose we have a variable `vs` of type `Vector A (Succ n)` in the proof context. If we apply the tactic `inversion vs`, Coq will inspect the type of `vs` and attempt to infer which inductive constructors must have been used to generate `vs`, such that the resulting type is `Vector A (Succ n)`. In this case, since only the `VCons` constructor can build terms of type `Vector A (Succ n)`, inversion will *invert* `vs` and conclude there must exist terms `v` and `vs'` such that `vs = VCons v vs'`. This style of reasoning is very useful because it allows proofs to infer the case of an inductive by the constraints on the inductive's type. We will use it to mechanize the inspection of parse results, such as the proof step shown in Lemma 3.2.

B. Syntax and Semantics

Next we describe the syntax and semantics for our Coq implementation of dependent grammars. We use `nat` and `list` to refer to Coq's standard library `nat` and `list` types (instead of the expository types in Figure 5). Where possible, we will use Coq's dependent function syntax to give

[6] It turns out that more complicated recursion schemes can also be implemented in Coq, but we will not need them.

[7] By contrast to traditional statically typed functional languages such as OCaml and Haskell, Gallina does not distinguish between value-level and type-level types.

[8] There are several techniques for manually specifying implicit arguments but we will not need to do so.

[9] An elegant feature of Coq and Gallina is that proofs and types are identical. While it is not necessary to understand for the purposes of this work, in fact, the definition of `concat_nil` introduces a dependent function from type indices and lists to equality proofs.

```
Inductive parser: Set → Type :=
| pFail : forall {A: Set}, parser A
| pAny : parser ascii
| pAlt : forall {A: Set}
  (l: parser A) (r: parser A), parser A
| pPure : forall {A: Set} (a: A), parser A
| pBind : forall {A B: Set}
  (p: parser A) (f: forall a: A, parser B),
  parser B
| pStar : forall {A: Set}
  (p: parser A),
  parser (list A).
```

Fig. 6. Coq inductive type for dependent grammars.

names to variables. For example, we would write the type of natural number increment, which takes as input a nat n and produces a nat as `forall (n: nat), nat`. Throughout our implementation, we represent characters using Coq's `ascii` type and strings as a list of ascii (i.e. `list ascii`). Surprisingly, we do not make use of ascii-specific functions in either our derivative operator or our epsilon-interpretation function. This is because we implement specific character matching as a derived combinator with `filter` (whereas traditionally specific character matching is part of the syntax of the regular language). [10]

Syntax: We represent the grammar syntax as an inductive datatype, given in Figure 6. One twist is that we index the type family by the return type of the parser. As a result we can build well-typed parsers by construction. While dependent programming can be tricky in practice, we found that the standard type constraints for monadic bind mechanized well. Using this definition, a parser for `pure 1` is defined as `pPure 1.` and has type `parser nat`.

1) Parsing relation: We represent the parsing relation also as an inductive datatype, but in Prop, Coq's sort of propositions. We name this relation `Parses`, given in Figure 7, and similar to the big-step semantics, `Parses` relates lists of characters and parsers to parse values. With this encoding, we can prove properties by induction over parses, as well as inspect the previous parse result by inversion.

2) Derivatives and epsilon interpretation: We implement the epsilon interpretation function and derivative function as recursive functions over `parsers`. The correctness definitions from before straightforwardly translate to Coq Props. We make heavy use of parsing relation inversion in our correctness proofs (which corresponds to inspecting the parse derivation). We also implement the lifted derivative function and the parsing function, which are straightforward recursive functions over lists.

Smart constructors: A key element to performant parsing with derivatives is *smart constructors*, which perform correctness-preserving optimizations on the fly. This is useful because without algebraic simplifications, the derivative operator can cause an exponential blow-up to the grammar [15]. We implement and prove sound several smart constructors, which are presented in Figure 8. Where possible, our derivative implementation uses these smart constructors instead of the plain grammar constructors.

[10]Of course the implementation of filter does use `ascii`'s decidable equality implementation to compare characters.

```
Inductive Parses :
    list ascii →
    forall {A: Set},
    parser A →
    A →
    Prop :=
| Any: forall (c: ascii),
     Parses (c :: nil) pAny c
| AltL:
  forall {A: Set}
    (s: list ascii) (l r: parser A) (v: A),
      Parses s p v →
      Parses s (pAlt l r) v
| AltR:
  forall {A: Set}
    (s: list ascii) (l r: parser A) (v: A),
      Parses s p' v →
      Parses s (pAlt l r) v
| Pure : forall {A: Set} (a: A),
     Parses nil (pPure a) a
| Bind:
  forall {A B: Set}
    (s s': list ascii) (p: parser A)
    (f: A → parser B) (v: A) (v': B),
      Parses s p v →
      Parses s' (f v) v' →
      Parses (s ++ s') (pBind p f) v'
| StarNone:
  forall {A: Set} (p: parser A),
     Parses nil (pStar p) nil
| StarIter:
  forall {A: Set}
    (s s' : list ascii)
    (p: parser A) (v: A) (vs: list A),
      Parses s p v →
      Parses s' (pStar p) vs →
      s <> nil →
      Parses (s ++ s') (pStar p) (v :: vs).
```

Fig. 7. Coq inductive type for the parsing relation.

We implement simplifications for alternation and for bind. For alternation, we check the arguments to see if either is `pFail`; if it is, we return the other argument. For bind i.e. `p >>= f`, we optimize if the first argument is either `pFail` or `pPure`. If it is `pFail`, we know that the overall bind will never succeed, so we replace it with `pFail`. If it is `pPure v` for some value v, we know that the new parser will be exactly `f v`, so we return this. Otherwise we construct a bind as normal.

Notably missing from these optimizations is a rule for Kleene star, in particular, the identity `pStar pFail = pFail`. In fact, that is because it is *false* for our semantics of Kleene star. This is because `pFail` has no possible semantic interpretation, whereas `pStar pFail` parses the empty string to the empty list.

```
(* Smart constructor for alternation *)
Definition alt {A: Set}
  (l r : parser A) : parser A :=
    match p, p' with
    | pFail, _ ⇒ p'
    | _, pFail ⇒ p
    | _, _ ⇒ pAlt p p'
    end.

Lemma alt_spec :
  forall {A: Set}
    (s: list ascii) (p p': parser A)
    (v: A),
      Parses s (alt p p') v ↔
      Parses s (pAlt p p') v.

(* Smart constructor for bind *)
Definition bind {A B: Set}
  (p : parser A) (f: A → parser B) : parser B :=
    match p with
    | pFail ⇒ pFail
    | (pPure v) ⇒ f v
    | _ ⇒ pBind p f
    end.

Lemma bind_spec :
  forall {A B: Set}
    (s: list ascii) (p: parser A)
    (f: A → parser B) (v: B),
      Parses s (bind p f) v ↔
      Parses s (pBind p f) v.
```

Fig. 8. Smart constructors for algebraic simplification, as well as their specifications.

This optimization could be added with an extra precondition to the semantics of Kleene star in the empty case.

C. Combinators

We next detail a variety of combinators for implementing useful derived parsers.

1) Basic combinators: First, for our most basic combinators, we implement `map` and `cat` in Figure 9 as defined in Figure 2.

We also provide some *notation* to make our Coq parsers easier to read. In particular, we write `$` and `@` as infix operators for map and concat and `||` and `>>=` as infix operators for alternation and bind.

We also define two derived binary operators `$>` and `<$`, which parse two grammars and discard one of the results. This is very useful in practice, for example to recognize characters that are part of the input but do not contribute to the parse value.

2) Dependent combinators: Next we implement two *dependent* combinators presented in Figure 10: `filter p f` from the overview, for conditional parsing, and `repeat n p`, which runs a parser `p` n times and produces a list of the parse results. While repeat is not obviously dependent, because it takes as input a `nat` n, we can use it as the argument to bind e.g. `digit >>= fun n => repeat n pAny`.

```
Definition map {A B: Set}
  (p: parser A) (f: A → B) : parser B :=
    pBind p (fun x ⇒ pPure (f x)).

Definition cat {A B: Set}
  (p: parser A) (p': parser B) : parser (A * B) :=
    pBind p (fun l ⇒ map p' (fun r ⇒ (l, r))).

Infix "$" := (cat) (at level 80).
Infix "@" := (map) (at level 80).
Infix "||" := (pAlt)
  (at level 50, left associativity).
Infix ">>=" := (pBind) (at level 80).

Notation "p1 $> p2" :=
  (cat p1 p2 @ fun '(_,x) ⇒ x) (at level 50).
Notation "p1 <$ p2" :=
  (cat p1 p2 @ fun '(x,_) ⇒ x) (at level 50).
```

Fig. 9. Basic combinators for map and concatenation, as well as some notation for more concise syntax. In Gallina, the type (A * B) is the type of pairs of A and B (i.e. the product of A and B).

```
Definition filter {A: Set}
  (p: parser A) (f: A → bool) : parser A :=
    p >>= fun a ⇒ if f a then pPure a else pFail.

Lemma filter_spec:
  forall {A: Set}
    (f: A → bool) (s: list ascii)
    (p: parser A) (v : A),
      Parses s (filter p f) v →
      f v = true.

Fixpoint repeat {A: Set}
  (n: nat) (p: parser A) : parser (list A) :=
    match n with
    | 0 ⇒ pPure nil
    | S n' ⇒
      (p $ (repeat n' p)) @ (fun '(x, xs) ⇒ x :: xs)
    end.

Lemma repeat_spec :
  forall {A: Set}
    (n: nat) (p: parser A)
    (s: list ascii) (vs: list A),
      Parses s (repeat n p) vs →
      length vs = n.
```

Fig. 10. Filter and repeat dependent combinators, as well as their specifications.

We further prove some interesting properties about `filter` and `repeat`. In particular, our lemma `filter_spec` states that if `filter p f` parses a value `v`, then it must be the case that $f\ v = \texttt{true}$. In addition, if `repeat p n` produces a list of results, the resulting list must have length n.

While these facts might seem obviously true from the definitions of `filter` and `repeat`, it's important to mechanize specifications for larger proofs. In fact, we will make use of this repeat specification in our proof of netstring functional correctness in section V.

```
Definition char (c: ascii) :=
    filter pAny (fun c' ⇒ eqb c c').

Definition many {A: Set}
  (p: parser A) : parser (list A) :=
    pStar p.

Fixpoint any {A: Set}
  (ps: list (parser A)) : parser A :=
    match ps with
    | nil ⇒ pFail
    | p :: ps' ⇒ p || (any ps')
    end.

Fixpoint sequence {A: Set}
  (ps: list (parser A)) : parser (list A) :=
    match ps with
    | nil ⇒
      pPure nil
    | p :: ps' ⇒
      (p $ (sequence ps')) ·(fun '(x, xs) ⇒ x :: xs)
    end.

Fixpoint fold {A B: Set}
  (f: A → B → B) (acc: B)
  (ps: list (parser A)) : parser B :=
    match ps with
    | nil ⇒ pPure acc
    | p :: ps' ⇒
       (p $ (fold f acc ps')) @ (fun '(a, b) ⇒ f a b)
    end.
```

Fig. 11. Popular monadic parser combinators using dependent grammars.

D. Parser combinator utilities

Finally we consider a variety of common monadic parser combinator utilities that are not necessarily dependent, but can be implemented in our framework of dependent grammars because we have access to monadic bind and pure. We discuss each in turn.

Char: `char c` accepts just a single character `c`. We implement this using `pAny` and `filter`.

Many: `many p` accepts 0 or more repetitions of a parser `p` and returns a list of the results. This is exactly our definition of `pStar`.

Any: `any ps` takes a list of parsers `ps` and returns either the union of all of their parse results, or just the first successful parse result from the elements of the list. While we don't have biased choice, we can support the union over all results. If the individual parsers of `ps` are disjoint then the behavior is equivalent.

Sequence: `sequence ps` takes a list of parsers `ps` and runs each of them in sequence, returning a list of their parse results. Just as `any` lifts alternation to a list of parsers, `sequence` lifts concatenation to a list of parsers.

Fold: `fold f ps base` generalizes `sequence` to an input combination function `f` and a base case `base`. Just as sequence separates the input parses with concatenation, `fold` also uses concatenation to join the parsers together.

Finally, with a library of parsing combinators in hand, we implement a certified netstring parser.

V. Case Study: Certified Netstrings

Netstrings [28] are a method of encoding strings in an easy-to-parse way. Netstrings prefix a string with its length as a decimal number, wrapping the string with a colon and terminating it with a comma. Netstrings can also be *nested*, in which several elements of a netstring are themselves netstrings.

For example, the string "13:hello, world!," is a valid netstring, while the strings "2:long,", "6:short," and "invalid" are not because their interior strings are too long, too short, and not length-prefixed, respectively. The string "12:2:ab,4:cdef,," is an example of a valid nested netstring.

This format is deceptively simple, known as calc-regular languages in the literature [29], and in fact, it is neither regular nor context free. We first discuss our implementation of a netstring parser, and then discuss its specification and the proof burden.

A. Implementation

We develop a parser for a single level of netstrings in Figure 12. Notice that while it can parse a nested netstring, it does not evaluate the validity of the interior length prefixes (if present), or the well-formedness of the interior separators. While this is interesting and a key part of the challenge for calc-regular languages, nested netstrings are not very well defined without having prior knowledge of the format structure. For example, should the string "9:ab4:cdef,," be rejected? If the interior format is two netstrings, then it should be rejected because the left-substring is not a valid netstring. If the interior format is only one netstring (or no interior netstrings at all), then it should be accepted, as the interior string is the correct length and well-formed.

More generally, if a particular nesting structure is known ahead of time, one can develop a valid netstring parser for that format by reusing the netstring parser presented here.

We define a netstring (with the `net_str` parser) as a number followed by a body. To parse a number (`num`), we parse zero or more digits into the corresponding decimal number using helper parsers `dig`, `digs`, and a conversion function `digs_2_num`. To parse a body, we take as input a nat `n` and parse `n` characters (of any value) between the netstring separators. This parser returns the interior body as its parse value.

B. Certification

While this parser is not terribly complicated, its functional specification is a bit more weighty. Because we are working within Coq, we can phrase (and prove) a rich functional specification:

```
Theorem net_str_spec:
  forall (s: list ascii) (v: list ascii),
    Parses s net_str v ↔
    exists (s': list ascii),
      s = s' ++ (":" :: v ++ ("," :: nil)) ∧
      Parses s' num (length v).
```

```
Definition dig : parser nat :=
   (char "0" @ fun _ ⇒ 0) ||
   (char "1" @ fun _ ⇒ 1) ||
   (char "2" @ fun _ ⇒ 2) ||
   (char "3" @ fun _ ⇒ 3) ||
   (char "4" @ fun _ ⇒ 4) ||
   (char "5" @ fun _ ⇒ 5) ||
   (char "6" @ fun _ ⇒ 6) ||
   (char "7" @ fun _ ⇒ 7) ||
   (char "8" @ fun _ ⇒ 8) ||
   (char "9" @ fun _ ⇒ 9).

Fixpoint digs_2_num (digs: list nat) : nat :=
   match digs with
   | nil ⇒ 0
   | d :: ds ⇒ d + digs_2_num ds * 10
   end.

Definition digs : parser (list nat) := pStar dig.

Definition num : parser nat :=
   digs @ fun x ⇒ digs_2_num (rev x).

Definition net_str_body (n: nat) :
  parser (list ascii) :=
   (char ":" $> ((repeat n pAny) <$ char ",")).

Definition net_str : parser (list ascii) :=
   num >>= net_str_body.
```

Fig. 12. Netstring parser implementation.

In plain English, if `net_str` parses a string `s` to a body `v`, then there must have existed a prefix `s'` that parses to the length of the body `v`, and the body was exactly wrapped by a colon and comma. Moreover we phrase this as an if-and-only-if, so we prove the reverse direction as well (i.e. the `net_str` parser will succeed if it has a well-formed body and a numeric prefix corresponding to the length of the body).

This proof was surprisingly involved. To get it to work, we had to reuse the `repeat` specification from section IV, as well as prove a new lemma about `repeat`. In particular we proved that if `p` *always* parses a particular number of characters n', then `repeat n p` parses $n * n'$ characters. This is necessary to show that `repeat n pAny` parses `n` characters (as `pAny` parses one character).

C. Future challenges

We leave open several directions for future work. First, it would be interesting to extend our netstring parser (and functional correctness specification) to nested netstrings. Second, while we do implement some optimizations in the form of smart constructors, we do not make a serious attempt to optimize the implementation. We ran some synthetic benchmarks and found that `vm_compute`ing our parsing function took about 60 seconds for a 1024 character netstring. This is surprisingly tractable but unrealistic for actual parsing applications. It would be interesting to debug this performance and experiment with extracting to OCaml, or implementing a more performant automata model (instead of relying on arbitrary Gallina functions for the bind operator). Third, our proof of netstring functional correctness was surprisingly difficult (given the simplicity of the format). While it was promising that we could prove such a rich specification at all, it's unlikely that our naive proof technique could scale to more complicated applications of parser combinators. It would be interesting to adapt the productivity efforts of [9] to a proof assistant, to reduce the proof burden for more complicated libraries, and to develop novel proof engineering centered around certification of parser combinator programs (such as those developed by Hammer [7]). Finally, another direction is to improve the expressiveness to forms of left-recursion by incorporating ideas in [22] and [23]. Both of these techniques require the end-user to provide extra static annotations in exchange for the extra expressiveness (by contrast, our technique does not). It would be interesting to try to incorporate these ideas while minimizing the extra annotations.

ACKNOWLEDGMENT

We thank our paper shepherd, Julien Vanegue, and our anonymous reviewers for excellent feedback on technical content and presentation. This work was supported by DARPA research grant HR0011-19-C-0073.

REFERENCES

[1] J. A. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM (JACM)*, vol. 11, no. 4, pp. 481–494, 1964.
[2] G. Hutton and E. Meijer, "Monadic parser combinators," *Functional Programming Paper Collection CPSC 521*, 2011.
[3] P. Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 1990, pp. 61–78.
[4] G. Hutton and E. Meijer, "Monadic parsing in haskell," *Journal of functional programming*, vol. 8, no. 4, pp. 437–444, 1998.
[5] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," 2001.
[6] "Parser combinators built for speed and memory efficiency," https://github.com/inhabitedtype/angstrom, accessed: 2021-02-07.
[7] "Parser combinators for binary formats, in c," https://github.com/UpstandingHackers/hammer, accessed: 2021-02-07.
[8] G. Couprie, "Nom, a byte oriented, streaming, zero copy, parser combinators library in rust," in *2015 IEEE Security and Privacy Workshops*. IEEE, 2015, pp. 142–148.
[9] N. R. Krishnaswami and J. Yallop, "A typed, algebraic approach to parsing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 379–393.
[10] A. Afroozeh and A. Izmaylova, "Iguana: a practical data-dependent parsing framework," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 267–268.
[11] T. Jim, Y. Mandelbaum, and D. Walker, "Semantics and algorithms for data-dependent grammars," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2010, pp. 417–430.
[12] T. Jim and Y. Mandelbaum, "A new method for dependent parsing," in *European Symposium on Programming*. Springer, 2011, pp. 378–397.
[13] S. Owens, J. Reppy, and A. Turon, "Regular-expression derivatives re-examined," *Journal of Functional Programming*, vol. 19, no. 2, pp. 173–190, 2009.
[14] M. Might, D. Darais, and D. Spiewak, "Parsing with derivatives: a functional pearl," *Acm sigplan notices*, vol. 46, no. 9, pp. 189–195, 2011.
[15] M. D. Adams, C. Hollenbeck, and M. Might, "On the complexity and performance of parsing with derivatives," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 224–236.
[16] S. Lasser, C. Casinghino, K. Fisher, and C. Roux, "Costar: a verified all

(*) parser," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 420–434.
[17] X. Jia, A. Kumar, and G. Tan, "A derivative-based parser generator for visibly pushdown grammars," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–24, 2021.
[18] R. Edelmann, J. Hamza, and V. Kunčak, "Zippy ll (1) parsing with derivatives," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1036–1051.
[19] D. Egolf, S. Lasser, and K. Fisher, "Verbatim++: verified, optimized, and semantically rich lexing with derivatives," in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2022, pp. 27–39.
[20] C. McBride and J. McKinna, "Seeing and doing," in *Presentation (given by McBride) at the Workshop on Termination and Type Theory, Hindås, Sweden*, 2002.
[21] "Strongly specified parser combinators," http://muaddibspace.blogspot.com/2009/04/strongly-specified-parser-combinators.html, accessed: 2021-02-07.
[22] N. A. Danielsson, "Total parser combinators," in *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, 2010, pp. 285–296.
[23] G. Allais, "agdarsec–total parser combinators."
[24] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: better, faster, stronger sfi for the x86," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 395–404.
[25] R. Harper, "Proof-directed debugging," *Journal of functional programming*, vol. 9, no. 4, pp. 463–469, 1999.
[26] T. Coq Development Team, "The Coq proof assistant, version 8.14," Available electronically at https://coq.inria.fr/, accessed: 2022-03-15.
[27] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013, available electronically at http://adam.chlipala.net/cpdt/.
[28] "Netstrings," http://cr.yp.to/proto/netstrings.txt, accessed: 2021-02-07.
[29] S. Lucks, N. M. Grosch, and J. König, "Taming the length field in binary data: calc-regular languages," in *2017 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2017, pp. 66–79.