A predicate transformer semantics of parser combinators

Tim Baanen
Vrije Universiteit Amsterdam

Wouter Swierstra
Utrecht University

1   Introduction

There is a significant body of work on parsing using combinators in functional programming languages [others?; Hut92; SD96; Wad85],. Yet how can we ensure that these parsers are correct? There is notably less work that attempts to answer this question [Dan10; Fir16].

Reasoning about such parser combinators is not at all trivial; they use a variety of effects: state to store the string being parsed; non-determinism to handle backtracking; and general recursion to deal with recursive grammars. Proof techniques, such as equational reasoning, that commonly used when reasoning about pure functional programs, are less suitable when verifying effectful programs.

In this paper, we explore a novel approach, drawing inspiration from recent work on algebraic effects [BP15; WSH14; McB15]. We demonstrate how to reason about all parsers uniformly using predicate transformers [SB19]. We extend our previous work that uses predicate transformer semantics to reason about a single effect to handle the combinations of effects used by parsers. Our semantics is modular, allowing us to introduce concepts only when they are needed, without having to rework the previous definitions. In particular, our careful treatment of general recursion lets us separate the partial correctness of the combinators from their termination cleanly. Most existing proofs require combinators to guarantee that the string being parsed decreases, conflating termination and correctness.

In particular, this paper makes the following novel contributions:

- The non-recursive fragment of regular expressions can be correctly parsed using non-determinism (Section 3); by combining non-determinism with general recursion (Section 4), support for the Kleene star can be added without compromising our previous definitions (Section 5). Although the resulting parser is not guaranteed to terminate, it is refined by another implementation using Brzozowski derivatives that does terminate (Section 6).

- Next, we show how this approach may be extended to handle context-free languages. To do so, we show how to write parsers using algebraic effects (Section 7), and map grammars to parsers (Section 8). Once again, we can cleanly separate the proofs of partial correctness (Section 9) and termination (Section 10).

All the programs and proofs in this paper are written in the dependently typed language Agda [Nor07].

2   Recap: algebraic effects and predicate transformers

Algebraic effects separate the syntax and semantics of effectful operations. In this paper, we will model the by taking the free monad over a given signature, describing certain operations. The type of such a signature is defined as follows:

```
record Sig : Set where
  constructor mkSig
  field
    C : Set
    R : C → Set
```

Here the type *C* contains the 'commands', or effectful operations that a given effect supports. For each command *c* : *C*, the type *R c* describes the possible responses. The structure on a signature is that of a container [AAG03]. For example, the following signature describes two operations: the non-deterministic choice between two values, *Choice*; and a failure operator, *Fail*.

```
data CNondet : Set where
  Choice : CNondet
  Fail    : CNondet
RNondet : CNondet → Set
RNondet Choice = Bool
RNondet Fail    = ⊥
Nondet = mkSig CNondet RNondet
```

We represent effectful programs that use a particular effect using the corresponding free monad:

```
data Free (e : Sig) (a : Set) : Set where
  Pure : a → Free e a
  Op : (c : C e) → (R e c → Free e a) → Free e a
```

This gives a monad, with the bind operator defined as follows:

```
_>>=_ : Free e a → (a → Free e b) → Free e b
Pure x >>= f = f x
Op c k >>= f = Op c (λ x → k x >>= f)
```

To facilitate programming with effects, we define the following smart constructors, sometimes referred to as generic effects in the literature [PP03]:

```
fail : Free Nondet a
fail = Op Fail λ ()
choice : Free Nondet a → Free Nondet a → Free Nondet a
choice S₁ S₂ = Op Choice λ b → if b then S₁ else S₂
```

In this paper, we will assign semantics to effectful programs by mapping them to predicate transformers. Each semantics will be computed by a fold over the free monad, mapping some predicate *P* : *a* → *Set* to a predicate on the result of the free monad to a predicate of the entire computation of type *Free* (*eff C R*) *a* → *Set*.

```
⟦·⟧. : ((c : C) → (R c → Set) → Set) →
  Free (mkSig C R) a → (a → Set) → Set
```

$$\llbracket Pure\ x \rrbracket_{alg}\ P\ =\ P\ x$$
$$\llbracket Op\ c\ k \rrbracket_{alg}\ P\ =\ alg\ c\ \lambda\ x\ \to\ \llbracket k\ x \rrbracket_{alg}\ P$$

The predicate transformer nature of these semantics becomes evident when we assume the type of responses R does not depend on the command $c : C$. The type of $alg : (c : C) \to (R\ c \to Set) \to Set$ then becomes $C \to (R \to Set) \to Set$, which is isomorphic to $(R \to Set) \to (C \to Set)$. Thus, alg has the form of a predicate transformer from postconditions of type $R \to Set$ into preconditions of type $C \to Set$. Two considerations cause us to define the types $alg : (c : C) \to (R\ c \to Set) \to Set$, and analogously $\llbracket \cdot \rrbracket_{alg} : Free\ (mkSig\ C\ R)\ a \to (a \to Set) \to Set$. By having the command as first argument to alg, we allow R to depend on C. Moreover, $\llbracket \cdot \rrbracket_{alg}$ computes semantics, so it should take a program $S : Free\ (mkSig\ C\ R)\ a$ as its argument and return the semantics of S, which is then of type $(a \to Set) \to Set$.

In the case of non-determinism, for example, we may want to require that a given predicate P holds for all possible results that may be returned:

$$ptAll : (c : CNondet) \to (RNondet\ c \to Set) \to Set$$
$$ptAll\ Fail\ P\ \ \ \ \ =\ \top$$
$$ptAll\ Choice\ P\ =\ P\ True\ \wedge\ P\ False$$

$$\llbracket \cdot \rrbracket_{all} : Free\ Nondet\ a \to (a \to Set) \to Set$$
$$\llbracket S \rrbracket_{all}\ =\ \llbracket \cdot \rrbracket.\ ptAll\ S$$

Predicate transformers provide a single semantic domain to relate programs and specifications. Throughout this paper, we will consider specifications consisting of a pre- and postcondition:

module *Spec* **where**
   record *Spec* $(a : Set)$: *Set* **where**
      constructor $[\_, \_]$
      field
         pre : *Set*
         post : $a \to Set$

Inspired by work on the refinement calculus, we can assign a predicate transformer semantics to specifications as follows:

$$\llbracket \cdot, \cdot \rrbracket_{spec} : Spec\ a \to (a \to Set) \to Set$$
$$\llbracket pre, post \rrbracket_{spec}\ P\ =\ pre\ \wedge\ (\forall\ o \to post\ o \to P\ o)$$

This computes the 'weakest precondition' necessary for a specification to imply that the desired postcondition P holds. In particular, the precondition *pre* should hold and any possible result satisfying the postcondition *post* should imply the postcondition P.

Finally, we use the refinement relation to compare programs and specifications:

$$\_\sqsubseteq\_\ :\ (pt_1\ pt_2 : (a \to Set) \to Set) \to Set$$
$$pt_1\ \sqsubseteq\ pt_2\ =\ \forall\ P \to pt_1\ P\ \to\ pt_2\ P$$

Together with the predicate transformer semantics we have defined above, this refinement relation can be used to relate programs to their specifications. The refinement relation is both transitive and reflexive.

3   Regular languages without recursion

To illustrate how to reason about non-deterministic code, we begin by defining a regular expression matcher. Initially, we will restrict ourselves to non-recursive regular expressions; we will add recursion in the next section.

    We begin by defining the *Regex* datatype for regular expressions as follows: An element of this type represents the syntax of a regular

> **data** *Regex* : *Set* **where**
>   *Empty*   : *Regex*
>   *Epsilon*   : *Regex*
>   *Singleton* : *Char* → *Regex*
>   _|_   : *Regex* → *Regex* → *Regex*
>   _·_   : *Regex* → *Regex* → *Regex*
>   _⋆   : *Regex* → *Regex*

Note that the *Empty* regular expression corresponds to the empty language, while the *Epsilon* expression only matches the empty string. Furthermore, our language for regular expressions is closed under choice (_|_), concatenation (_·_) and linear repetition denoted by the Kleene star (_⋆).

    What should our regular expression matcher return? A Boolean value is not particularly informative; yet we also choose not to provide an intrinsically correct definition, instead performing extrinsic verification using our predicate transformer semantics. The Tree data type below, captures a potential parse tree associated with a given regular expression:

> Tree : *Regex* → *Set*
> Tree *Empty*   = ⊥
> Tree *Epsilon*   = ⊤
> Tree (*Singleton* _) = *Char*
> Tree (*l* | *r*)   = *Either* (Tree *l*) (Tree *r*)
> Tree (*l* · *r*)   = *Pair* (Tree *l*) (Tree *r*)
> Tree (*r* ⋆)   = *List* (Tree *r*)

In the remainder of this section, we will develop a regular expression matcher with the following type:

> *match* : (*r* : *Regex*) (*xs* : *String*) → *Free Nondet* (Tree *r*)

Before we do so, however, we will complete our specification. Although the type above guarantees that we return a parse tree matching the regular expression *r*, there is no relation between the tree and the input string. To capture this relation, we define the following *Match* data type. A value of type *Match r xs t* states that the string *xs* is in the language given by the regular expression *r* as witnessed by the parse tree *t*:

> **data** *Match* : (*r* : *Regex*) → *String* → Tree *r* → *Set* **where**
>   *Epsilon*   : *Match Epsilon Nil tt*
>   *Singleton* : *Match* (*Singleton x*) (*x* :: *Nil*) *x*
>   *OrLeft*   : *Match l xs x* → *Match* (*l* | *r*) *xs* (*Inl x*)

$$\begin{array}{ll}
\textit{OrRight} & : \textit{Match } r \textit{ xs } x \ \rightarrow \ \textit{Match } (l \mid r) \textit{ xs } (\textit{Inr } x) \\
\textit{Concat} & : \textit{Match } l \textit{ ys } y \ \rightarrow \ \textit{Match } r \textit{ zs } z \ \rightarrow \\
& \quad \textit{Match } (l \cdot r) \ (\textit{ys} \mathbin{+\mkern-10mu+} \textit{zs}) \ (y \, , z) \\
\textit{StarNil} & : \textit{Match } (r \star) \textit{ Nil Nil} \\
\textit{StarConcat} & : \textit{Match } (r \cdot (r \star)) \textit{ xs } (y \, , \textit{ys}) \ \rightarrow \ \textit{Match } (r \star) \textit{ xs } (y :: \textit{ys})
\end{array}$$

Note that there is no constructor for *Match Empty xs ms* for any *xs* or *ms*, as there is no way to match the *Empty* language with a string *xs*. Similarly, the only constructor for *Match Epsilon xs ms* is where *xs* is the empty string *Nil*. There are two constructors that produce a *Match* for a regular expression of the form $l \mid r$, corresponding to the choice of matching either *l* or *r*.

The cases for concatenation and iteration are more interesting. Crucially the *Concat* constructor constructs a match on the concatenation of the strings *xs* and *zs* – although there may be many possible ways to decompose a string into two substrings. Finally, the two constructors for the Kleene star, *r* match zero (*StarNil*) or many (*StarConcat*) repetitions of *r*.

We will now turn our attention to the *match* function. The complete definition, by induction on the argument regular expression, can be found in Figure 1. Most of the cases are straightforward—the most difficult case is that for concatenation, where we non-deterministically consider all possible splittings of the input string *xs* into a pair of strings *ys* and *zs*. The *allSplits* function, defined below, computes all possible splittings:

$$\begin{array}{l}
\textit{allSplits} : (xs : \textit{List a}) \ \rightarrow \ \textit{Free Nondet } (\textit{List a} \times \textit{List a}) \\
\textit{allSplits Nil} \ = \ \textit{Pure } (\textit{Nil} \, , \textit{Nil}) \\
\textit{allSplits } (x :: xs) \ = \ \textit{choice} \\
\quad (\textit{Pure } (\textit{Nil} \, , (x :: xs))) \\
\quad (\textit{allSplits xs} \mathbin{>\!\!>\!\!=} \lambda \, \{ (\textit{ys} \, , zs) \ \rightarrow \ \textit{Pure } ((x :: ys) \, , zs) \})
\end{array}$$

Finally, we cannot yet handle the case for the Kleene star. We could attempt to mimick the case for concatenation, attempting to match $r \cdot (r)$. This definition, however, is rejected by Agda as it is not structurally recursive. For now, however, we choose to simply fail on all such regular expressions.

Still, we can prove that the *match* function behaves correctly on all regular expressions that do not contain iteration. The *hasNo*∗ predicate holds of all such iteration-free regular expressions:

$$\textit{hasNo}{*} : \textit{Regex} \ \rightarrow \ \textit{Set}$$

To verify our matcher is correct, we need to prove that it satisfies the specification consisting of the following pre- and postcondition:

$$\begin{array}{l}
\textit{pre} : (r : \textit{Regex}) \ (xs : \textit{String}) \ \rightarrow \ \textit{Set} \\
\textit{pre r xs} \ = \ \textit{hasNo}{*} \ r \\
\textit{post} : (r : \textit{Regex}) \ (xs : \textit{String}) \ \rightarrow \ \textit{Tree } r \ \rightarrow \ \textit{Set} \\
\textit{post} \ = \ \textit{Match}
\end{array}$$

The main correctness result can now be formulated as follows:

$$\textit{matchSound} : \forall \, r \textit{ xs} \ \rightarrow \ [\![(\textit{pre r xs}), (\textit{post r xs})]\!]_{\text{spec}} \sqsubseteq [\![\textit{match r xs}]\!]_{\text{all}}$$

This lemma guarantees that all the parse trees computed by the *match* function satisfy the *Match* relation, provided the input regular expression does not contain iteration. Although we have

$$
\begin{aligned}
&match : (r : Regex)\,(xs : String) \rightarrow Free\,Nondet\,(Tree\,r) \\
&match\;Empty \qquad\quad xs \qquad\qquad\;\; = fail \\
&match\;Epsilon \qquad\;\; Nil \qquad\qquad\;\; = Pure\;tt \\
&match\;Epsilon \qquad\;\; (\_ :: \_) \qquad\;\; = fail \\
&match\;(Singleton\;c)\;Nil \qquad\qquad = fail \\
&match\;(Singleton\;c)\;(x :: Nil) \qquad \textbf{with } c \overset{?}{=} x \\
&match\;(Singleton\;c)\;(.c :: Nil) \quad\;\; |\;yes\;refl\; = Pure\;c \\
&match\;(Singleton\;c)\;(x :: Nil) \qquad |\;no\;\neg p\;\; = fail \\
&match\;(Singleton\;c)\;(\_ :: \_ :: \_) = fail \\
&match\;(l\;|\;r) \qquad\quad\; xs \qquad\qquad = choice\;(Inl\;\langle\$\rangle\;match\;l\;xs)\;(Inr\;\langle\$\rangle\;match\;r\;xs) \\
&match\;(l\;\cdot\;r) \qquad\quad xs \qquad\qquad = \text{do} \\
&\quad (ys\,,zs) \leftarrow allSplits\;xs \\
&\quad y \leftarrow match\;l\;ys \\
&\quad z \leftarrow match\;r\;zs \\
&\quad Pure\;(y\,,z) \\
&match\;(r\;\star)\;xs\; = fail
\end{aligned}
$$

<center>Figure 1: The definition of the match function</center>

omitted the proof, we will sketch the key lemmas and definitions that are necessary to complete it.

First of all, we quickly run into problems as soon as we need to reason about programs composed using the monadic bind operator. In particular, when verifying the case for $l \cdot r$, we would like to use our induction hypotheses on two recursive calls. To do, we prove the following lemma that allows us to replace the semantics of a composite program built using the monadic bind operation with the composition of the underlying predicate transformers:

$$
\begin{aligned}
&consequence : \forall\,pt\,(mx : Free\,es\,a)\,(f : a \rightarrow Free\,es\,b) \rightarrow \\
&\quad [\![mx]\!]_{pt}\,(\lambda\,x \rightarrow [\![f\,x]\!]_{pt}\,P) \equiv [\![mx \ggg f]\!]_{pt}\,P
\end{aligned}
$$

Substituting along this equality gives us the lemmas we need to deal with the $\_\ggg\_$ operator:

$$
\begin{aligned}
&wpToBind : (mx : Free\,es\,a)\,(f : a \rightarrow Free\,es\,b) \rightarrow \\
&\quad [\![mx]\!]_{pt}\,(\lambda\,x \rightarrow [\![f\,x]\!]_{pt}\,P) \rightarrow [\![mx \ggg f]\!]_{pt}\,P \\
&wpFromBind : (mx : Free\,es\,a)\,(f : a \rightarrow Free\,es\,b) \rightarrow \\
&\quad [\![mx \ggg f]\!]_{pt}\,P \rightarrow [\![mx]\!]_{pt}\,(\lambda\,x \rightarrow [\![f\,x]\!]_{pt}\,P)
\end{aligned}
$$

The correctness proof for match closely matches the structure of match (and by extension allSplits). It uses the same recursion on Regex as in the definition of match. Since we make use of allSplits in the definition, we first give its correctness proof.

$$
\begin{aligned}
&allSplitsPost : String \rightarrow String \times String \rightarrow Set \\
&allSplitsPost\;xs\,(ys\,,zs) = xs \equiv ys \mathbin{+\!\!+} zs \\
&allSplitsSound : \forall\,xs \rightarrow [\![\top,(allSplitsPost\;xs)]\!]_{\mathrm{spec}} \sqsubseteq [\![allSplits\;xs]\!]_{\mathrm{all}}
\end{aligned}
$$

We refer to the accompanying code for the complete details of these proofs.

4   General recursion and non-determinism

The matcher we have defined in the previous section is unfinished, since it is not able to handle regular expressions that incorporate the Kleene star. The fundamental issue is that the Kleene star allows for arbitrarily many distinct matchings in certain cases. For example, matching *Epsilon* $\star$ with the empty string `""` will allow for repeating the *Epsilon* arbitrarily often, since *Epsilon* $\cdot$ (*Epsilon* $\star$) is equivalent to both *Epsilon* and *Epsilon* $\star$. Thus, we cannot implement *match* on the _$\star$ operator in the naïve way.

What we will do instead is to deal with the recursion as an effect. A recursively defined (dependent) function of type $(i : I) \to O\ i$ can instead be given as an element of the type $(i : I) \to Free\ (Rec\ I\ O)\ (O\ i)$, where *Rec I O* is the signature of general recursion [McB15]:

$$Rec : (I : Set)\ (O : I \to Set) \to Sig$$
$$Rec\ I\ O\ =\ mkSig\ I\ O$$

Defining **match** in the *Free* (*Rec* _ _) monad is not sufficient to implement it fully either, since replacing the effect *Nondet* with *Rec* does not allow for nondeterminism anymore. While the Kleene star might work, the other parts of **match** do not work anymore. We need a way to combine effects.

We can combine two effects in a straightforward way: given signatures *mkSig C_1 R_1* and *mkSig C_2 R_2*, we can define a combined signature by taking the disjoint union of the commands and responses, resulting in *mkSig* (*Either C_1 C_2*) [R_1 , R_2], where [R_1 , R_2] is the unique map given by applying R_1 to values in C_1 and R_2 on C_2 [WSH14]. If we want to support more effects, we can repeat this process of disjoint unions, but this quickly becomes somewhat cumbersome. For example, the disjoint union construction is associative semantically, but not syntactically. If two programs have the same set of effects that is associated differently, we cannot directly compose them.

Instead of building a new effect type, we modify the *Free* monad to take a list of signatures instead of a single signature. The *Pure* constructor remains as it is, while the *Op* constructor additionally takes an index into the list to specify which effect is invoked.

> **data** *Free* (*es* : *List Sig*) (*a* : *Set*) : *Set* **where**
>   *Pure* : $a \to$ *Free es a*
>   *Op* : $(i : e \in es)\ (c : C\ e)\ (k : R\ e\ c \to$ *Free es a*) $\to$ *Free es a*

By using a list of effects instead of allowing arbitrary disjoint unions, we have effectively chosen that the disjoint unions canonically associate to the right. Since the disjoint union is also commutative, it would be cleaner to have the collection of effects be unordered as well. Unfortunately, Agda does not provide a multiset type that is easy to work with.

We choose to use the same names and almost the same syntax for this new definition of *Free*, since all definitions that use the old version can be ported over with almost no change. Thus, we will not repeat definitions such as _$\gg\!=$_ and *consequence* for the new *Free* type.

Most of this bookkeeping can be inferred by Agda's typeclass inference, so we make the indices instance arguments, indicated by the double curly braces {{}} surrounding the arguments.

> *fail* : {{ *iND* : *Nondet* $\in$ *es* }} $\to$ *Free es a*
> *fail* {{ *iND* }} $=$ *Op iND Fail* λ ()

$$choice : \{\{ iND : Nondet \in es \}\} \rightarrow Free\ es\ a \rightarrow Free\ es\ a \rightarrow Free\ es\ a$$
$$choice\ \{\{ iND \}\}\ S_1\ S_2\ =\ Op\ iND\ Choice\ \lambda\ b\ \rightarrow\ \text{if}\ b\ \text{then}\ S_1\ \text{else}\ S_2$$
$$call : \{\{ iRec : Rec\ I\ O \in es \}\} \rightarrow (i : I) \rightarrow Free\ es\ (O\ i)$$
$$call\ \{\{ iRec \}\}\ i\ =\ Op\ iRec\ i\ Pure$$

For convenience of notation, we introduce the $\_ \overset{es}{\leftrightarrow} \_$ notation for general recursion, i.e. Kleisli arrows into *Free* (*Rec* $\_\_$ :: *es*).

$$\_ \overline{\leftrightarrow} \_ : (C : Set)\ (es : List\ Sig)\ (R : C \rightarrow Set) \rightarrow Set$$
$$C \overset{es}{\leftrightarrow} R\ =\ (c : C)\ \rightarrow\ Free\ (mkSig\ C\ R\ ::\ es)\ (R\ c)$$

With the syntax for combinations of effects defined, let us turn to semantics. Since the weakest precondition predicate transformer for a single effect is given as a fold over the effect's predicate transformer, the semantics for a combination of effects can be given as a fold over a (dependent) list of predicate transformers.

> **record** *PT* (*e* : *Sig*) : *Set* **where**
>   **constructor** *mkPT*
>   **field**
> $pt : (c : C\ e) \rightarrow (R\ e\ c \rightarrow Set) \rightarrow Set$
> $mono : \forall\ c\ P\ P' \rightarrow P \subseteq P' \rightarrow pt\ c\ P \rightarrow pt\ c\ P'$
> **data** *PTs* : *List Sig* $\rightarrow$ *Set* **where**
>   *Nil* : *PTs Nil*
> $\_::\_ : PT\ e \rightarrow PTs\ es \rightarrow PTs\ (e\ ::\ es)$

The record type *PT* not only contains a predicate transformer *pt*, but also a proof that *pt* is monotone in its predicate. The requirement of monotonicity is needed to prove the lemma terminates-fmap we introduce later, and makes intuitive sense: if the precondition holds for a certain postcondition, a weaker postcondition should also have its precondition hold.

Given a such a list of predicate transformers, defining the semantics of an effectful program is a straightforward generalization of the previously defined semantics $[\![\cdot]\!]_.$. The *Pure* case is identical, and in the *Op* case we can apply the predicate transformer returned by the *lookupPT* helper function.

> $lookupPT : (pts : PTs\ es)\ (i : mkSig\ C\ R \in es) \rightarrow$
> $(c : C) \rightarrow (R\ c \rightarrow Set) \rightarrow Set$
> $lookupPT\ (pt\ ::\ pts) \in \text{Head}\ =\ PT.pt\ pt$
> $lookupPT\ (pt\ ::\ pts)\ (\in \text{Tail}\ i)\ =\ lookupPT\ pts\ i$

This results in the following definition of the semantics for combinations of effects.

> $[\![\cdot]\!]_. : (pts : PTs\ es) \rightarrow Free\ es\ a \rightarrow (a \rightarrow Set) \rightarrow Set$
> $[\![Pure\ x]\!]_{pts}\ P\ =\ P\ x$
> $[\![Op\ i\ c\ k]\!]_{pts}\ P\ =\ lookupPT\ pts\ i\ c\ \lambda\ x\ \rightarrow\ [\![k\ x]\!]_{pts}\ P$

The effects we are planning to use for *match* are a combination of nondeterminism and general recursion. We re-use the *ptAll* semantics of nondeterminism, packaging them in a *PT* record.

However, it is not as easy to give a predicate transformer for general recursion, since the intended semantics of a recursive call depend on the function that is being called, i.e. the function that is being defined.

As long as we have a specification of a function of type $(i : I) \to O\,i$, for example in terms of a relation of type $(i : I) \to O\,i \to Set$, we are able to define a predicate transformer:

$$ptRec : ((i : I) \to O\,i \to Set) \to PT\,(Rec\,I\,O)$$
$$PT.pt\,(ptRec\,R)\,i\,P \,=\, \forall\,o \to R\,i\,o \to P\,o$$
$$PT.mono\,(ptRec\,R)\,c\,P\,P'\,imp\,asm\,o\,h \,=\, imp\,\_\,(asm\,\_\,h)$$

In the case of verifying the *match* function, the *Match* relation will play the role of *R*. If we use *ptRec R* as a predicate transformer to check that a recursive function satisfies the relation *R*, then we are proving partial correctness, since we assume each recursive call succesfully returns a correct value according to the relation *R*.

5   Recursively parsing every regular expression

To deal with the Kleene star, we rewrite *match* as a generally recursive function using a combination of effects. Since *match* makes use of *allSplits*, we also rewrite that function to use a combination of effects. The types become:

$$allSplits : \{\!\{\, iND : Nondet \in es \,\}\!\} \to List\,a \to Free\,es\,(List\,a \times List\,a)$$
$$match : \{\!\{\, iND : Nondet \in es \,\}\!\} \to Regex \times String \overset{es}{\leadsto} \mathrm{Tree} \circ Pair.fst$$

Since the index argument to the smart constructor is inferred by Agda, the only change in the definition of *match* and *allSplits* will be that *match* now implements the Kleene star:

$$match\,((r \star)\,,Nil) \,=\, Pure\,Nil$$
$$match\,((r \star)\,,xs@\,(\_\, ::\, \_)) \,=\, \mathrm{do}$$
$$\quad (y\,,ys) \leftarrow call\,((r \cdot (r \star))\,,xs)$$
$$\quad Pure\,(y\, ::\, ys)$$

The effects we need to use for running *match* are a combination of nondeterminism and general recursion. As discussed, we first need to give the specification for *match* before we can verify a program that performs a recursive *call* to *match*.

$$matchSpec : (r,xs : Pair\,Regex\,String) \to \mathrm{Tree}\,(Pair.fst\,r,xs) \to Set$$
$$matchSpec\,(r\,,xs)\,ms \,=\, Match\,r\,xs\,ms$$
$$[\![\cdot]\!]_{\mathrm{match}} : Free\,(Rec\,(Pair\,Regex\,String)\,(\mathrm{Tree} \circ Pair.fst) :: Nondet :: Nil)\,a \to$$
$$\quad (a \to Set) \to Set$$
$$[\![S]\!]_{\mathrm{match}} \,=\, [\![S]\!]_{ptRec\,matchSpec\,::\,ptAll\,::\,Nil}$$

We can reuse exactly the same proof to show *allSplits* is correct, since we use the same semantics for the effects in *allSplits*. Similarly, the correctness proof of *match* will be the same on all cases except the Kleene star. Now we are able to prove correctness of *match* on a Kleene star.

$$matchSound~((r\star),Nil)\qquad P~(preH,postH)~=~postH~\_~StarNil$$
$$matchSound~((r\star),(x::xs))~P~(preH,postH)~o~H~=~postH~\_~(StarConcat~H)$$

At this point, we have defined a parser for regular languages and formally proved that its output is always correct. However, *match* does not necessarily terminate: if *r* is a regular expression that accepts the empty string, then calling *match* on *r* ⋆ and a string *xs* results in the first nondeterministic alternative being an infinitely deep recursion.

The next step is then to write a parser that always terminates and show that *match* is refined by it. Our approach is to do recursion on the input string instead of on the regular expression.

6   Termination, using derivatives

Since recursion on the structure of a regular expression does not guarantee termination of the parser, we can instead perform recursion on the string to be parsed. To do this, we make use of an operation on languages called the Brzozowski derivative.

Definition 1 ([Brz64]) The Brzozowski derivative of a formal language *L* with respect to a character *x* consists of all strings *xs* such that *x* :: *xs* ∈ *L*.

Importantly, if *L* is regular, so are all its derivatives. Thus, let *r* be a regular expression, and *d r /d x* an expression for the derivative with respect to *x*, then *r* matches a string *x* :: *xs* if and only if *d r /d x* matches *xs*. This suggests the following implementation of matching an expression *r* with a string *xs*: if *xs* is empty, check whether *r* matches the empty string; otherwise let *x* be the head of the string and *xs'* the tail and go in recursion on matching *d r /d x* with *xs'*.

The first step in implementing a parser using the Brzozowski derivative is to compute the derivative for a given regular expression. Following Brzozowski [Brz64], we use a helper function *ε*? that decides whether an expression matches the empty string.

$$\varepsilon?~:~(r~:~Regex)~\rightarrow~Dec~(\textstyle\sum~(\text{Tree}~r)~(Match~r~Nil))$$

The definitions of *ε*? is given by structural recursion on the regular expression, just as the derivative operator is:

$$d\_/d\_~:~Regex~\rightarrow~Char~\rightarrow~Regex$$
$$d~Empty~/d~c~=~Empty$$
$$d~Epsilon~/d~c~=~Empty$$
$$d~Singleton~x~/d~c~\textbf{with}~c~\overset{?}{=}~x$$
$$...~|~yes~p~=~Epsilon$$
$$...~|~no~\neg p~=~Empty$$
$$d~l\cdot r~/d~c~\textbf{with}~\varepsilon?~l$$
$$...~|~yes~p~=~((d~l~/d~c)\cdot r)~|~(d~r~/d~c)$$
$$...~|~no~\neg p~=~(d~l~/d~c)\cdot r$$
$$d~l~|~r~/d~c~=~(d~l~/d~c)~|~(d~r~/d~c)$$
$$d~r\star~/d~c~=~(d~r~/d~c)\cdot(r\star)$$

In order to use the derivative of *r* to compute a parse tree for *r*, we need to be able to convert a tree for *d r /d x* to a tree for *r*. We do this with the function *integralTree*:

$$integralTree~:~(r~:~Regex)~\rightarrow~\text{Tree}~(d~r~/d~x)~\rightarrow~\text{Tree}~r$$

We can also define it with exactly the same case distinction as we used to define $d\_/d\_$.

The code for the parser, *dmatch*, itself is very short. As we sketched, for an empty string we check that the expression matches the empty string, while for a non-empty string we use the derivative to perform a recursive call.

$$dmatch : \{\!\{\, iND : Nondet \in es \,\}\!\} \rightarrow Regex \times String \overset{es}{\rightsquigarrow} \mathrm{Tree} \circ Pair.fst$$
$$dmatch\ (r\ ,\ Nil)\ \textbf{with}\ \varepsilon?\ r$$
$$...\ |\ yes\ (ms\ ,\ \_)\ =\ Pure\ ms$$
$$...\ |\ no\ \neg p\qquad =\ fail$$
$$dmatch\ (r\ ,\ (x :: xs))\ =\ integralTree\ r\ \langle\$\rangle\ call\ ((d\ r\ /d\ x)\ ,\ xs)$$

Since *dmatch* always consumes a character before going in recursion, we can easily prove that each recursive call only leads to finitely many other calls. This means that for each input value we can unfold the recursive step in the definition a bounded number of times and get a computation with no recursion. Intuitively, this means that *dmatch* terminates on all input. If we are going to give a formal proof of termination, we should first determine the correct formalization of this notion. For that, we need to consider what it means to have no recursion in the unfolded computation. A definition for the *while* loop using general recursion looks as follows:

$$while : \{\!\{\, iRec : Rec\ a\ (K\ a) \in es \,\}\!\} \rightarrow$$
$$\quad (a \rightarrow Bool) \rightarrow (a \rightarrow a) \rightarrow (a \rightarrow Free\ es\ a)$$
$$while\ cond\ body\ i\ =\ \mathrm{if}\ cond\ i\ \mathrm{then}\ Pure\ i\ \mathrm{else}\ (call\ (body\ i))$$

We would like to say that some *while* loops terminate, yet the definition of *while* always contains a *call* in it. Thus, the requirement should not be that there are no more calls left, but that these calls are irrelevant.

Intuitively, we could say that a definition S calling f terminates if we make the unfolded definition into a *Partial* computation by replacing *call* with *fail*, the definition terminates if the *Partial* computation still works the same, i.e. it refines S. However, this mixes the concepts of correctness and termination. We want to see that the *Partial* computation gives some output, without caring about which output this is. Thus, we should only have a trivial postcondition. We formalize this idea in the terminates-in predicate.

$$terminates\text{-}in : (pts : PTs\ es)$$
$$\quad (f : C \overset{es}{\rightsquigarrow} R)\ (S : Free\ (mkSig\ C\ R :: es)\ a) \rightarrow \mathbb{N} \rightarrow Set$$
$$terminates\text{-}in\ pts\ f\ (Pure\ x)\ n\ =\ \top$$
$$terminates\text{-}in\ pts\ f\ (Op \in \mathrm{Head}\ c\ k)\ Zero\ =\ \bot$$
$$terminates\text{-}in\ pts\ f\ (Op \in \mathrm{Head}\ c\ k)\ (Succ\ n)\ =$$
$$\quad terminates\text{-}in\ pts\ f\ (f\ c \ggg k)\ n$$
$$terminates\text{-}in\ pts\ f\ (Op\ (\in \mathrm{Tail}\ i)\ c\ k)\ n\ =$$
$$\quad lookupPT\ pts\ i\ c\ (\lambda\ x \rightarrow terminates\text{-}in\ pts\ f\ (k\ x)\ n)$$

Since *dmatch* always consumes a character before going in recursion, we can bound the number of recursive calls with the length of the input string. The proof goes by induction on this string. Unfolding the recursive *call* gives *integralTree* $\langle\$\rangle$ *dmatch* $(d\ r\ /d\ x\ ,\ xs)$, which we rewrite using the associativity monad law in a lemma called terminates-fmap.

$dmatchTerminates$: $(r : Regex)$ $(xs : String)$ $\rightarrow$
    terminates-in $(ptAll :: Nil)$ $(dmatch)$ $(dmatch\ (r , xs))$ $(length\ xs)$
$dmatchTerminates\ r\ Nil$ **with** $\varepsilon?\ r$
$dmatchTerminates\ r\ Nil\ |\ yes\ p\ =\ tt$
$dmatchTerminates\ r\ Nil\ |\ no\ \neg p\ =\ tt$
$dmatchTerminates\ r\ (x :: xs)\ =$ terminates-fmap $(length\ xs)$
    $(dmatch\ ((d\ r\ /d\ x)\ , xs))$
    $(dmatchTerminates\ (d\ r\ /d\ x)\ xs)$

To show partial correctness of *dmatch*, we can use the transitivity of the refinement relation. If we apply transitivity, it suffices to show that *dmatch* is a refinement of *match*. Our first step is to show that the derivative operator is correct, i.e. *d r /d x* matches those strings *xs* such that *r* matches *x* :: *xs*.

$derivativeCorrect$: $\forall\ r\ \rightarrow$
    $Match\ (d\ r\ /d\ x)\ xs\ y\ \rightarrow\ Match\ r\ (x :: xs)\ (integralTree\ r\ y)$

The proof mirrors the definitions of these functions, being structured as a case distinction on the regular expression.

Before we can prove the correctness of *dmatch* in terms of *match*, it turns out that we also need to describe *match* itself better. The meaning of our goal, to show that *match* is refined by *dmatch*, is to prove that the output of *dmatch* is a subset of that of *match*. Since *match* makes use of *allSplits*, we first prove that all splittings of a string *xs* are in the output of *allSplits xs*. This following lemma and *allSplitsSound* together show that calling *allSplits xs* is equivalent, under the semantics $[\![\cdot]\!]_{\mathrm{match}}$, to its specification $[\ \top\ ,\ allSplitsPost\ xs\ ]$.

$allSplitsComplete$: $(xs : String)\ \rightarrow\ [\![allSplits\ xs]\!]_{\mathrm{match}}\ \sqsubseteq\ [\![\top, allSplitsPost]\!]_{\mathrm{spec}}\ ]$

The proof mirrors *allSplits*, performing induction on *xs*.

Using the preceding lemmas, we can prove the partial correctness of *dmatch* by showing it refines *match*:

$dmatchSound$: $\forall\ r\ xs\ \rightarrow\ [\![match\ (r , xs)]\!]_{\mathrm{match}}\ \sqsubseteq\ [\![dmatch\ (r , xs)]\!]_{\mathrm{match}}$

Since we need to perform the case distinctions of *match* and of *dmatch*, the proof is longer than that of *matchSoundness*. Despite the length, most of it consists of performing the case distinction, then giving a simple argument for each case. Therefore, we omit the proof.

With the proof of *dmatchSound* finished, we can conclude that *dmatch* always returns a correct parse tree, i.e. that *dmatch* is sound. However, *dmatch* is not complete with respect to the *Match* relation: since *dmatch* never makes a nondeterministic choice, it will not return all possible parse trees as specified by *Match*, only the first tree that it encounters. Still, we can express the property that *dmatch* finds a parse tree if it exists. In other words, we will show that if there is a valid parse tree, *dmatch* returns any parse tree (and this is a valid tree by *dmatchSound*). To express that *dmatch* returns something, we use a trivially true postcondition, and replace the demonic choice of the *ptAll* semantics with the angelic choice of *ptAny*:

$dmatchComplete$: $\forall\ r\ xs\ y\ \rightarrow\ Match\ r\ xs\ y\ \rightarrow$
    $[\![dmatch\ (r , xs)]\!]_{ptRec\ matchSpec\ ::\ ptAny\ ::\ Nil}\ (\lambda\ \_\ \rightarrow\ \top)$

The proof is short, since *dmatch* can only *fail* when it encounters an empty string and a regex that does not match the empty string, contradicting the assumption immediately:

> *dmatchComplete r Nil y H* **with** ε? *r*
> *...* | *yes p* = *tt*
> *...* | *no* ¬*p* = ¬*p* (_ , *H*)
> *dmatchComplete r* (*x* :: *xs*) *y H y' H'* = *tt*

Note that *dmatchComplete* does not show that *dmatch* terminates: the semantics for the recursive case assume that *dmatch* always returns some value *y'*.

In the proofs of *dmatchSound* and *dmatchComplete*, we demonstrate the power of predicate transformer semantics for effects: by separating syntax and semantics, we can easily describe different aspects (soundness and completeness) of the one definition of *dmatch*. Since the soundness and completeness result we have proved imply partial correctness, and partial correctness and termination imply total correctness, we can conclude that *dmatch* is a totally correct parser for regular languages.

Note the correspondences of this section with a Functional Pearl by Harper [Har99], which also uses the parsing of regular languages as an example of principles of functional software development. Starting out with defining regular expressions as a data type and the language associated with each expression as an inductive relation, both use the relation to implement essentially the same *match* function, which does not terminate. In both, the partial correctness proof of *match* uses a specification expressed as a postcondition, based on the inductive relation representing the language of a given regular expression. Where we use nondeterminism to handle the concatenation operator, Harper uses a continuation-passing parser for control flow. Since the continuations take the unparsed remainder of the string, they correspond almost directly to the *Parser* effect of the following section. Another main difference between our implementation and Harper's is in the way the non-termination of *match* is resolved. Harper uses the derivative operator to rewrite the expression in a standard form which ensures that the *match* function terminates. We use the derivative operator to implement a different matcher *dmatch* which is easily proved to be terminating, then show that *match*, which we have already proven partially correct, is refined by *dmatch*. The final major difference is that Harper uses manual verification of the program and our work is formally computer-verified. Although our development takes more work, the correctness proofs give more certainty than the informal arguments made by Harper. In general, choosing between informal reasoning and formal verification will always be a trade-off between speed and accuracy.

7   Parsing as effect

In the previous sections, we wrote parsers as nondeterministic functions. For more complicated classes of languages than regular expressions, explicitly passing around the string to be parsed becomes cumbersome quickly. The traditional solution is to switch from nondeterminism to stateful nondeterminism, where the state contains the unparsed portion of the string [Hut92]. The combination of nondeterminism and state can be represented by the *ListOfSuccesses* monad:

> *ListOfSuccesses* : *Set* → *Set*
> *ListOfSuccesses a* = *String* → *List* (*a* × *String*)

Since our development makes use of algebraic effects, we can introduce the effect of mutable state without having to change existing definitions. We introduce this using the *Parser* effect, which has one command *Symbol*. Calling *Symbol* will return the current symbol in the state (advancing the state by one) or fail if all symbols have been consumed.

> **data** *CParser* : *Set* **where**
>   *Symbol* : *CParser*
> *RParser* : *CParser* → *Set*
> *RParser Symbol* = *Char*
> *Parser* = *mkSig CParser RParser*
>
> *symbol* : {{ *iP* : *Parser* ∈ *es* }} → *Free es Char*
> *symbol* {{ *iP* }} = *Op iP Symbol Pure*

We could add more commands such as *EOF* for detecting the end of the input, but we do not need them in the current development. In the semantics we will define that parsing was successful if the input string has been completely consumed.

Note that *Parser* is not sufficient by itself to implement even simple parsers such as *dmatch*: we need to be able to choose between parsing the next character or returning a value for the empty string. This is why we usually combine *Parser* with nondeterminism and general recursion.

The denotational semantics of a parser in the *Free* monad take the form of a fold, handling each command in the *Parser* monad.

> *runParser* : *Free* (*Nondet* :: *Parser* :: *Nil*) *a* → *ListOfSuccesses a*
> *runParser* (*Pure x*) *Nil* = (*x*, *Nil*) :: *Nil*
> *runParser* (*Pure x*) (_ :: _) = *Nil*
> *runParser* (*Op* ∈Head *Fail k*) *xs* = *Nil*
> *runParser* (*Op* ∈Head *Choice k*) *xs* =
>   *runParser* (*k True*) *xs* ++ *runParser* (*k False*) *xs*
> *runParser* (*Op* (∈Tail ∈Head) *Symbol k*) *Nil* = *Nil*
> *runParser* (*Op* (∈Tail ∈Head) *Symbol k*) (*x* :: *xs*) = *runParser* (*k x*) *xs*

In this article, we are more interested in the predicate transformer semantics of *Parser*. Since the semantics of *Parser* refer to a state, the predicates depend on this state. We can incorporate a mutable state of type *s* in predicate transformer semantics by replacing the propositions in *Set* with predicates over the state in *s* → *Set*. We define the resulting type of stateful predicate transformers for an effect with signature *e* to be PT^S *s e*, as follows:

> **record** PT^S (*s* : *Set*) (*e* : *Sig*) : *Set* **where**
>   **constructor** *mkPTS*
>   **field**
>   *pt* : (*c* : *C e*) → (*R e c* → *s* → *Set*) → *s* → *Set*
>   *mono* : ∀ *c P P′* → (∀ *x t* → *P x t* → *P′ x t*) → *pt c P* ⊆ *pt c P′*

If we define PTs^S and *lookupPTS* analogously to *PTs* and *lookupPT*, we have fond a predicate transformer semantics that incorporates the current state:

> ⟦·⟧^{S} : (*pts* : PTs^S *s es*) → *Free es a* → (*a* → *s* → *Set*) → *s* → *Set*
> ⟦*Pure x*⟧$^S_{pts}$ *P* = *P x*
> ⟦*Op i c k*⟧$^S_{pts}$ *P* = *lookupPTS pts i c* λ *x* → ⟦*k x*⟧$^S_{pts}$ *P*

In this definition for $\llbracket \cdot \rrbracket^S$, we assume that all effects share access to one mutable variable of type *s*. We can allow for more variables by setting *s* to be a product type over the effects. With a suitable modification of the predicate transformers, we could set it up so that each effect can only modify its own associated variable. Thus, the previous definition is not limited in generality by writing it only for one variable.

To give the predicate transformer semantics of the *Parser* effect, we need to choose the meaning of failure, for the case where the next character is needed and all characters have already been consumed. Since we want all results returned by the parser to be correct, we use demonic choice and the *ptAll* predicate transformer as the semantics for *Nondet*. Using *ptAll*'s semantics for the *Fail* command gives the following semantics for the *Parser* effect.

> *ptParse* : PT^S *String Parser*
> *PTS.pt ptParse Symbol P Nil* $= \top$
> *PTS.pt ptParse Symbol P* (*x* :: *xs*) $= P\ x\ xs$

With the predicate transformer semantics of *Parser*, we can define the language accepted by a parser in the *Free* monad as a predicate over strings: a string *xs* is in the language of a parser *S* if the postcondition "all characters have been consumed" is satisfied.

> *empty?* : *List a* $\to$ *Set*
> *empty? Nil* $= \top$
> *empty?* (_ :: _) $= \bot$
>
> _$\in$[_] : *String* $\to$ *Free* (*Nondet* :: *Parser* :: *Nil*) *a* $\to$ *Set*
> *xs* $\in$ [*S*] $= \llbracket S \rrbracket^S_{ptAll\,::\,ptParse\,::\,Nil}$ (λ _ $\to$ *empty?*) *xs*

8 Parsing context-free languages

In Section 6, we developed and formally verified a parser for regular languages. The class of regular languages is small, and does not include most programming languages. A class of languages that is more expressive than the regular languages, while remaining tractable in parsing is that of the context-free language. The expressiveness of context-free languages is enough to cover most programming languages used in practice [AU77]. We will represent context-free languages in Agda by giving a grammar in the style of Brink, Holdermans, and Löh [BHL10], in a similar way as we represent a regular language using an element of the *Regex* type. Following their development, we parametrize our definitions over a collection of nonterminal symbols.

> **record** *GrammarSymbols* : *Set* **where**
> **field**
>   *Nonterm* : *Set*
> $\llbracket\_\rrbracket$: *Nonterm* $\to$ *Set*
>   _$\overset{?}{=}$_ : *Decidable* {*A* $=$ *Nonterm*} _$==$_

The elements of the type *Char* are the terminal symbols. The elements of the type *Nonterm* are the nonterminal symbols, representing the language constructs. As for *Char*, we also need to be

able to decide the equality of nonterminals. The (disjoint) union of *Char* and *Nonterm* gives all the symbols that we can use in defining the grammar.

$$Symbol \; = \; Either \; Char \; Nonterm$$
$$Symbols \; = \; List \; Symbol$$

For each nonterminal *A*, our goal is to parse a string into a value of type ⟦ *A* ⟧, based on a set of production rules. A production rule $A \rightarrow xs$ gives a way to expand the nonterminal *A* into a list of symbols *xs*, such that successfully matching each symbol of *xs* with parts of a string gives a match of the string with *A*. Since matching a nonterminal symbol *B* with a (part of a) string results in a value of type ⟦ *B* ⟧, a production rule for *A* is associated with a semantic function that takes all values arising from submatches and returns a value of type ⟦ *A* ⟧, as expressed by the following type:

$$⟦\_\|\_⟧ \; : \; Symbols \; \rightarrow \; Nonterm \; \rightarrow \; Set$$
$$⟦ \; Nil \quad\quad \| \; A \; ⟧ \; = \; ⟦ \; A \; ⟧$$
$$⟦ \; Inl \; x \;\; :: \; xs \; \| \; A \; ⟧ \; = \; ⟦ \; xs \; \| \; A \; ⟧$$
$$⟦ \; Inr \; B \; :: \; xs \; \| \; A \; ⟧ \; = \; ⟦ \; B \; ⟧ \; \rightarrow \; ⟦ \; xs \; \| \; A \; ⟧$$

Now we can define the type of production rules. A rule of the form $A \rightarrow BcD$ is represented as $prod \, A \, (Inr \, B \; :: \; Inl \, c \; :: \; Inr \, D \; :: \; Nil) \, f$ for some f.

record *Prod* : *Set* **where**
   constructor *prod*
   field
      lhs : *Nonterm*
      rhs : *Symbols*
      sem : ⟦ *rhs* ∥ *lhs* ⟧

We use the abbreviation *Prods* to represent a list of productions, and a grammar will consist of the list of all relevant productions.

    We want to show that a generally recursive function making use of the effects *Parser* and *Nondet* can parse any context-free grammar. To show this claim, we implement a function *fromProds* that constructs a parser for any context-free grammar given as a list of *Prod*s, then formally verify the correctness of *fromProds*. Our implementation mirrors the definition of the *generateParser* function by Brink, Holdermans, and Löh, differing in the naming and in the system that the parser is written in: our implementation uses the *Free* monad and algebraic effects, while Brink, Holdermans, and Löh use a monad *Parser* that is based on parser combinators.

    We start by defining two auxiliary types, used as abbreviations in our code.

$$FreeParser \; = \; Free \, (mkSig \; Nonterm \; ⟦\_⟧ \; :: \; Nondet \; :: \; Parser \; :: \; Nil)$$

record *ProdRHS* (*A* : *Nonterm*) : *Set* **where**
   constructor *prodrhs*
   field
      rhs : *Symbols*
      sem : ⟦ *rhs* ∥ *A* ⟧

    The core algorithm for parsing a context-free grammar consists of the following functions, calling each other in mutual recursion:

$$fromProds \quad : (A : Nonterm) \ \to \ FreeParser \ [\![A]\!]$$
$$filterLHS \quad : (A : Nonterm) \ \to \ Prods \ \to \ List \ (ProdRHS \ A)$$
$$fromProd \quad : ProdRHS \ A \ \to \ FreeParser \ [\![A]\!]$$
$$buildParser : (xs : Symbols) \ \to \ FreeParser \ ([\![xs \parallel A]\!] \ \to \ [\![A]\!])$$
$$exact \quad \quad : a \ \to \ Char \ \to \ FreeParser \ a$$

The main function is *fromProds*: given a nonterminal, it selects the productions with this non-terminal on the left hand side using *filterLHS*, and makes a nondeterministic choice between them.

$$filterLHS \ A \ Nil \ = \ Nil$$
$$filterLHS \ A \ (prod \ lhs \ rhs \ sem \ :: \ ps) \ \textbf{with} \ A \ \overset{?}{=} \ lhs$$
$$... \mid yes \ refl \ = \ prodrhs \ rhs \ sem \ :: \ filterLHS \ A \ ps$$
$$... \mid no \ \_ \quad = \ filterLHS \ A \ ps$$
$$fromProds \ A \ = \ choices \ (map \ fromProd \ (filterLHS \ A \ prods))$$

The *choices* operator takes a list of computations and nondeterministically chooses one of them to execute.

The function *fromProd* takes a single production and tries to parse the input string using this production. It then uses the semantic function of the production to give the resulting value.

$$fromProd \ (prodrhs \ rhs \ sem) \ = \ buildParser \ rhs \ \ggg \ \lambda f \ \to \ Pure \ (f \ sem)$$

The function *buildParser* iterates over the *Symbols*, calling *exact* for each literal character symbol, and making a recursive *call* to *fromProds* for each nonterminal symbol.

$$buildParser \ Nil \ = \ Pure \ id$$
$$buildParser \ (Inl \ x \ :: \ xs) \ = \ exact \ tt \ x \ \ggg \ \lambda \ \_ \ \to \ buildParser \ xs$$
$$buildParser \ (Inr \ B \ :: \ xs) \ = \ \text{do}$$
$$\quad x \leftarrow \ call \ B$$
$$\quad o \leftarrow \ buildParser \ xs$$
$$\quad Pure \ \lambda f \ \to \ o \ (f \ x)$$

Finally, *exact* uses the *symbol* command to check that the next character in the string is as expected, and *fail*s if this is not the case.

$$exact \ x \ t \ = \ symbol \ \ggg \ \lambda \ t' \ \to \ \text{if} \ t \ \overset{?}{=} \ t' \ \text{then} \ Pure \ x \ \text{else} \ fail$$

9 Partial correctness of the parser

Partial correctness of the parser is relatively simple to show, as soon as we have a specification. Since we want to prove that *fromProds* correctly parses any given context free grammar given as an element of *Prods*, the specification consists of a relation between many sets: the production rules, an input string, a nonterminal, the output of the parser, and the remaining unparsed string. Due to the many arguments, the notation is unfortunately somewhat unwieldy. To make it a bit easier to read, we define two relations in mutual recursion, one for all productions of a nonterminal, and for matching a string with a single production rule.

data _ ⊢ _ ∈ ⟦ _ ⟧ ⇒ _ , _ *prods* **where**
   Produce : *prod lhs rhs sem* ∈ *prods* →
                  prods ⊢ *xs* ∼ *rhs* ⇒ *f* , *ys* →
                  prods ⊢ *xs* ∈ ⟦ *lhs* ⟧ ⇒ *f sem* , *ys*
data _ ⊢ _ ∼ _ ⇒ _ , _ *prods* **where**
   Done :   *prods* ⊢ *xs* ∼ *Nil* ⇒ *id* , *xs*
   Next :   *prods* ⊢ *xs* ∼ *ps* ⇒ *o* , *ys* →
                  prods ⊢ (*x* :: *xs*) ∼ (*Inl x* :: *ps*) ⇒ *o* , *ys*
   Call :   *prods* ⊢ *xs* ∈ ⟦ *A* ⟧ ⇒ *o* , *ys* →
                  prods ⊢ *ys* ∼ *ps* ⇒ *f* , *zs* →
                  prods ⊢ *xs* ∼ (*Inr A* :: *ps*) ⇒ (λ *g* → *f* (*g o*)) , *zs*

With these relations, we can define the specification *parserSpec* to be equal to _ ⊢ _ ∈ ⟦ _ ⟧ ⇒ _ , _
(up to reordering some arguments), and show that *fromProds* refines this specification. To state
that the refinement relation holds, we first need to determine the semantics of the effects. We
choose *ptAll* as the semantics of nondeterminism, since we want to ensure all output of the parser
is correct.

$$pts\ prods = ptRec\ (parserSpec\ prods) :: ptAll :: ptParse :: Nil$$

$$⟦S⟧_{\text{fromProd}}\ prods = ⟦S⟧^S_{pts\ prods}$$

$$partialCorrectness : (prods : Prods)\ (A : Nonterm) →$$
$$⟦⊤, (parserSpec\ prods\ A)⟧_{\text{spec}} ⊑ ⟦fromProds\ prods\ A⟧_{\text{fromProd}}\ prods$$

Let us fix the production rules *prods*. How do we prove the partial correctness of a parser for
prods? Since the structure of *fromProds* is of a nondeterministic choice between productions to
be parsed, and we want to show that all alternatives for a choice result in success, we will first
give a lemma expressing the correctness of each alternative. Correctness in this case is expressed
by the semantics of a single production rule, i.e. the _ ⊢ _ ∼ _ ⇒ _ , _ relation. Thus, we want
to prove the following lemma:

$$parseStep : ∀ A\ xs\ P\ str →$$
$$(∀ o\ str' → prods ⊢ str ∼ xs ⇒ o , str' → P\ o\ str') →$$
$$⟦buildParser\ prods\ xs⟧_{\text{fromProd}}\ prods\ P\ str$$

The lemma can be proved by reproducing the case distinctions used to define *buildParser*; there is
no complication apart from having to use the *wpToBind* lemma to deal with the _ ⟫= _ operator
in a few places.

$$parseStep\ A\ Nil\ P\ t\ H = H\ id\ t\ Done$$
$$parseStep\ A\ (Inl\ x :: xs)\ P\ Nil\ H = tt$$
$$parseStep\ A\ (Inl\ x :: xs)\ P\ (x' :: t)\ H\ \textbf{with}\ x \stackrel{?}{=} x'$$
$$... \mid yes\ refl = parseStep\ A\ xs\ P\ t\ λ\ o\ t'\ H' → H\ o\ t'\ (Next\ H')$$
$$... \mid no\ ¬p = tt$$
$$parseStep\ A\ (Inr\ B :: xs)\ P\ t\ H\ o\ t'\ Ho =$$
$$\quad wpToBind\ (buildParser\ prods\ xs)\ \_\ \_$$
$$\qquad (parseStep\ A\ xs\ \_\ t'\ λ\ o'\ str'\ Ho' → H\ \_\ \_\ (Call\ Ho\ Ho'))$$

To combine the *parseStep* for each of the productions that the nondeterministic choice is made between, it is tempting to define another lemma *filterStep* by induction on the list of productions. But we must be careful that the productions that are used in the *parseStep* are the full list *prods*, not the sublist *prods'* used in the induction step. Additionally, we must also make sure that *prods'* is indeed a sublist, since using an incorrect production rule in the *parseStep* will result in an invalid result. Thus, we parametrise *filterStep* by a list *prods'* and a proof that it is a sublist of *prods*. Again, the proof uses the same distinction as *fromProds* does, and uses the *wpToBind* lemma to deal with the _>>=_ operator.

$$filterStep : \forall \, prods' \, A \to (p \in prods' \to p \in prods) \to$$
$$\llbracket \top, (parserSpec \; prods \, A) \rrbracket_{\text{spec}} \sqsubseteq \llbracket choices \, (map \, (fromProd \; prods) \, (filterLHS \; prods \, A \; prods')) \rrbracket_{\text{fromProd}} \; prods$$
$$filterStep \; Nil \, A \; subset \, P \; xs \, H \; = \; tt$$
$$filterStep \; (prod \; lhs \; rhs \; sem \; :: \; prods') \, A \; subset \, P \; xs \, H \; \textbf{with} \, A \overset{?}{=} lhs$$
$$filterStep \; (prod \; .A \; rhs \; sem \; :: \; prods') \, A \; subset \, P \; xs \, (\_ , H) \; | \; yes \; refl$$
$$\quad = \; wpToBind \; (buildParser \; prods \; rhs) \; \_ \_$$
$$\quad (parseStep \, A \; rhs \; \_ \; xs \; \lambda \; o \; t' \; H' \to H \; \_ \_ \, (Produce \, (subset \in \text{Head}) \, H'))$$
$$\quad , filterStep \; prods' \, A \; (subset \circ \in \text{Tail}) \, P \; xs \, (\_ , H)$$
$$... \; | \; no \; \neg p \; = \; filterStep \; prods' \, A \; (subset \circ \in \text{Tail}) \, P \; xs \, H$$

With these lemmas, *partialCorrectness* just consists of applying *filterStep* to the subset of *prods* consisting of *prods* itself.

10   Termination of the parser

To show termination we need a somewhat more subtle argument: since we are able to call the same nonterminal repeatedly, termination cannot be shown simply by inspecting each alternative in the definition. Consider the grammar given by $E \to aE; E \to b$, where we see that the string that matches E in the recursive case is shorter than the original string, but the definition itself can be expanded to unbounded length. By taking into account the current state, i.e. the string to be parsed, in the variant, we can show that a decreasing string length leads to termination.

But not all grammars feature this decreasing string length in the recursive case, with the most pathological case being those of the form $E \to E$. The issues do not only occur in edge cases: the grammar $E \to E + E; E \to 1$ representing very simple expressions will already result in non-termination for *fromProds* as it will go in recursion on the first non-terminal without advancing the input string. Since the position in the string and current nonterminal together fully determine the state of *fromParsers*, it will not terminate. We need to ensure that the grammars passed to the parser do not allow for such loops.

Intuitively, the condition on the grammars should be that they are not left-recursive, since in that case, the parser should always advance its position in the string before it encounters the same nonterminal. This means that the number of recursive calls to *fromProds* is bounded by the length of the string times the number of different nonterminals occurring in the production rules. The type we will use to describe the predicate "there is no left recursion" is constructively somewhat stronger: we define a left-recursion chain from A to B to be a sequence of nonterminals $A, \ldots, A_i, A_{i+1}, \ldots, B$, such that for each adjacent pair A_i, A_{i+1} in the chain, there is a production of the form $A_{i+1} \to B_1 B_2 \ldots B_n A_i \ldots$, where $B_1 \ldots B_n$ are all nonterminals. In other words, we can advance the parser to A starting in B without consuming a character. Disallowing (unbounded)

left recursion is not a limitation for our parsers: Brink, Holdermans, and Löh [BHL10] have
shown that the left-corner transform can transform left-recursive grammars into an equivalent
grammar without left recursion. Moreover, they have implemented this transform, including
formal verification, in Agda. In this work, we assume that the left-corner transform has already
been applied if needed, so that there is an upper bound on the length of left-recursive chains in
the grammar.

We formalize one link of this left-recursive chain in the type *LRec*, while a list of such links
forms the *Chain* data type.

> **record** *LRec* (*prods* : *Prods*) (*A B* : *Nonterm*) : *Set* **where**
>   **field**
>   *rec* : *prod A* (*map Inr xs* ++ (*Inr B* :: *ys*)) *sem* ∈ *prods*

(We leave *xs*, *ys* and *sem* as implicit fields of *LRec*, since they are fixed by the type of *rec*.)

> **data** *Chain* (*prods* : *Prods*) : *Nonterm* → *Nonterm* → *Set* **where**
>   *Nil* : *Chain prods A A*
>   _::_ : *LRec prods B A* → *Chain prods A C* → *Chain prods B C*

Now we say that a set of productions has no left recursion if all such chains have an upper bound
on their length.

> *chainLength* : *Chain prods A B* → ℕ
> *chainLength Nil* = *0*
> *chainLength* (*c* :: *cs*) = *Succ* (*chainLength cs*)
>
> *leftRecBound* : *Prods* → ℕ → *Set*
> *leftRecBound prods n* = (*cs* : *Chain prods A B*) → *chainLength cs* < *n*

If we have this bound on left recursion, we are able to prove termination, since each call to
fromProds will be made either after we have consumed an extra character, or it is a left-recursive
step, of which there is an upper bound on the sequence.

This informal proof fits better with a different notion of termination than in the petrol-
driven semantics. The petrol-driven semantics are based on a syntactic argument: we know a
computation terminates because expanding the call tree will eventually result in no more *call*s.
Here, we want to capture the notion that a recursive definition terminates if all recursive calls
are made to a smaller argument, according to a well-founded relation.

Definition 2 ([Acz77]) In intuitionistic type theory, we say that a relation _≺_ on a type *a* is
well-founded if all elements *x* : *a* are accessible, which is defined by (well-founded) recursion to
be the case if all elements in the downset of *x* are accessible.

> **data** *Acc* (_≺_ : *a* → *a* → *Set*) : *a* → *Set* **where**
>   *acc* : (∀ *y* → *y* ≺ *x* → *Acc* _≺_ *y*) → *Acc* _≺_ *x*

To see that this is equivalent to the definition of well-foundedness in set theory, recall that a
relation _≺_ on a set *a* is well-founded if and only if there is a monotone function from *a*
to a well-founded order. Since all inductive data types are well-founded, and the termination

checker ensures that the argument to *acc* is a monotone function, there is a function from $x : a$ to $Acc \_\prec\_ x$ if and only if $\_\prec\_$ is a well-founded relation in the set-theoretic sense.

The condition that all calls are made to a smaller argument is related to the notion of a loop variant in imperative languages. While an invariant is a predicate that is true at the start and end of each looping step, the variant is a relation that holds between successive looping steps.

Definition 3 Given a recursive definition $f : I \overset{es}{\hookrightarrow} O$, a relation $\_\prec\_$ on C is a recursive variant if for each argument c, and each recursive call made to c' in the evaluation of $f\ c$, we have $c' \prec c$. Formally:

$$
\begin{aligned}
&variant' : (pts : \mathrm{PTs}^S\ s\ (mkSig\ C\ R :: es))\ (f : C \overset{es}{\hookrightarrow} R) \\
&\quad (\_\prec\_ : (C \times s) \to (C \times s) \to Set) \\
&\quad (c : C)\ (t : s)\ (S : Free\ (mkSig\ C\ R :: es)\ a) \to s \to Set \\
&variant'\ pts\ f\ \_\prec\_\ c\ t\ (Pure\ x)\ t' = \top \\
&variant'\ pts\ f\ \_\prec\_\ c\ t\ (Op \in \mathrm{Head}\ c'\ k)\ t' \\
&\quad = ((c'\ ,\ t') \prec (c\ ,\ t)) \times lookupPTS\ pts \in \mathrm{Head}\ c' \\
&\qquad (\lambda\ x \to variant'\ pts\ f\ \_\prec\_\ c\ t\ (k\ x))\ t' \\
&variant'\ pts\ f\ \_\prec\_\ c\ t\ (Op\ (\in \mathrm{Tail}\ i)\ c'\ k)\ t' \\
&\quad = lookupPTS\ pts\ (\in \mathrm{Tail}\ i)\ c'\ (\lambda\ x \to variant'\ pts\ f\ \_\prec\_\ c\ t\ (k\ x))\ t' \\[1ex]
&variant : (pts : \mathrm{PTs}^S\ s\ (mkSig\ C\ R :: es))\ (f : C \overset{es}{\hookrightarrow} R) \to \\
&\quad (\_\prec\_ : (C \times s) \to (C \times s) \to Set) \to Set \\
&variant\ pts\ f\ \_\prec\_ = \forall\ c\ t \to variant'\ pts\ f\ \_\prec\_\ c\ t\ (f\ c)\ t
\end{aligned}
$$

Note that *variant* depends on the semantics *pts* we give to the recursive function f. We cannot derive the semantics in *variant* from the structure of f as we do for the petrol-driven semantics, since we do not yet know whether f terminates. Using *variant*, we can define another termination condition on f: there is a well-founded variant for f.

$$
\begin{aligned}
&\textbf{record}\ \textit{Termination}\ (pts : \mathrm{PTs}^S\ s\ (mkSig\ C\ R :: es))\ (f : C \overset{es}{\hookrightarrow} R) : Set\ \textbf{where} \\
&\quad \textbf{field} \\
&\qquad \_\prec\_ : (C \times s) \to (C \times s) \to Set \\
&\qquad w-f : \forall\ c\ t \to Acc\ \_\prec\_\ (c\ ,\ t) \\
&\qquad var : variant\ pts\ f\ \_\prec\_
\end{aligned}
$$

A generally recursive function that terminates in the petrol-driven semantics also has a well-founded variant, given by the well-order $\_<\_$ on the amount of fuel consumed by each call. The converse also holds: if we have a descending chain of calls *cs* after calling f with argument c, we can use induction on the type $Acc\ \_\prec\_ c$ to bound the length of *cs*. This bound gives the amount of fuel consumed by evaluating a call to f on c.

In our case, the relation *RecOrder* will work as a recursive variant for *fromProds*:

$$
\begin{aligned}
&\textbf{data}\ \textit{RecOrder}\ (prods : Prods) : (x\ y : Nonterm \times String) \to Set\ \textbf{where} \\
&\quad Left : length\ str < length\ str' \to \\
&\qquad RecOrder\ prods\ (A\ ,\ str)\ (B\ ,\ str') \\
&\quad Right : length\ str \leq length\ str' \to \\
&\qquad LRec\ prods\ A\ B \to RecOrder\ prods\ (A\ ,\ str)\ (B\ ,\ str')
\end{aligned}
$$

With the definition of *RecOrder*, we can complete the correctness proof of *fromProds*, by giving
an element of the corresponding *Termination* type. We assume that the length of recursion is
bounded by *bound* : ℕ.

> *fromProdsTerminates* : ∀ *prods bound* → *leftRecBound prods bound* →
>   *Termination* (*pts prods*) (*fromProds prods*)
> *Termination.__≺__* (*fromProdsTerminates prods bound H*) = *RecOrder prods*

To show that the relation *RecOrder* is well-founded, we need to show that there is no infinite
descending chain starting from some nonterminal *A* and string *str*. The proof is based on iteration
on two natural numbers *n* and *k*, which form an upper bound on the number of allowed left-
recursive calls in sequence and unconsumed characters in the string respectively. Note that the
number *bound* is an upper bound for *n* and the length of the input string is an upper bound
for *k*. Since each nonterminal in the production will decrease *n* and each terminal will decrease
k, we eventually reach the base case *0* for either. If *n* is zero, we have made more than *bound*
left-recursive calls, contradicting the assumption that we have bounded left recursion. If *k* is
zero, we have consumed more than *length str* characters of *str*, also a contradiction.

> *Termination.w − f* (*fromProdsTerminates prods bound H*) *A str*
>   = *acc* (*go A str* (*length str*) ≤-refl *bound Nil* ≤-refl)
>   **where**
>   *go* : ∀ *A str* →
>   (*k* : ℕ) → *length str* ≤ *k* →
>   (*n* : ℕ) (*cs* : *Chain prods A B*) → *bound* ≤ *chainLength cs* + *n* →
>   ∀ *y* → *RecOrder prods y* (*A* , *str*) → *Acc* (*RecOrder prods*) *y*

Our next goal is that *RecOrder* is a variant for *fromProds*, as abbreviated by the *prodsVariant*
type. We cannot follow the definitions of *fromProds* as closely as we did for the partial correctness
proof; instead we need a complicated case distinction to keep track of the left-recursive chain
we have followed in the proof. For this reason, we split the *parseStep* apart into two lemmas
parseStepAdv and *parseStepRec*, both showing that *buildParser* maintains the variant. We also
use a *filterStep* lemma that calls the correct *parseStep* for each production in the nondeterministic
choice.

> *prodsVariant* = *variant'* (*pts prods*) (*fromProds prods*) (*RecOrder prods*)
>
> *parseStepAdv* : ∀ *A xs str str'* → *length str'* < *length str* →
>   *prodsVariant A str* (*buildParser xs*) *str'*
> *parseStepRec* : ∀ *A xs str str'* → *length str'* ≤ *length str* →
>   ∀ *ys* → *prod A* (*map Inr ys* ++ *xs*) *sem* ∈ *prods* →
>   *prodsVariant A str* (*buildParser xs*) *str'*
> *filterStep* : ∀ *prods'* → (*x* ∈ *prods'* → *x* ∈ *prods*) →
>   ∀ *A str str'* → *length str'* ≤ *length str* →
>   *prodsVariant A str*
>   (*foldr* (*choice*) (*fail*) (*map fromProd* (*filterLHS A prods'*)))
>   *str'*

In the *parseStepAdv*, we deal with the situation that the parser has already consumed at least
one character since it was called. This means we can repeatedly use the *Left* constructor of
RecOrder to show the variant holds.

In the *parseStepRec*, we deal with the situation that the parser has only encountered nonterminals in the current production. This means that we can use the *Right* constructor of *RecOrder* to show the variant holds until we consume a character, after which we call *parseStepAdv* to finish the proof.

The lemma *filterStep* shows that the variant holds on all subsets of the production rules, analogously to the *filterStep* of the partial correctness proof. It calls *parseStepRec* since the parser only starts consuming characters after it selects a production rule.

> *filterStep Nil A str str′ lt subset = tt*
> *filterStep (prod lhs rhs sem ∷ prods′) subset A str str′ lt* **with** $A \overset{?}{=} lhs$
> *... | yes refl*
>   = *variant − fmap (pts prods) (fromProds prods) (buildParser rhs)*
>   *(parseStepRec A rhs str str′ lt Nil (subset ∈ Head))*
>   *, filterStep prods′ (subset ∘ ∈ Tail) A str str′ lt*
> *... | no ¬p = filterStep prods′ (subset ∘ ∈ Tail) A str str′ lt*

As for partial correctness, we obtain the proof of termination by applying *filterStep* to the subset of *prods* consisting of *prods* itself.

11   Discussion

11.1   Related work

In this paper, we have described a representation of parsers and shown how to perform verification of parsers in this representation. We will discuss how our work relates to other parser verifications. Our sections on regular expressions have a similar structure to a Functional Pearl by Harper [Har99]. The main difference is that our work is based on formal verification using Agda, while Harper uses manual and informal reasoning. The sections on context-free grammars could be compared to work by Danielsson [Dan10] and Firsov [Fir16]. Here the difference, apart from a different parsing algorithm, can be found in how (non)termination is dealt with. We opt for a strong separation of syntax and semantics, using the *Rec* effect to give the syntax of programs regardless of termination, later proving the semantic property of termination. In contrast, Danielsson; Firsov deal with termination syntactically, either by incorporating delay and force operators in the grammar, or explicitly passing around a proof of termination in the definition of the parser.

A different representation of languages used in verification is the coinductive trie [Abe16]. The approach of Abel is in the opposite direction to ours: in order to verify constructions on automata, the language they accept is mapped to a trie, then this trie is compared to the trie that we get by applying the corresponding constructions on tries. Similarly, Abel, Adelsberger, and Setzer [AAS17] use a coinductive type to represent effectful programs with arbitrarily large input. These two coinductive constructions carry proofs of productivity, in the form of sized types, in their definitions, again mixing syntax and semantics.

11.2   Open issues

In the process of this verification, we have solved some open issues in the area of predicate transformer semantics and leave others open. Swierstra and Baanen [SB19] mention two avenues

of further work that our work makes advances on: the semantics for combinations of effects and the verification of non-trivial programs using algebraic effects. Still, we chose to verify parsers with applying predicate transformers to them in the back of our mind, so the goal of verifying a practical program remains a step further.

We have described how coproducts allow for combinations of effect syntax and semantics, but not how these combinations interact with effect handlers. The interaction between different effects means applying handlers in a different order can result in different semantics. We assign predicate transformer semantics to a combination of effects all at once, specifying their interaction explicitly. Can we assign semantics to effects such that they interact in a similar way as handlers do?

Another issue that remains is dealing with other representations of the free monad. The *Free* datatype could be replaced with more efficient versions to run practical computations [KSS13; KI15]. We expect that predicate transformer semantics, although arising from a fold on the *Free* monad, will generalize without problems to these more advanced representations.

11.3   Conclusions

In conclusion, the two distinguishing features of our work are formality and modularity. We could introduce the combination of effects, petrol-driven termination, semantics for state and variant-based termination without impacting existing definitions. We strictly separate the syntax and semantics of the programs, and partial correctness from termination. This results in verification proofs that do not need to carry around many goals, allowing most of them to consist of unfolding the definition and filling in the obvious terms.

We should also note that the engineering effort expected by Swierstra and Baanen has not been needed for our paper. The optimist can conclude that the elegance of our framework caused it to prevent the feared level of complication; the pessimist can conclude that the real hard work will be required as soon as we encounter a real-world application.

References

[AAG03]   Michael Abbott, Thorsten Altenkirch, and Neil Ghani. "Categories of Containers". In: In Proceedings of Foundations of Software Science and Computation Structures. 2003.

[AAS17]   Andreas Abel, Stephan Adelsberger, and Anton Setzer. "Interactive programming in Agda – Objects and graphical user interfaces". In: Journal of Functional Programming 27 (Feb. 2017). DOI: `10.1017/S0956796816000319`.

[Abe16]   Andreas Abel. "Equational Reasoning about Formal Languages in Coalgebraic Style". preprint available at `http://www.cse.chalmers.se/~abela/jlamp17.pdf`. Dec. 2016.

[Acz77]   Peter Aczel. "An Introduction to Inductive Definitions". In: Handbook of Mathematical Logic. Ed. by Jon Barwise. Vol. 90. Studies in Logic and the Foundations of Mathematics. Elsevier, 1977, pp. 739–782. DOI: `https://doi.org/10.1016/S0049-237X(08)71120-0`.

[AU77]   Alfred Aho and Jeffrey D. Ullman. Principles of compiler design. Reading, Mass: Addison-Wesley Pub. Co, 1977. ISBN: 0201000229.

[BHL10]   Kasper Brink, Stefan Holdermans, and Andres Löh. "Dependently Typed Grammars". In: June 2010, pp. 58–79. DOI: `10.1007/978-3-642-13321-3_6`.

[BP15]   Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: Journal of Logical and Algebraic Methods in Programming 84.1 (2015). Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011, pp. 108–123. ISSN: 2352-2208. DOI: `https://doi.org/10.1016/j.jlamp.2014.02.001`.

[Brz64]   Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: J. ACM 11.4 (Oct. 1964), pp. 481–494. ISSN: 0004-5411. DOI: `10.1145/321239.321249`. URL: `http://doi.acm.org/10.1145/321239.321249`.

[Dan10]   Nils Anders Danielsson. "Total Parser Combinators". In: SIGPLAN Not. 45.9 (Sept. 2010), pp. 285–296. ISSN: 0362-1340. DOI: `10.1145/1932681.1863585`. URL: `http://doi.acm.org/10.1145/1932681.1863585`.

[Fir16]   Denis Firsov. "Certification of Context-Free Grammar Algorithms". PhD thesis. Institute of Cybernetics at Tallinn University of Technology, 2016.

[Har99]   Robert Harper. "Proof-directed debugging". In: Journal of Functional Programming 9.4 (1999), pp. 463–469. DOI: `10.1017/S0956796899003378`.

[Hut92]   Graham Hutton. "Higher-order functions for parsing". In: Journal of Functional Programming 2.3 (1992), pp. 323–343. DOI: `10.1017/S0956796800000411`.

[KI15]   Oleg Kiselyov and Hiromi Ishii. "Freer Monads, More Extensible Effects". In: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell. Haskell '15. Vancouver, BC, Canada: ACM, 2015, pp. 94–105. ISBN: 978-1-4503-3808-0. DOI: `10.1145/2804302.2804319`. URL: `http://doi.acm.org/10.1145/2804302.2804319`.

[KSS13]   Oleg Kiselyov, Amr Sabry, and Cameron Swords. "Extensible Effects: An Alternative to Monad Transformers". In: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell. Haskell '13. Boston, Massachusetts, USA: ACM, 2013, pp. 59–70. ISBN: 978-1-4503-2383-3. DOI: `10.1145/2503778.2503791`. URL: `http://doi.acm.org/10.1145/2503778.2503791`.

[McB15]   Conor McBride. "Turing-Completeness Totally Free". In: Mathematics of Program Construction. Ed. by Ralf Hinze and Janis Voigtländer. Cham: Springer International Publishing, 2015, pp. 257–275. ISBN: 978-3-319-19797-5.

[Nor07]   Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Chalmers University of Technology, 2007.

[PP03]   Gordon Plotkin and John Power. "Algebraic Operations and Generic Effects". In: Applied Categorical Structures 11.1 (Feb. 2003), pp. 69–94. ISSN: 1572-9095. DOI: `10.1023/A:1023064908962`. URL: `https://doi.org/10.1023/A:1023064908962`.

[SB19]   Wouter Swierstra and Tim Baanen. "A predicate transformer semantics for effects (Functional Pearl)". In: Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming. ICFP '19. 2019. DOI: `10.1145/3341707`.

[SD96]   S. Doaitse Swierstra and Luc Duponcheel. "Deterministic, Error-Correcting Combinator Parsers". In: Advanced Functional Programming. Springer-Verlag, 1996, pp. 184–207.

[Wad85]   Philip Wadler. "How to Replace Failure by a List of Successes". In: Proc. Of a Conference on Functional Programming Languages and Computer Architecture. Nancy, France: Springer-Verlag New York, Inc., 1985, pp. 113–128. ISBN: 3-387-15975-4. URL: `http://dl.acm.org/citation.cfm?id=5280.5288`.

[WSH14]   Nicolas Wu, Tom Schrijvers, and Ralf Hinze. "Effect Handlers in Scope". In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. Haskell '14. Gothenburg, Sweden: ACM, 2014, pp. 1–12. ISBN: 978-1-4503-3041-1. DOI: `10.1145/2633357.2633358`.