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 are commonly used when reasoning about pure functional programs, are less suitable when verifying effectful programs.

In this paper, we explore a different 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, we can define another implementation using Brzozowski derivatives (Section 6), introducing an extra effect to our combinations and its handler in the process.
- Finally, we show that the derivative-based implementation terminates and refines the original parser (Section 7).

The structure of our article is similar to 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

To appear in EPTCS.

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.

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 $P : a \to Set$ to a predicate on the result of the free monad to a predicate of the entire computation of type $Free\ (eff\ C\ R)\ a \to Set$.

```
⟦·⟧. : ((c : C) → (R c → Set) → Set) →
  Free (mkSig C R) a → (a → Set) → Set
⟦Pure x⟧alg P = P x
⟦Op c k⟧alg P = alg c λ x → ⟦k x⟧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 $⟦·⟧_{alg} : Free\ (mkSig\ C\ R)\ a \to (a \to Set) \to Set$. By having the command as first argument to alg, we allow R to depend on C. Moreover, $⟦·⟧_{alg}$ 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) → (RNondet c → Set) → Set
ptAll Fail P    = ⊤
ptAll Choice P = P True ∧ P False

⟦·⟧all : Free Nondet a → (a → Set) → Set
⟦S⟧all = ⟦·⟧. 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 → Set
```

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

$$[\![\cdot,\cdot]\!]_{\text{spec}} : \mathit{Spec}\ a \to (a \to \mathit{Set}) \to \mathit{Set}$$
$$[\![\mathit{pre},\mathit{post}]\!]_{\text{spec}}\ P = \mathit{pre} \wedge (\forall\ o \to \mathit{post}\ o \to P\ o)$$

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:

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

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)
  OrRight     : Match r xs x → Match (l | r) xs (Inr x)
  Concat      : Match l ys y → Match r zs z →
                Match (l · r) (ys ++ zs) (y , z)
  StarNil     : Match (r ⋆) Nil Nil
  StarConcat  : Match (r · (r ⋆)) xs (y , ys) → Match (r ⋆) xs (y :: ys)
```

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:

```
allSplits : (xs : List a) → Free Nondet (List a × List a)
allSplits Nil = Pure (Nil , Nil)
allSplits (x :: xs) = choice
  (Pure (Nil , (x :: xs)))
  (allSplits xs >>= λ {(ys , zs) → Pure ((x :: ys) , zs)})
```

```
match : (r : Regex) (xs : String) → Free Nondet (Tree r)
match Empty       xs          = fail
match Epsilon     Nil         = Pure tt
match Epsilon     (_ :: _)    = fail
match (Singleton c) Nil       = fail
match (Singleton c) (x :: Nil)  with c ≟ x
match (Singleton c) (.c :: Nil) | yes refl = Pure c
match (Singleton c) (x :: Nil)  | no ¬p    = fail
match (Singleton c) (_ :: _ :: _) = fail
match (l ∣ r)     xs          = choice (Inl ⟨$⟩ match l xs) (Inr ⟨$⟩ match r xs)
match (l · r)     xs          = do (ys , zs) ← allSplits xs
                                   y ← match l ys
                                   z ← match r zs
                                   Pure (y , z)
match (r ⋆) xs                = fail
```

Figure 1: The definition of the *match* function

Finally, we cannot yet handle the case for the Kleene star. We could attempt to mimic 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:

```
hasNo* : Regex → Set
```

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

```
pre : (r : Regex) (xs : String) → Set
pre r xs = hasNo* r
post : (r : Regex) (xs : String) → Tree r → Set
post = Match
```

The main correctness result can now be formulated as follows:

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

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 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:

```
consequence : ∀ pt (mx : Free es a) (f : a → Free es b) →
  ⟦mx⟧pt (λ x → ⟦f x⟧pt P) ≡ ⟦mx >>= f⟧pt P
```

Substituting along this equality gives us the lemmas we need to deal with the _>>=_ operator:

```
wpToBind : (mx : Free es a) (f : a → Free es b) →
  ⟦mx⟧pt (λ x → ⟦f x⟧pt P) → ⟦mx >>= f⟧pt P
wpFromBind : (mx : Free es a) (f : a → Free es b) →
  ⟦mx >>= f⟧pt P → ⟦mx⟧pt (λ x → ⟦f x⟧pt P)
```

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.

```
allSplitsPost : String → String × String → Set
allSplitsPost xs (ys , zs) = xs ≡ ys ++ zs
allSplitsSound : ∀ xs → ⟦⊤, (allSplitsPost xs)⟧spec ⊑ ⟦allSplits xs⟧all
```

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 incomplete, since it fails to handle regular expressions that use the Kleene star. The fundamental issue is that the Kleene star allows for arbitrarily many matches in certain cases, that in turn, leads to problems with Agda's termination checker. For example, matching *Epsilon* ⋆ with the empty string `""` may unfold the Kleene star infinitely often without ever terminating. As a result, we cannot implement *match* for the Kleene star using recursion directly.

Instead, we will deal with this (possibly unbounded) recursion by introducing a new effect. We will represent a recursively defined (dependent) function of type $(i : I) \to O\ i$ as an element of the type $(i : I) \to$ *Free* (*Rec I O*) $(O\ i)$. Here *Rec I O* is a synonym of the the signature type we saw previously [McB15]:

```
Rec : (I : Set) (O : I → Set) → Sig
Rec I O = mkSig I O
```

Intuitively, you may want to think of values of type $(i : I) \to$ *Free* (*Rec I O*) $(O\ i)$ as computing a (finite) call graph for every input $i : I$. Instead of recursing directly, the 'effects' that this signature support require an input $i : I$—corresponding to the argument of the recursive call; the continuation abstracts over a value of type $O\ i$, corresponding to the result of a recursive call. Note that the functions defined in this style are not recursive; instead we will need to write handlers to unfold the function definition or prove termination separately.

We cannot, however, define a *match* function of the form *Free* (*Rec* _ _) directly, as our previous definition also used non-determinism. To account for both non-determinism and unbounded recursion, we need a way to combine effects. Fortunately, free monads are known to be

closed under coproducts; there is a substantial body of work that exploits this to (syntactically) compose separate effects [WSH14; Swi08].

Rather than restrict ourselves to the binary composition using coproducts, we modify the *Free* monad to take a list of signatures as its argument, taking the coproduct of the elements of the list as its signature functor. The *Pure* constructor remains as unchanged; while the *Op* constructor additionally takes an index into the list to specify the effect that is invoked.

```
data Free (es : List Sig) (a : Set) : Set where
  Pure : a → Free es a
  Op : (i : e ∈ es) (c : C e) (k : R e c → Free es a) → 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. We choose to use the same names and (almost) the same syntax for this new definition of *Free*, since all the definitions that we have seen previously can be readily adapted to work with this data type instead.

Most of this bookkeeping involved with different effects can be inferred using Agda's instance arguments. Instance arguments, marked using the double curly braces {{ }}, are automatically filled in by Agda, provided a unique value of the required type can be found. For example, we can define the generic effects that we saw previously as follows:

```
fail : {{ iND : Nondet ∈ es }} → Free es a
fail {{ iND }} = Op iND Fail λ ()
choice : {{ iND : Nondet ∈ es }} → Free es a → Free es a → Free es a
choice {{ iND }} S1 S2 = Op iND Choice λ b → if b then S1 else S2
call : {{ iRec : Rec I O ∈ es }} → (i : I) → Free es (O i)
call {{ iRec }} i = Op iRec i Pure
```

These now operate over any free monad with effects given by *es*, provided we can show that the list *es* contains the *NonDet* and *Rec* effects respectively. For convenience of notation, we introduce the $\_ \overset{es}{\leftarrow} \_$ notation for general recursion, i.e. Kleisli arrows into *Free* (*Rec* _ _ :: *es*).

```
_ ←[es]– _ : (C : Set) (es : List Sig) (R : C → Set) → Set
C ←[es]– R = (c : C) → 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 signature, the semantics for a combination of effects can be given by a list of such semantics.

```
record PT (e : Sig) : Set where
  constructor mkPT
  field
    pt   : (c : C e) → (R e c → Set) → Set
    mono : ∀ c P P' → P ⊆ P' → pt c P → pt c P'
data PTs : List Sig → Set where
  Nil  : PTs Nil
  _::_ : PT e → PTs es → PTs (e :: es)
```

The record type *PT* not only contains a predicate transformer *pt*, but also a proof that this predicate transformer is monotone. Several lemmas throughout this paper, such as the terminates-fmap lemma below, rely on the monotonicity of the underlying predicate transformers.

Given a such a list of predicate transformers, defining the semantics of an effectful program is a straightforward generalization of the previously defined semantics. 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 ∈ es) → (c : C) → (R c → Set) → Set
lookupPT (pt :: pts) (∈Head)   = PT.pt pt
lookupPT (pt :: pts) (∈Tail i) = lookupPT pts i
```

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

```
⟦·⟧ : (pts : PTs es) → Free es a → (a → Set) → Set
⟦Pure x⟧pts  P = P x
⟦Op i c k⟧pts P = lookupPT pts i c λ x → ⟦k x⟧pts P
```

The effects that we will use for our *match* function consist of a combination of nondeterminism and general recursion. Although we can reuse the *ptAll* semantics of nondeterminism, we have not yet given the semantics for recursion. However, it is not as easy to give a predicate transformer semantics for recursion in general, since the intended semantics of a recursive call depend on the function that is being defined. Instead, to give semantics to a recursive function, we assume that we have been provided with a relation of the type $(i : I) \to O\ i \to Set$, reminiscent of a loop invariant in an imperative program. The semantics then establishes whether or not the recursive function adheres to this invariant or not:

```
ptRec : ((i : I) → O i → Set) → PT (Rec I O)
PT.pt      (ptRec R) i P                = ∀ o → R i o → P o
PT.mono (ptRec R) c P P′ imp asm o h = imp _ (asm _ h)
```

As we shall see shortly, when revisiting the *match* function, the *Match* relation defined previously will fulfill the role of this 'invariant.'

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 ∈ es }} → List a → Free es (List a × List a)
match : {{ iND : Nondet ∈ es }} → Regex × String ↪[es] Tree ∘ 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 does have a meaningful branch for the Kleene star case:

```
match ((r ⋆) , Nil)  =  Pure Nil
match ((r ⋆) , xs@ (_ :: _))  =  do
  (y , ys)  ←  call ((r · (r ⋆)) , xs)
  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)  →  Tree (Pair.fst r,xs)  →  Set
matchSpec (r , xs) ms  =  Match r xs ms
⟦·⟧match : Free (Rec (Pair Regex String) (Tree ∘ Pair.fst) :: Nondet :: Nil) a  →
  (a  →  Set)  →  Set
⟦S⟧match  =  ⟦S⟧ptRec matchSpec :: ptAll :: Nil
```

We can reuse exactly our proof that *allSplits* is correct, since we use the same semantics for the non-determinism used in the definition of *allSplits*. Similarly, the partial 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 ⋆) , Nil)        P (preH , postH)      =  postH _ StarNil
matchSound ((r ⋆) , (x :: xs)) P (preH , postH) o H  =  postH _ (StarConcat H)
```

At this point, we have defined a matcher for regular languages and formally proven that when it succeeds in recognizing a given string, this string is indeed in the language generated by the argument regular expression. However, the *match* function does not necessarily terminate: if r is a regular expression that accepts the empty string, then calling *match* on $r \star$ and a string xs will diverge. In the next section, we will write a new parser that is guaranteed to terminate and show that this parser refines the *match* function defined above.

6 Derivatives and handlers

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 will use the Brzozowski derivative [Brz64] of regular languages:

Definition 1 The Brzozowski derivative of a formal language L with respect to a character x consists of all strings xs such that $x :: xs \in L$.

Crucially, 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 remove the head x of the string and try to match $d\, r\, /d\, x$.

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 $\varepsilon?$ that decides whether an expression matches the empty string.

```
ε? : (r : Regex) → Dec (∑ (Tree r) (Match r Nil))
```

The definition of $\varepsilon?$ is given by structural recursion on the regular expression, just as the derivative operator is:

```
d_/d_ : Regex → Char → Regex
d Empty       /d c = Empty
d Epsilon     /d c = Empty
d Singleton x /d c with c ≟ x
...                 | yes p = Epsilon
...                 | no ¬p = Empty
d l · r       /d c with ε? l
...                 | yes p = ((d l /d c) · r) | (d r /d c)
...                 | no ¬p = (d l /d c) · r
d l | r       /d c = (d l /d c) | (d r /d c)
d r ⋆         /d c = (d r /d c) · (r ⋆)
```

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. As this function 'inverts' the result of derivation, we name it *integralTree*:

```
integralTree : (r : Regex) → Tree (d r /d x) → Tree r
```

Its definition closely follows the pattern matching performed in the definition of $d\_/d\_$.

The description of a derivative-based matcher is stateful: a step is done by removing a character from the input string. This state can be encapsulated in a new effect *Parser*. The *Parser* effect has one command *Symbol*. Calling *Symbol* will return the head of the unparsed remainder (advancing the string by one character) or *nothing* if the string has been totally consumed.

```
data CParser : Set where
  Symbol : CParser
RParser : CParser → Set
RParser Symbol = Maybe Char
Parser = mkSig CParser RParser
symbol : {{ iP : Parser ∈ es }} → Free es (Maybe Char)
symbol {{ iP }} = Op iP Symbol Pure
```

The code for the parser, *dmatch*, is now only a few lines long. When the input string is empty, we check that the expression matches the empty string; for a non-empty string we use the derivative to match the first character and recurse:

```
dmatch : {{ iP : Parser ∈ es }} {{ iND : Nondet ∈ es }} → Regex ↬es Tree
dmatch r = symbol >>= maybe
  (λ x → integralTree r <$> call {{ ∈Head }} (d r /d x))
  (if ε? r then (λ p → Pure (Sigma.fst p)) else fail)
```

Here, *maybe f y* takes a *Maybe* value and applies f to the value in *just*, or returns y if it is *nothing*.

Adding the new effect *Parser* to our repertoire also requires specifying its semantics. We gave the effects *Nondet* and *Rec* predicate transformer semantics in the form of a *PT* record. After introducing the *Parser* effect, the pre- and postcondition become more complicated: not only do they reference the 'pure' values passed to and returned (here of type $r : \mathit{Regex}$ and $\mathit{Tree}\ r$ respectively), there is also the current state, containing a *String*, to keep track of. With these augmented predicates, the predicate transformer semantics for the *Parser* effect can be given as:

$$
\begin{aligned}
&\mathit{ptParser} : (c : \mathit{CParser}) \to (\mathit{RParser}\ c \to \mathit{String} \to \mathit{Set}) \to \mathit{String} \to \mathit{Set} \\
&\mathit{ptParser}\ \mathit{Symbol}\ P\ \mathit{Nil} \quad\ = P\ \mathit{nothing}\ \mathit{Nil} \\
&\mathit{ptParser}\ \mathit{Symbol}\ P\ (x :: \mathit{xs}) = P\ (\mathit{just}\ x)\ \mathit{xs}
\end{aligned}
$$

In this article, we want to demonstrate the modularity of predicate transformer semantics. Thus, we do not use *ptParser* as semantics for *Parser* in the remainder, so we can illustrate how the semantics mesh well with other forms of semantics. We give denotational semantics, in the form of an effect handler for *Parser*:

$$
\begin{aligned}
&\mathit{hParser} : \{\!\{\, \mathit{iND} : \mathit{Nondet} \in \mathit{es} \,\}\!\} \to (c : \mathit{CParser}) \to \mathit{String} \to \mathit{Free}\ \mathit{es}\ (\mathit{RParser}\ c \times \mathit{String}) \\
&\mathit{hParser}\ \mathit{Symbol}\ \mathit{Nil} \quad\ = \mathit{Pure}\ (\mathit{nothing}\ , \mathit{Nil}) \\
&\mathit{hParser}\ \mathit{Symbol}\ (x :: \mathit{xs}) = \mathit{Pure}\ (\mathit{just}\ x \quad , \mathit{xs})
\end{aligned}
$$

The function *handleRec* folds a given handler over a recursive definition, allowing us to handle the *Parser* effect in *dmatch*.

$$
\begin{aligned}
&\mathit{handleRec} : ((c : C) \to s \to \mathit{Free}\ \mathit{es}\ (R\ c \times s)) \to a \overset{\mathit{mkSig}\ C\ R :: \mathit{es}}{\hookrightarrow} b \to a \times s \overset{\mathit{es}}{\hookrightarrow} b \circ \mathit{Pair.fst} \\
&\mathit{dmatch}' : \{\!\{\, \mathit{iND} : \mathit{Nondet} \in \mathit{es} \,\}\!\} \to \mathit{Regex} \times \mathit{String} \overset{\mathit{es}}{\hookrightarrow} \mathrm{Tree} \circ \mathit{Pair.fst} \\
&\mathit{dmatch}' = \mathit{handleRec}\ \mathit{hParser}\ (\mathit{dmatch})
\end{aligned}
$$

Note that *dmatch*′ has exactly the type of the previously defined *match*, allowing us to more easily compare the two matchers.

7 Proving total correctness

Since *dmatch* always consumes a character before recursing, the number of recursive calls is bounded by the length of the input string As a result, we 'handle' the recursive effect by unfolding the definition a bounded number of times. In the remainder of this section, we will make this argument precise and relate the *dmatch* function above to the *match* function defined previously.

To ensure the termination of a recursive computation, we define the following predicate, terminates-in. Given any recursive computation $f : C \overset{\mathit{es}}{\hookrightarrow} R$, we check whether the computation requires no more than a fixed number of steps to terminate:

$$
\begin{aligned}
&\text{terminates-in} : (\mathit{pts} : \mathit{PTs}\ \mathit{es}) \\
&\quad (f : C \overset{\mathit{es}}{\hookrightarrow} R)\ (S : \mathit{Free}\ (\mathit{mkSig}\ C\ R :: \mathit{es})\ a) \to \mathbb{N} \to \mathit{Set} \\
&\text{terminates-in}\ \mathit{pts}\ f\ (\mathit{Pure}\ x) \qquad\ n \qquad\ = \top \\
&\text{terminates-in}\ \mathit{pts}\ f\ (\mathit{Op} \in \mathrm{Head}\ c\ k)\ \mathit{Zero} \quad = \bot \\
&\text{terminates-in}\ \mathit{pts}\ f\ (\mathit{Op} \in \mathrm{Head}\ c\ k)\ (\mathit{Succ}\ n) = \text{terminates-in}\ \mathit{pts}\ f\ (f\ c \gg\!= k)\ n \\
&\text{terminates-in}\ \mathit{pts}\ f\ (\mathit{Op}\ (\in \mathrm{Tail}\ i)\ c\ k)\ n \quad = \\
&\quad \mathit{lookupPT}\ \mathit{pts}\ i\ c\ (\lambda\ x \to \text{terminates-in}\ \mathit{pts}\ f\ (k\ x)\ n)
\end{aligned}
$$

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. We formalize this argument in the lemma *dmatchTerminates*. Note that *dmatch′* is defined using the *hParser* handler, showing that we can mix denotational and predicate transformer semantics. The proof goes by induction on this string. Unfolding the recursive *call* gives *integralTree r* ⟨$⟩ *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) →
  terminates-in (ptAll :: Nil) (dmatch′) (dmatch′ (r , xs)) (length xs)
dmatchTerminates r Nil with ε? r
dmatchTerminates r Nil | yes p  = tt
dmatchTerminates r Nil | no ¬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 will show that *dmatch* is a refinement of *match*. By the transitivity of the refinement relation, we can conclude that it also satisfies the specification given by our original *Match* relation. The 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 : ∀ r → Match (d r /d x) xs y → Match r (x :: xs) (integralTree r y)
```

The proof is straightforward by induction on the derivation of type *Match* (*d r /d x*) *xs y*.

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

```
dmatchSound : ∀ r xs → ⟦match (r , xs)⟧match ⊑ ⟦dmatch′ (r , xs)⟧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.

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: the function *dmatch* never makes a non-deterministic choice. It will not return all possible parse trees that satisfy the *Match* relation, only the first tree that it encounters. We can, however, prove that *dmatch* will find a parse tree if it exists. To express that *dmatch* returns a result, we use a trivially true postcondition; by furthermore replacing the demonic choice of the *ptAll* semantics with the angelic choice of *ptAny*, we require that *dmatch* must return a result:

```
dmatchComplete : ∀ r xs y → Match r xs y →
  ⟦dmatch′ (r , xs)⟧ptRec matchSpec :: ptAny :: Nil (λ _ → ⊤)
```

The proof is short, since *dmatch* can only *fail* when it encounters an empty string and a regular expression that does not match the empty string, which contradicts the assumption *Match r xs y*:

```
dmatchComplete r Nil y H with ε? r
... | yes p = tt
```

```
... | no ¬p = ¬p (_ , H)
dmatchComplete r (x :: xs) y H y' H' = tt
```

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.

8 Discussion

8.1 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. The main body, 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 appendices 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.

8.2 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, and how an individual handler interacts with these semantics. 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.

8.3 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.

[Swi08] Wouter Swierstra. “Data types à la carte”. In: Journal of Functional Programming 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

[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.

A 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 $s \to Set$. We define the resulting type of stateful predicate transformers for an effect with signature e to be $\mathrm{PT}^S\ s\ e$, as follows:

$$
\begin{array}{l}
\textbf{record}\ \mathrm{PT}^S\ (s : Set)\ (e : Sig) : Set\ \textbf{where} \\
\quad \textbf{constructor}\ mkPTS \\
\quad \textbf{field} \\
\qquad pt : (c : C\ e) \to (R\ e\ c \to s \to Set) \to s \to Set \\
\qquad mono : \forall\ c\ P\ P' \to (\forall\ x\ t \to P\ x\ t \to P'\ x\ t) \to pt\ c\ P \subseteq pt\ c\ P'
\end{array}
$$

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

$$
\begin{array}{l}
[\![\cdot]\!]^S : (pts : \mathrm{PTs}^S\ s\ es) \to Free\ es\ a \to (a \to s \to Set) \to s \to Set \\
[\![Pure\ x]\!]^S_{pts}\ P = P\ x \\
[\![Op\ i\ c\ k]\!]^S_{pts}\ P = lookupPTS\ pts\ i\ c\ \lambda\ x \to [\![k\ x]\!]^S_{pts}\ P
\end{array}
$$

In this definition for $[\![\cdot]\!]^S$, 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.

$$
\begin{array}{l}
ptParse : \mathrm{PT}^S\ String\ Parser \\
PTS.pt\ ptParse\ Symbol\ P\ Nil = \top \\
PTS.pt\ ptParse\ Symbol\ P\ (x :: xs) = P\ x\ xs
\end{array}
$$

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.

$$
\begin{array}{l}
empty? : List\ a \to Set \\
empty?\ Nil = \top \\
empty?\ (\_ :: \_) = \bot \\
\_ \in [\_] : String \to Free\ (Nondet :: Parser :: Nil)\ a \to Set \\
xs \in [\ S\ ] = [\![S]\!]^S_{ptAll\ ::\ ptParse\ ::\ Nil}\ (\lambda\ \_ \to empty?)\ xs
\end{array}
$$

B 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 non-terminal symbols.

```
record GrammarSymbols : Set where
  field
    Nonterm : Set
    ⟦_⟧ : Nonterm → Set
    _≟_ : Decidable {A = Nonterm} _==_
```

The elements of the type *Char* are the terminal symbols. The elements of the type *Nonterm* are the non-terminal symbols, representing the language constructs. As for *Char*, we also need to be able to decide the equality of non-terminals. 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 non-terminal A, our goal is to parse a string into a value of type $[\![A]\!]$, based on a set of production rules. A production rule $A \to xs$ gives a way to expand the non-terminal 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 non-terminal 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 → Nonterm → Set
⟦ Nil         ∣ A ⟧ = ⟦ A ⟧
⟦ Inl x  :: xs ∣ A ⟧ = ⟦ xs ∣ A ⟧
⟦ Inr B :: xs ∣ A ⟧ = ⟦ B ⟧ → ⟦ xs ∣ A ⟧
```

Now we can define the type of production rules. A rule of the form $A \to BcD$ 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  : (A : Nonterm) → FreeParser ⟦ A ⟧
filterLHS  : (A : Nonterm) → Prods → List (ProdRHS A)
fromProd   : ProdRHS A → FreeParser ⟦ A ⟧
buildParser : (xs : Symbols) → FreeParser (⟦ xs ‖ A ⟧ → ⟦ A ⟧)
exact      : a → Char → FreeParser a
```

The main function is *fromProds*: given a non-terminal, 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) with A ≟ lhs
... | yes refl = prodrhs rhs sem :: filterLHS A ps
... | no _     = 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 >>= λ f → 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 non-terminal symbol.

```
buildParser Nil = Pure id
buildParser (Inl x :: xs) = exact tt x >>= λ _ → buildParser xs
```

```
buildParser (Inr B :: xs) = do
  x ← call B
  o ← buildParser xs
  Pure λ f → 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 >>= λ t' → if t ≟ t' then Pure x else fail
```

C 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 non-terminal, 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 non-terminal, 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 $\_\vdash\_\in[\![\_]\!]\Rightarrow\_,\_$ (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⟧fromProd prods = ⟦S⟧^S_(pts prods)
partialCorrectness : (prods : Prods) (A : Nonterm) →
  ⟦⊤, (parserSpec prods A)⟧spec ⊑ ⟦fromProds prods A⟧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 $\_ \vdash \_ \sim \_ \Rightarrow \_ , \_$ relation. Thus, we want to prove the following lemma:

$$
\begin{array}{l}
\mathit{parseStep} : \forall\, A\ \mathit{xs}\ P\ \mathit{str} \rightarrow \\
\quad (\forall\, o\ \mathit{str}' \rightarrow \mathit{prods} \vdash \mathit{str} \sim \mathit{xs} \Rightarrow o\ , \mathit{str}' \rightarrow P\ o\ \mathit{str}') \rightarrow \\
\quad [\![\mathit{buildParser}\ \mathit{prods}\ \mathit{xs}]\!]_{\mathrm{fromProd}}\ \mathit{prods}\ P\ \mathit{str}
\end{array}
$$

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 $\_ \gg\!= \_$ operator in a few places.

$$
\begin{array}{l}
\mathit{parseStep}\ A\ \mathit{Nil}\ P\ t\ H = H\ \mathit{id}\ t\ \mathit{Done} \\
\mathit{parseStep}\ A\ (\mathit{Inl}\ x :: \mathit{xs})\ P\ \mathit{Nil}\ H = \mathit{tt} \\
\mathit{parseStep}\ A\ (\mathit{Inl}\ x :: \mathit{xs})\ P\ (x' :: t)\ H\ \mathbf{with}\ x \stackrel{?}{=} x' \\
\ldots \mid \mathit{yes}\ \mathit{refl} = \mathit{parseStep}\ A\ \mathit{xs}\ P\ t\ \lambda\ o\ t'\ H' \rightarrow H\ o\ t'\ (\mathit{Next}\ H') \\
\ldots \mid \mathit{no}\ \neg p = \mathit{tt} \\
\mathit{parseStep}\ A\ (\mathit{Inr}\ B :: \mathit{xs})\ P\ t\ H\ o\ t'\ \mathit{Ho} = \\
\quad \mathit{wpToBind}\ (\mathit{buildParser}\ \mathit{prods}\ \mathit{xs})\ \_\ \_ \\
\qquad (\mathit{parseStep}\ A\ \mathit{xs}\ \_\ t'\ \lambda\ o'\ \mathit{str}'\ \mathit{Ho}' \rightarrow H\ \_\ \_\ (\mathit{Call}\ \mathit{Ho}\ \mathit{Ho}'))
\end{array}
$$

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 parametrize *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 $\_ \gg\!= \_$ operator.

$$
\begin{array}{l}
\mathit{filterStep} : \forall\, \mathit{prods}'\ A \rightarrow (p \in \mathit{prods}' \rightarrow p \in \mathit{prods}) \rightarrow \\
\quad [\![\top, (\mathit{parserSpec}\ \mathit{prods}\ A)]\!]_{\mathrm{spec}} \sqsubseteq [\![\mathit{choices}\ (\mathit{map}\ (\mathit{fromProd}\ \mathit{prods})\ (\mathit{filterLHS}\ \mathit{prods}\ A\ \mathit{prods}'))]\!]_{\mathrm{fromProd}}\ \mathit{prods} \\
\mathit{filterStep}\ \mathit{Nil}\ A\ \mathit{subset}\ P\ \mathit{xs}\ H = \mathit{tt} \\
\mathit{filterStep}\ (\mathit{prod}\ \mathit{lhs}\ \mathit{rhs}\ \mathit{sem} :: \mathit{prods}')\ A\ \mathit{subset}\ P\ \mathit{xs}\ H\ \mathbf{with}\ A \stackrel{?}{=} \mathit{lhs} \\
\mathit{filterStep}\ (\mathit{prod}\ .A\ \mathit{rhs}\ \mathit{sem} :: \mathit{prods}')\ A\ \mathit{subset}\ P\ \mathit{xs}\ (\_\ , H) \mid \mathit{yes}\ \mathit{refl} \\
\quad = \mathit{wpToBind}\ (\mathit{buildParser}\ \mathit{prods}\ \mathit{rhs})\ \_\ \_ \\
\quad (\mathit{parseStep}\ A\ \mathit{rhs}\ \_\ \mathit{xs}\ \lambda\ o\ t'\ H' \rightarrow H\ \_\ \_\ (\mathit{Produce}\ (\mathit{subset} \in \mathrm{Head})\ H')) \\
\quad , \mathit{filterStep}\ \mathit{prods}'\ A\ (\mathit{subset} \circ \in \mathrm{Tail})\ P\ \mathit{xs}\ (\_\ , H) \\
\ldots \mid \mathit{no}\ \neg p = \mathit{filterStep}\ \mathit{prods}'\ A\ (\mathit{subset} \circ \in \mathrm{Tail})\ P\ \mathit{xs}\ H
\end{array}
$$

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

D Termination of the parser

To show termination we need a somewhat more subtle argument: since we are able to call the same non-terminal repeatedly, termination cannot be shown simply by inspecting each alternative in the definition. Consider the grammar given by $E \rightarrow aE; E \rightarrow b$, 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 non-terminal 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 non-terminal. This means that the number of recursive calls to *fromProds* is bounded by the length of the string times the number of different non-terminals 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 non-terminals $A, \ldots, A_i, A_{i+1}, \ldots, B$, 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 non-terminals. 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 $\_\prec\_$ 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.

$$\begin{aligned}&\textbf{data}\ \mathit{Acc}\ (\_\prec\_ : a \to a \to \mathit{Set}) : a \to \mathit{Set}\ \textbf{where}\\ &\quad \mathit{acc} : (\forall\, y \to y \prec x \to \mathit{Acc}\ \_\prec\_\ y) \to \mathit{Acc}\ \_\prec\_\ x\end{aligned}$$

To see that this is equivalent to the definition of well-foundedness in set theory, recall that a relation $\_\prec\_$ 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 $\mathit{Acc}\ \_\prec\_\ x$ 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 $f : I \overset{es}{\nrightarrow} O$, 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:

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

$$\begin{aligned}&\mathit{variant} : (\mathit{pts} : \mathrm{PTs}^S\ s\ (\mathit{mkSig}\ C\ R :: es))\ (f : C \overset{es}{\nrightarrow} R) \to\\ &\quad (\_\prec\_ : (C \times s) \to (C \times s) \to \mathit{Set}) \to \mathit{Set}\\ &\mathit{variant}\ \mathit{pts}\ f\ \_\prec\_ = \forall\, c\ t \to \mathit{variant}'\ \mathit{pts}\ f\ \_\prec\_\ c\ t\ (f\ c)\ t\end{aligned}$$

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.

```
record Termination (pts : PTs^S s (mkSig C R :: es)) (f : C -es-> R) : Set where
  field
    _≺_ : (C × s) → (C × s) → Set
    w−f : ∀ c t → Acc _≺_ (c , t)
    var : variant pts f _≺_
```

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*:

```
data RecOrder (prods : Prods) : (x y : Nonterm × String) → Set where
  Left : length str < length str′ →
    RecOrder prods (A , str) (B , str′)
  Right : length str ≤ length str′ →
    LRec prods A B → RecOrder prods (A , str) (B , str′)
```

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* : $\mathbb{N}$.

```
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 non-terminal *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 non-terminal 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 non-terminals 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 ≟ 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.

E Discussion

E.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. The main body, 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 appendices 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.

E.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, and how an individual handler interacts with these semantics. 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.

E.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.

[Swi08] Wouter Swierstra. “Data types à la carte”. In: Journal of Functional Programming 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

[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.