Verified parsers using the refinement calculus and algebraic effects

Tim Baanen and Wouter Swierstra

Vrije Universiteit Amsterdam, Utrecht University
{t.baanen@vu.nl,w.s.swierstra@uu.nl}

There are various ways to write a parser in functional languages, for example using parser combinations. How do we ensure these parsers are correct? Previous work has shown that predicate transformers are useful for verification of programs using algebraic effects. This paper will show how predicate transformers and algebraic effects allow for formal verification of parsers.

1 Recap: algebraic effects and predicate transformers

Algebraic effects were introduced to allow for incorporating side effects in functional languages. For example, the effect *ENondet* allows for nondeterministic programs:

```
record Effect : Set where
  constructor eff
  field
    C : Set
    R : C → Set
data CNondet : Set where
  Fail : CNondet
  Choice : CNondet
RNondet : CNondet → Set
RNondet Fail = ⊥
RNondet Choice = Bool
ENondet = eff CNondet RNondet
```

We represent effectful programs using the *Free* datatype.

```
data Free (e : Effect) (a : Set) : Set where
  Pure : a → Free e a
  Step : (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
Step c k ≫= f = Step c (λ x → k x ≫= f)
```

The easiest way to use effects is with smart constructors:

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

To give specifications of programs that incorporate effects, we can use predicate transformers.

```
wp : {C : Set} {R : C → Set} → ((c : C) → (R c → Set) → Set) →
  {a : Set} → Free (eff C R) a → (a → Set) → Set
wp alg (Pure x) P = P x
wp alg (Step c k) P = alg c λ x → wp alg (k x) P
```

Interestingly, these predicate transformers are exactly the catamorphisms from *Free* to *Set*.

```
ptAll : (c : CNondet) → (RNondet c → Set) → Set
ptAll Fail P = ⊤
ptAll Choice P = P True ∧ P False


wpNondetAll : Free ENondet a → (a → Set) → Set
wpNondetAll S P = wp ptAll S P
```

We use pre- and postconditions to give a specification for a program. If the precondition holds on the input, the program needs to ensure the postcondition holds on the output.

```
module Spec where
  record Spec (a : Set) : Set where
    constructor [_,_]
    field
      pre : Set
      post : a → Set
  wpSpec : Spec a → (a → Set) → Set
  wpSpec [ pre , post ] P = pre ∧ (∀ o → post o → P o)
```

The refinement relation expresses when one program is “better” than another. We need to take into account the semantics we want to impose on the program, so we define it in terms of the predicate transformer associated with the program.

```
_⊑_ : (pt1 pt2 : (a → Set) → Set) → Set
pt1 ⊑ pt2 = ∀ P → pt1 P → pt2 P
```

2 Almost parsing regular languages

To see how we can use the *Free* monad for writing and verifying a parser, and more specifically how we use the *ENondet* effect for writing and the *wpNondetAll* semantics for verifying a parser, we will look at parsing a given regular language. Our approach is first to define the specification of a parser, then inspect this specification to write the first implementation and prove (partial) correctness of this implementation. We will later improve this implementation by refining it.

Definition 1 ([AU77]) *The class of* regular languages *is the smallest class such that:*

- *the empty language is regular,*
- *the language containing only the empty string is regular,*
- *for each character* `x`*, the language containing only the string* `"x"` *is regular,*
- *the union and concatenation of regular languages are regular, and*
- *the repetition of a regular language is regular.*

A regular language can defined using a regular expression, which we will represent as an element of the *Regex* datatype. An element of this type represents the syntax of a regular language, and we will generally identify a regular expression with the language it denotes.

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

Here, *Empty* is an expression for empty language (which matches no strings at all), while *Epsilon* is an expression for the language of the empty string (which matches exactly one string: `""`).

What should a parser for regular languages output? If we only want to know whether a string matches a regular expression, we can return a *Bool*. If we want to know more, we could annotate the regular expression with capture groups, and say that the output of the parser maps each capture group to the substring that the capture group matches. We can also return a full parse tree, mirroring the structure of the 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 Agda, we can represent the semantics of the *Regex* type by giving a relation between a *Regex* and a *String* on the one hand (the input of the matcher), and a parse tree on the other hand (the output of the parser). Note that the *Tree* type itself is not sufficient to represent the semantics, since it does not say which strings result in any given parse tree. If the *Regex* and *String* do not match, there should be no output, otherwise the output consists of all relevant parse trees. We give the relation using the following inductive definition:

```
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*, which we interpret as that 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*.

Since the definition of *Match* allows for multiple ways that a given *Regex* and *String* may match, such as in the trivial case where the *Regex* is of the form $r \mid r$, and it also has cases where there is no way to match a *Regex* and a *String*, such as where the *Regex* is *Empty*, we can immediately predict some parts of the implementation of the *match* function. Whenever we encounter an expression of the form $l \mid r$, we make a nondeterministic *Choice* between either l or r. Similarly, whenever we encounter the *Empty* expression, we immediately *fail*. In the previous analysis steps, we have already assumed that we implement the parser by structural recursion on the *Regex*, so let us consider other cases.

The implementation for concatenation is not as immediately obvious. One way that we can deal with it is to not only return a *Tree* from the parser. Instead, the parser also returns the unmatched portion of the string, and when we have to match a regular expression of the form $l \cdot r$ with a string *xs*, we match l with *xs* giving a left over string *ys*, then match r with *ys*. We can also do without changing the return values of the parser, by nondeterministically splitting the string *xs* into *ys* ++ *zs*. That is what we do in a helper function *allSplits*, which nondeterministically chooses such *ys* and *zs* and returns them as a pair.

```
allSplits : (xs : List a) → Free ENondet (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)})
```

Armed with this helper function, we can write the first part of a nondeterministic regular expression matcher, that does a case distinction on the expression and then checks that the string has the correct format.

```
match : (r : Regex) (xs : String) → Free ENondet (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 = do
  (ys , zs) ← allSplits xs
  y ← match l ys
  z ← match r zs
  Pure (y , z)
match (l | r) xs = choice (Inl ⟨$⟩ match l xs) (Inr ⟨$⟩ match r xs)
```

Unfortunately, we get stuck in the case of _⋆. We could do a similar construction to $l \cdot r$, where we split the string into two parts and match the first part with r and the second part with $r \star$, but this definition will be rejected by Agda, since it does not terminate. Since there is no easy way to handle this case for now, we just *fail* when we encounter a regex $r \star$.

```
match (r ⋆) xs = fail
```

Still, we can prove that this matcher works, as long as the regular expression does not contain _⋆. In other words, we can prove that the *match* function satisfies the postcondition given by the type *Match*, as long as the precondition *hasNo∗* holds:

```
hasNo∗ : Regex → Set
hasNo∗ Empty = ⊤
hasNo∗ Epsilon = ⊤
hasNo∗ (Singleton x) = ⊤
hasNo∗ (l · r) = hasNo∗ l ∧ hasNo∗ r
hasNo∗ (l | r) = hasNo∗ l ∧ hasNo∗ r
hasNo∗ (r ⋆) = ⊥
pre : (r : Regex) (xs : String) → Set
pre r xs = hasNo∗ r
post : (r : Regex) (xs : String) → Tree r → Set
post = Match
```

In order to state that *match* works correctly, we need to determine its semantics: is the nondeterminism angelic or demonic? Since the use of nondeterminism in *match* is to find all correct matches, we want that all values potentially returned are correct, as specified by the *ptAll* semantics used in *wpNondetAll*.

If we now try to give a correctness proof with respect to this pre- and postcondition, we run into an issue in cases where the definition makes use of the

_≫=_ operator. The *wp*-based semantics completely unfolds the left hand side, before it can talk about the right hand side. Whenever our matcher makes use of structural recursion on the left hand side of a _≫=_ (more specifically, in the definition of *allSplits* and in the cases of $l \cdot r$ and $l \mid r$), we cannot make progress in our proof without reducing this left hand side to a recursion-less expression. We need a lemma relating the semantics of program composition to the semantics of individual programs, which is also known as the law of consequence for traditional predicate transformer semantics.cite?

```
consequence : ∀ pt (mx : Free es a) (f : a → Free es b) →
  wp pt mx (λ x → wp pt (f x) P) == wp pt (mx ≫= f) P
consequence pt (Pure x) f = refl
consequence pt (Step c k) f = cong (pt c)
  (extensionality λ x → consequence pt (k x) f)
wpToBind : (mx : Free es a) (f : a → Free es b) →
  wp pt mx (λ x → wp pt (f x) P) → wp pt (mx ≫= f) P
wpToBind mx f H = subst id (consequence pt mx f) H
wpFromBind : (mx : Free es a) (f : a → Free es b) →
  wp pt (mx ≫= f) P → wp pt mx (λ x → wp pt (f x) P)
wpFromBind mx f H = subst id (sym (consequence pt mx f)) H
```

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 →
  wpSpec [ ⊤ , allSplitsPost xs ] ⊑ wpNondetAll (allSplits xs)
allSplitsSound Nil       P (preH , postH) = postH _ refl
allSplitsSound (x :: xs) P (preH , postH) = postH _ refl ,
  wpToBind (allSplits xs) _ (allSplitsSound xs _ (tt ,
    λ _ H → postH _ (cong (x :: _) H)))
```

Then, using *wpToBind*, we incorporate this correctness proof in the correctness proof of *match*. Apart from having to introduce *wpToBind*, the proof essentially follows automatically from the definitions.

```
matchSound : ∀ r xs →
  wpSpec [ pre r xs , post r xs ] ⊑ wpNondetAll (match r xs)
matchSound Empty xs           P (preH , postH) = tt
matchSound Epsilon Nil        P (preH , postH) = postH _ Epsilon
matchSound Epsilon (x :: xs)  P (preH , postH) = tt
matchSound (Singleton x) Nil  P (preH , postH) = tt
matchSound (Singleton x) (c :: Nil) P (preH , postH) with x ≟ c
```

```
... | yes refl = postH _ Singleton
... | no ¬p = tt
matchSound (Singleton x) (_ :: _ :: _) P (preH , postH) = tt
matchSound (l · r) xs P ((preL , preR) , postH) =
  wpToBind (allSplits xs) _ (allSplitsSound xs _ (tt ,
  λ {(ys , zs) splitH → wpToBind (match l ys) _ (matchSound l ys _ (preL ,
  λ y lH → wpToBind (match r zs) _ ((matchSound r zs _ (preR ,
  λ z rH → postH (y , z) (subst (λ xs → Match _ xs _) (sym splitH)
  (Concat lH rH)))))))}))
matchSound (l | r) xs P ((preL , preR) , postH) =
  wpToBind (match l xs) _ (matchSound l xs _ (preL ,
    λ _ lH → postH _ (OrLeft lH))) ,
  wpToBind (match r xs) _ (matchSound r xs _ (preR ,
    λ _ rH → postH _ (OrRight rH)))
matchSound (r ⋆) xs P (() , postH)
```

3 Combining nondeterminism and general recursion

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* ⋆ with the empty string `""` will allow for repeating the *Epsilon* arbitrarily often, since *Epsilon* · (*Epsilon* ⋆) is equivalent to both *Epsilon* and *Epsilon* ⋆. Thus, we cannot implement *match* on the _⋆ operator by helping Agda's termination checker.

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

```
ERec : (I : Set) (O : I → Set) → Effect
ERec I O = eff I O
```

Defining *match* with the *ERec* effect is not sufficient to implement it fully either, since replacing the effect *ENondet* with *ERec* does not allow for nondeterminism anymore, so 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 *eff* C_1 R_1 and *eff* C_2 R_2, we can define a new effect by taking the disjoint union of the commands and responses, resulting in *eff* (*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 to 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 disjount 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 effects instead of a single effect. The *Pure* constructor remains as it is, while the *Step* constructor takes an index into the list of effects and the command and continuation for the effect with this index.

```
data Free (es : List Effect) (a : Set) : Set where
  Pure : a → Free es a
  Step : (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. 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 _>>=_ 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 : ENondet ∈ es ⦄ → Free es a
fail ⦃ iND ⦄ = Step iND Fail λ ()
choice : ⦃ iND : ENondet ∈ es ⦄ → Free es a → Free es a → Free es a
choice ⦃ iND ⦄ S₁ S₂ = Step iND Choice λ b → if b then S₁ else S₂
call : ⦃ iRec : ERec I O ∈ es ⦄ → (i : I) → Free es (O i)
call ⦃ iRec ⦄ i = Step iRec i Pure
```

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

$$\begin{aligned}
&\_ \overset{-}{\looparrowright} \_ \ : \ (C \ : \ Set)\,(es \ : \ List\ Effect)\,(R \ : \ C \ \rightarrow \ Set) \ \rightarrow \ Set\\
&C \overset{es}{\looparrowright} R \ = \ (c \ : \ C) \ \rightarrow \ Free\,(eff\ \ C\ R \ :: \ es)\,(R\ c)
\end{aligned}$$

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 : Effect) : 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 Effect → Set where
  Nil : PTs Nil
  _::_ : ∀ {e es} → PT e → PTs es → 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 some lemmas later on which exactly?, 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 *wp*. The *Pure* case is identical, and in the *Step* case we find the predicate transformer at the corresponding index to the effect index $i : e \in es$ using the *lookupPT* helper function.

```
lookupPT : (pts : PTs es) (i : eff 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 *wp* for combinations of effects.

```
wp : (pts : PTs es) → Free es a → (a → Set) → Set
wp pts (Pure x) P = P x
wp pts (Step i c k) P = lookupPT pts i c λ x → wp pts (k x) 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.

However, if 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) → O i → Set) → PT (ERec 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)
```

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 terminates according to the relation R.

4 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:

$$\begin{array}{l} \mathit{allSplits} : \{\!\{\, \mathit{iND} : \mathit{ENondet} \in \mathit{es} \,\}\!\} \to \mathit{List}\ a \to \mathit{Free}\ \mathit{es}\ (\mathit{List}\ a \times \mathit{List}\ a) \\ \mathit{match} : \{\!\{\, \mathit{iND} : \mathit{ENondet} \in \mathit{es} \,\}\!\} \to \mathit{Regex} \times \mathit{String} \overset{es}{\hookrightarrow} \mathit{Tree} \circ \mathit{Pair.fst} \end{array}$$

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 ⋆) , 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
wpMatch : Free (ERec (Pair Regex String) (Tree ∘ Pair.fst) ∷ ENondet ∷ Nil) a →
  (a → Set) → Set
wpMatch = wp (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 ⋆) , 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 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 $r \star$ 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.

5 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 2 ([Brz64]) *The* Brzozowski derivative *of a formal language L with respect to a character x consists of all strings xs such that $x :: xs \in L$.*

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

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

The definitions 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 ⋆)
```

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) → Tree (d r /d x) → 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.

$$\begin{array}{l}
\mathit{dmatch} : \{\!\{\, \mathit{iND} : \mathit{ENondet} \in \mathit{es} \,\}\!\} \to \mathit{Regex} \times \mathit{String} \overset{es}{\looparrowright} \mathit{Tree} \circ \mathit{Pair.fst} \\
\mathit{dmatch}\ (r \mathbin{,} \mathit{Nil})\ \textbf{with}\ \varepsilon?\ r \\
...\ \mid\ \mathit{yes}\ (\mathit{ms} \mathbin{,} \_) \ = \mathit{Pure\ ms} \\
...\ \mid\ \mathit{no}\ \neg p \qquad\quad = \mathit{fail} \\
\mathit{dmatch}\ (r \mathbin{,} (x :: \mathit{xs})) = \mathit{integralTree}\ r\ \langle\$\rangle\ \mathit{call}\ ((d\ r\ /d\ x) \mathbin{,} \mathit{xs})
\end{array}$$

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:

$$\begin{array}{l}
\mathit{while} : \{\!\{\, \mathit{iRec} : \mathit{ERec}\ a\ (K\ a) \in \mathit{es} \,\}\!\} \to \\
\quad (a \to \mathit{Bool}) \to (a \to a) \to (a \to \mathit{Free\ es}\ a) \\
\mathit{while\ cond\ body}\ i = \textbf{if}\ \mathit{cond}\ i\ \textbf{then}\ \mathit{Pure}\ i\ \textbf{else}\ (\mathit{call}\ (\mathit{body}\ i))
\end{array}$$

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.

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

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 $\mathit{integralTree}\ \langle\$\rangle\ \mathit{dmatch}\ (d\ r\ /d\ x \mathbin{,} \mathit{xs})$, which we rewrite using the associativity monad law in a lemma called *terminates-fmap*.

$$\begin{array}{l}
\mathit{dmatchTerminates} : (r : \mathit{Regex})\ (\mathit{xs} : \mathit{String}) \to \\
\quad \mathit{terminates\text{-}in}\ (\mathit{ptAll} :: \mathit{Nil})\ (\mathit{dmatch})\ (\mathit{dmatch}\ (r \mathbin{,} \mathit{xs}))\ (\mathit{length\ xs})
\end{array}$$

```
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 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 : ∀ r →
  Match (d r /d x) xs y → 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 *allSplits* returns all possible splittings of a string.

```
allSplitsComplete : (xs ys zs : String) (P : String × String → Set) →
  wpMatch (allSplits xs) P → (xs == ys ++ zs) → P (ys , zs)
```

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 : ∀ r xs →
  wpMatch (match (r , xs)) ⊑ wpMatch (dmatch (r , xs))
```

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 : ∀ r xs y → Match r xs y →
  wp (ptRec matchSpec :: ptAny :: Nil) (dmatch (r , xs)) (λ _ → ⊤)
```

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 *EParser* 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.

6 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 *Parser* monad:

```
Parser : Set → Set
Parser 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 *EParser* 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
EParser = eff CParser RParser
symbol : ⦃ iP : EParser ∈ es ⦄ → Free es Char
symbol ⦃ iP ⦄ = Step 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 *EParser* 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 *EParser* with the nondeterminism effect *ENondet*, and the general recursion effect *ERec*.

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

```
toParser : Free (ENondet :: EParser :: Nil) a → Parser a
toParser (Pure x) Nil = (x , Nil) :: Nil
toParser (Pure x) (_ :: _) = Nil
toParser (Step ∈Head Fail k) xs = Nil
toParser (Step ∈Head Choice k) xs =
  toParser (k True) xs ++ toParser (k False) xs
toParser (Step (∈Tail ∈Head) Symbol k) Nil = Nil
toParser (Step (∈Tail ∈Head) Symbol k) (x :: xs) = toParser (k x) xs
```

In this article, we are more interested in the predicate transformer semantics of *EParse*. Since the semantics of the *EParse* effect 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 *e* to be $PT^S\ s\ e$, as follows:

```
record PT^S (s : Set) (e : Effect) : 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 can find a weakest precondition that incorporates the current state:

```
wp^S : (pts : PTs^S s es) → Free es a → (a → s → Set) → s → Set
wp^S pts (Pure x) P = P x
wp^S pts (Step i c k) P = lookupPTS pts i c λ x → wp^S pts (k x) P
```

In this definition for wp^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 *EParser* 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 *ENondet*. Using *ptAll*'s semantics for the *Fail* command gives the following semantics for the *EParser* effect.

```
ptParse : PT^S String EParser
PTS.pt ptParse Symbol P Nil = ⊤
PTS.pt ptParse Symbol P (x :: xs) = P x xs
```

With the predicate transformer semantics of *EParse*, 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 → Set
empty? Nil = ⊤
empty? (_ :: _) = ⊥
_∈[_] : String → Free (ENondet :: EParser :: Nil) a → Set
xs ∈ [ S ] = wp^S (ptAll :: ptParse :: Nil) S (λ _ → empty?) xs
```

7 Parsing context-free languages

In Section 5, 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
    ⟦_⟧ : Nonterm → Set
    _≟_ : 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 $\llbracket A \rrbracket$, based on a set of production rules. A production rule $A \to xs$ 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 $\llbracket B \rrbracket$, a production rule for A is associated with a *semantic function* that takes all values arising from submatches and returns a value of type $\llbracket A \rrbracket$, 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.

8 From abstract grammars to abstract parsers

We want to show that a generally recursive function making use of the effects *EParser* and *ENondet* 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 (eff Nonterm ⟦_⟧ ∷ ENondet ∷ EParser ∷ 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 nonterminal, it selects the productions with this nonterminal on the left hand side using *filterLHS*, and makes a nondeterministic choice between the productions.

```
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 = foldr (choice) (fail) (map fromProd (filterLHS A prods))
```

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 nonterminal 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
```

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
wpFromProd prods = wp^S (pts prods)
partialCorrectness : (prods : Prods) (A : Nonterm) →
  wpSpec [ ⊤ , (parserSpec prods A) ] ⊑
    wpFromProd prods (fromProds prods A)
```

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′) →
  wpFromProd prods (buildParser prods xs) 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 with x ≟ x′
... | yes refl = parseStep A xs P t λ o t′ H′ → H o t′ (Next H′)
... | no ¬p = tt
parseStep A (Inr B :: xs) P t H o t′ Ho =
  wpToBind (buildParser prods xs) _ _
    (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 : ∀ prods′ → (p ∈ prods′ → p ∈ prods) →
  ∀ A → wpSpec [ ⊤ , parserSpec prods A ] ⊑ wpFromProd prods
```

```
      (foldr (choice) (fail) (map (fromProd prods) (filterLHS prods A prods′)))
  filterStep Nil subset A P xs H = tt
  filterStep (prod lhs rhs sem :: prods′) subset A P xs H with A ≟ lhs
  filterStep (prod .A rhs sem :: prods′) subset A P xs (_ , H) | yes refl
     = wpToBind (buildParser prods rhs) _ _
     (parseStep A rhs _ xs λ o t′ H′ → H _ _ (Produce (subset ∈Head) H′))
     , filterStep prods′ (subset ∘ ∈Tail) A P xs (_ , H)
  ... | no ¬p = filterStep prods′ (subset ∘ ∈Tail) A 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 3 ([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.*

```
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 $\_\prec\_$ 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 4 *Given a recursive definition* $f : I \overset{es}{\looparrowright} O$*, 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:*

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

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

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.

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

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 $\mathit{Acc}\ \_{\prec}\_\ c$ 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*:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{RecOrder}\ (\mathit{prods} : \mathit{Prods}) : (x\ y : \mathit{Nonterm} \times \mathit{String}) \to \mathit{Set}\ \mathbf{where}\\
&\quad \mathit{Adv} : \mathit{length}\ \mathit{str} < \mathit{length}\ \mathit{str'} \to
\end{aligned}
$$

```
    RecOrder prods (A , str) (B , str′)
  Rec : 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* : 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 *Adv* 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 *Rec* 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.

11 Conclusions and discussion

Fill this!