Intrinsic Verification of a Regular Expression Matcher

Joomy Korkut, Maksim Trifunovski, Daniel R. Licata*
Wesleyan University

January 15, 2016

Abstract

Harper's 1999 Functional Pearl on regular expression matching is a strong example of the interplay between programming and proof, and has been used for many years in introductory functional programming classes. In this paper, we revisit this algorithm from the point of view of dependently typed programming. In the process of formalizing the algorithm and its correctness using the Agda proof assistant, we found three interesting variations. First, defunctionalizing the matcher allows Agda to see termination without an explicit metric, and provides a simple first-order matcher with a clear relationship to the original, giving an alternative to a later Educational Pearl by Yi. Second, intrinsically verifying the soundness of the algorithm has useful computational content, allowing the extraction of matching strings from the parse tree. Third, while Harper uses a negative definition of *standard* regular expressions (no starred subexpression accepts the empty string), using a syntactic definition of standardness simplifies the staging of the development. These variations provide a nice illustration of the benefits of thinking in a dependently typed language, and have some pedagogical value for streamlining and extending the presentation of this material.

*This material is based on research sponsored in part by by The United States Air Force Research Laboratory under agreement number FA9550-15-1-0053. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government or Carnegie Mellon University.

Contents

1 Introduction

Regular expression matching is a venerable problem, studied in many programming and theory of computation courses. Harper (Harper, 1999) presents a higher-order algorithm for regular expression matching, using continuation-passing to store the remainder of a matching problem when a concatenation is encountered, while using the ordinary control stack to represent the branching when an alternation is encountered. The code for the matcher is quite short, but also quite subtle; the emphasis of Harper's paper is on how the

correctness proof for the matcher informs the reader's understanding of the code. For example, the first matcher presented has a termination bug, which is revealed when the induction in the correctness proof breaks down. The problem can be fixed by restricting the domain of the function to *standard* regular expressions, which have no Kleene-stared subexpressions that accept the empty string, and then using a preprocessor translation to solve the original problem. Harper's algorithm has been used in first- and second-year functional programming courses at Carnegie Mellon for around 20 years, as a high-water example of integrating programming and program verification. A later paper by Yi (Yi, 2006) revisits the example, motivated by the author's sense that the higher-order matcher is too difficult for students in their introductory programming course, and gives a first-order matcher based on compilation to a state machine.

Motivated by its strong interplay between programming and proof and its pedagogical usefulness, we set out to formalize Harper's algorithm using the Agda proof assistant (Norell, 2007), believing that it could be a pedagogically useful example of dependently typed programming. The process of mechanizing the algorithm led us to a few new observations that streamline and extend its presentation—which was quite surprising to the third author, who has previously taught this material several times. Our goal in this paper is to document these variations on the matching algorithm, and how the process of programming it in Agda led us to them.

The three variations are as follows. First, because Agda is a total language, we must make the termination of the matcher evident in the code itself. Harper's original algorithm can be shown to terminate using lexicographic induction on first the regular expression and second well-founded induction on the string being matched, but in Agda the latter requires passing an explicit termination measure. We discovered that *defunctionalizing* (Reynolds, 1972) the matcher avoids the explicit termination measure, because the problematic recursive call is moved from the Kleene star case to the character literal base case, where it is clear that the string is getting smaller. The defunctionalized matcher is of interest not only because programming it in Agda is simpler: it also achieves Yi's goal of a first-order matcher in a simple way, which has a clear relationship to the higher-order matcher. Pedagogically, it could be used at a point in a course before higher-order functions have been introduced, and it could be used as a stepping stone to the more sophisticated higher-order matcher.

Second, the matcher discussed in Harper's paper, whose Agda type is

```
_accepts_ : RegExp → String → Bool
```

determines whether or not a string is accepted by a regular expression. However, for most applications of regexp matching (and for making compelling homework assignments), it is useful to allow a "bracket" or "grouping" construct that allows the user to specify sub-regular-expressions whose matching strings should be reported— for example, AG [.*] TC [(G⊕C) *] GA for extracting the parts of a DNA string surrounded by certain signal codes. When coding a program/proof in a dependently typed language, there is a choice between "extrinsic" verification (write the simply-typed code, and then prove it correct) and "intrinsic verification" (fuse the program and proof into one function, with a dependent type). We have formalized both a straightforward extrinsic verification, and an intrinsically *sound* verification[1], which has the dependent type

$$\mathsf{accepts\text{-}intrinsic} : (\mathsf{r} : \mathsf{RegExp}) \to (\mathsf{s} : \mathsf{List\ Char}) \to \mathsf{Maybe}\ (\mathsf{s} \in \mathsf{L}\ \mathsf{r})$$

When this matcher succeeds, it returns the derivation that the string is in the language of the regexp; completeness, which says that the matcher does not improperly reject strings, is still proved separately. The reason for this choice is that the *computational content* of the soundness proof is relevant to the above problem: the derivation gives a parse tree, which allows reporting the matching strings for each specified sub-expression. Indeed, even for the extrinsic matcher, one can run the separate soundness proof to produce the matching strings—but running the matcher and then its soundness proof (which has success of the matcher as a precondition) duplicates work, so we present the intrinsic version in the paper. Though we were led to this variation by coding the soundness proof of the matcher using dependent types, analogous code could be used in a simply typed language, with the less informative result type Maybe Derivation which does not say what string and regexp it is a derivation for.

A third variation is that, while Harper uses a negative semantic definition of standard regular expressions ("no subexpression of the form r^* accepts the empty string"), it is often more convenient in Agda to use positive/inductive criteria. While formalizing the notion of standard, we realized that it is possible to instead use a syntactic criterion, generating standard regular expressions by literals, concatenation, alternation, and Kleene *plus* (one or more occurrences) instead of Kleene *star* (zero or more occurrences), and omitting the empty string regexp ε. While the syntactic criterion omits some semantically standard expression (such as (ε · r) *, where r does not

[1]All formalizations are available from `http://github.com/joom/regexp-agda`. Use Agda version 2.4.2.2 with standard library version 0.11

accept the empty string), it still suffices to define a matcher for all regular expression. In addition to simplifying the Agda formalization, this observation has the pedagogical benefit of allowing a self-contained treatment of these syntactically standard regular expressions.

Though dependently typed programming led us to these new insights into a problem that has been very thoroughly studied from a very similar point of view, they can all be ported back to simply-typed languages. Thus, in addition to being a strong pedagogical example of dependently typed programming, these variations on regular expression matching could be used in introductory programming courses to offer a streamlined treatment—e.g. using the defunctionalized matcher for only syntactically standard regular expressions, which still captures the basic interplay between programming and proof—which scales to higher-order matching and more interesting homework assignments—e.g. by computing matching strings. Therefore, even though there is existing work on verified parsing of regular expressions and context-free grammars (see (Danielsson, 2010; Ridge, 2011; Firsov & Uustalu, 2013) for work using other algorithms; Wouter Swierstra and collaborators also worked on formalizing Harper's algorithm, but had unresolved termination issues[2]), we believe these new variations on Harper's algorithm will be of interest to the dependent types and broader functional programming communities.

The remainder of this paper is organized as follows. In Section 2, we define syntactically standard regular expressions. In Section 3, we give an intrinsically sound defunctionalized matcher, with no explicit termination measure. In Section 4, we give an intrinsically sound higher-order matcher, which uses an explicit termination measure, and explain the correspondence with the defunctionalized matcher. In Section 5, we show that these matchers suffice to match all regular expressions by translation. In Section 5.1 we discuss how to extract matching strings.

1.1 Agda Definitions

We assume the reader is familiar with Agda (Norell, 2007) and the notation from the Agda standard library. Though the Agda library differentiates between strings (String) and lists of characters (List Char), we will ignore this distinction in the paper. Because the intrinsically verified matcher returns a Maybe/option type, it will be useful to use monadic notation for Maybe (in Haskell terminology, _||_ is mplus and map is fmap):

[2]personal communication

```
return : ∀ {A} → A → Maybe A
return = just
fail : ∀ {A} → Maybe A
fail = nothing
_»=_ : ∀ {A B} → Maybe A → (A → Maybe B) → Maybe B
just x >>= f = f x
nothing >>= f = nothing
_||_ : ∀ {A} → Maybe A → Maybe A → Maybe A
just x || _ = just x
nothing || y = y
map : ∀ {A B} → (A → B) → Maybe A → Maybe B
map f (just x) = just (f x)
map _ nothing = nothing
```

2 Syntactically standard regular expressions

Rather than Harper's negative semantic criterion (no starred subexpression accepts the empty string), we use an inductive definition of standard regular expressions. Compared with the standard grammar of regular expressions, which includes the regexp matching the empty string (ε), the regexp matching the empty language ($\varnothing$), character literals, concatenation ($r_1 \cdot r_2$), alternation ($r_1 \oplus^s r_2$), and Kleene star/repetition (r^*), we omit ε, and replace Kleene star with Kleene plus, which represents repetition one or more times. As an operator on regular languages, Kleene plus (Σ^+) is equivalent to $\Sigma \cdot \Sigma^*$, where Σ^* is the Kleene star.

In Agda, we define a type of StdRegExp as follows:

```
data StdRegExp : Set where
  ∅ˢ : StdRegExp
  Litˢ : Char → StdRegExp
  _·ˢ_ : StdRegExp → StdRegExp → StdRegExp
  _⊕ˢ_ : StdRegExp → StdRegExp → StdRegExp
  _⁺ˢ : StdRegExp → StdRegExp
```

$\varnothing^s$ is the regular expression matching no strings, Lit^s is the character literal, $\_\cdot^s\_$ is concatenation, $\_\oplus^s\_$ is alternation, and $\_^{+s}$ is Kleene plus. We use the ^{s} superscript to differentiate standard regular expressions from the the full language, which is defined in Section 5.

Informally, Kleene plus is defined as the least language closed under the following rules:

$$\frac{s \in L(\mathsf{r})}{s \in L(\mathsf{r}\ ^{+\mathsf{s}})} \qquad \frac{s_1 s_2 = s \quad s_1 \in L(\mathsf{r}) \quad s_2 \in L(\mathsf{r}\ ^{+\mathsf{s}})}{s \in L(\mathsf{r}\ ^{+\mathsf{s}})}$$

In Agda, we represent sets of strings by something analogous to their membership predicates. For standard regular expressions, we define a binary relation $\mathsf{s} \in \mathsf{L}^{\mathsf{s}}\ \mathsf{r}$, which means s is in the language of r, by recursion on r, illustrating how it is possible to compute types based on values in a dependently typed language. This uses an auxiliary, inductively definition relation $\mathsf{s} \in \mathsf{L}^{+}\ \mathsf{r}$, represented by an Agda inductively defined datatype family, which corresponds to the inference rules above.

```
mutual
  _∈L^s_ : List Char → StdRegExp → Set
  _ ∈L^s ∅^s = ⊥
  s ∈L^s (Lit^s c) = s ≡ [c]
  s ∈L^s (r₁ ⊕^s r₂) = (s ∈L^s r₁) ⊎ (s ∈L^s r₂)
  s ∈L^s (r₁ ·^s r₂) =
    Σ (List Char × List Char) (λ {(p , q) → (p ++ q ≡ s) × (p ∈L^s r₁) × (q ∈L^s r₂)})
  s ∈L^s (r ^+s) = s ∈L^+ r
  data _∈L^+_ : List Char → StdRegExp → Set where
    S+ : ∀ {s r} → s ∈L^s r → s ∈L^+ r
    C+ : ∀ {s s₁ s₂ r} → s₁ ++ s₂ ≡ s → s₁ ∈L^s r → s₂ ∈L^+ r → s ∈L^+ r
```

Here, $\bot$ is the empty Agda type, $\uplus$ is disjoint union (Either), and $\times$ is the pair type. $\Sigma\ \mathsf{A}\ (\lambda\ \mathsf{x} \to \mathsf{B})$ is an existential/dependent pair, where the type of the second component depends on the value of the first—because this is not built in in Agda, it takes a type A and a function from A to Set as arguments. The notation $\lambda\ \{\mathsf{p} \to \mathsf{e}\}$ allows a pattern-matching anonymous function. The function $+\!\!+$ appends lists, and $\equiv$ is Agda's propositional equality. Thus, in full, the clause for alternation means "there exist strings p and q such that appending p and q gives s, where p is in the language of r_1 and q is in the language of r_2".

However, is is important to note that these membership "predicates" land in Set, the Agda type of types, and thus may have computational content. For example, a witness that $\mathsf{s} \in \mathsf{L}^{\mathsf{s}}\ (\mathsf{r}_1 \oplus \mathsf{r}_2)$ includes a bit (inj_1 or inj_2) that tells which possibility was taken, and a witness $\mathsf{s} \in \mathsf{L}^{+}\ \mathsf{r}$ is a non-empty list of strings matching r, which concatenate to s. Thus, there can be different witnesses that a string matches a regular expression, such as

$$\text{Derivation A} := \frac{\text{“}a\text{”} \in L(\mathsf{Lit^s\ 'a'})}{\text{“}a\text{”} \in L(\mathsf{Lit^s\ 'a'\ {+^s}})}$$

$$\text{Derivation B} := \frac{\text{“}a\text{”} +\!\!+ \text{“}a\text{”} = \text{“}aaa\text{”} \qquad \text{“}a\text{”} \in L(\mathsf{Lit^s\ 'a'}) \qquad \text{Derivation A}}{\text{“}aa\text{”} \in L(\mathsf{Lit^s\ 'a'\ {+^s}})}$$

$$\text{Derivation C} := \frac{\text{“}a\text{”} +\!\!+ \text{“}aa\text{”} = \text{“}aaa\text{”} \qquad \text{Derivation A} \qquad \text{Derivation B}}{\text{“}aaa\text{”} \in L((\mathsf{Lit^s\ 'a'\ {+^s}}) \cdot^s (\mathsf{Lit^s\ 'a'\ {+^s}}))}$$

$$\text{Derivation D} := \frac{\text{“}aa\text{”} +\!\!+ \text{“}a\text{”} = \text{“}aaa\text{”} \qquad \text{Derivation B} \qquad \text{Derivation A}}{\text{“}aaa\text{”} \in L((\mathsf{Lit^s\ 'a'\ {+^s}}) \cdot^s (\mathsf{Lit^s\ 'a'\ {+^s}}))}$$

We will exploit this in Section 5.1 to extract matching strings from such derivations.

3 Defunctionalized intrinsic matcher

In this section, we define a first-order matcher for standard regular expressions. This is a defunctionalization of Harper's algorithm, though we will describe it from first principles. The idea is to generalize from matching a string s against a regular expression r by adding an additional stack of regular expressions k that need to be matched against the suffix of s if some prefix of s matches r. We represent the stack by a list, and say that a string is in the language of a stack if it splits into strings in the language of each stack element:

```
_∈L^k_ : List Char → List StdRegExp → Set
s ∈L^k [] = s ≡ []
s ∈L^k (r :: rs) =
  Σ (List Char × List Char)
    (λ { (p , s') → (p ++ s' ≡ s) × (p ∈L^s r) × (s' ∈L^k rs) })
```

If the stack is empty, the string also has to be empty. If the stack has a head, then a prefix of the string should match the head of the list and the rest of the string should match with the rest of the list.

3.1 Definition

The soundness part of informal description of the matcher given above translates into the following dependent type/specification:

```
match : (r : StdRegExp) (s : List Char) (k : List StdRegExp)
        → Maybe (Σ (List Char × List Char)
                   (λ {(p, s') → (p ++ s' ≡ s) × (p ∈L^s r) × s' ∈L^k k}))
```

This says that the matcher takes a regular expression r, a string s and a stack k, and, if matching succeeds, returns a splitting (p ++ s' ≡ s), a derivation that p is in the language of r and a derivation that s' is in the language of the stack k. This specifies the soundness of successful matching, and has computational content that will let us extract matching strings, but leaves open the possibility that the matcher fails incorrectly (for example, the constantly nothing function has this type)—this will be addressed in the completeness proof below.

The complete code is in Figure 1. Agda can verify that this function terminates by induction on first the string s, and then the regular expression r. We now discuss the code case by case.

3.1.1 Base cases

First, the empty language does not accept any string:

```
match ∅^s s k = fail
```

For character literals

```
match (Lit^s c) [] k = fail
match (Lit^s c) (x :: xs) k =
  (isEqual x c) >>=
    (λ p → map (λ pf → ((([c], xs), cong (λ x → x :: xs) (sym p), refl, pf)))
      (match-helper k xs))
```

in the first case, if we are trying to match an empty list with a regular expression that requires a character, the matcher fails. In the next case, the isEqual x c call has type Maybe (x ≡ c)—i.e. it optionally shows that x is equal to c. Thus, by the monad bind, when x is not c, the matcher fails, and when x is c, we try to match the stack k against the suffix xs using match-helper.

The function match-helper is mutually recursive with our matcher and is defined as follows:

```
match-helper : (k : List StdRegExp) → (s : List Char) → Maybe (s ∈L^k k)
match-helper [] [] = return refl
match-helper [] (x :: s) = fail
match-helper (r :: k') s = match r s k'
```

It succeeds when matching the empty string against the empty stack, fails when matching a non-empty string against the empty stack, and otherwise refers back to match. Relative to Harper's algorithm, this is the application function for the defunctionalized continuation.

Returning to the character literal case, if the first character in our list matches the character literal c, then we call match-helper on the continuation and the rest of the list, which will produce a derivation of s ∈L^k k if the rest of the list indeed matches the rest of the StdRegExps in k. The remainder of the code packages this as a pair showing that therefore the list x :: xs splits as [x] ∈L^s (Lit c) and xs ∈L^k k.

Agda's termination checker is able to verify that, for the call from match to match-helper and back to match, the string x :: xs becomes xs, and is therefore smaller, which justifies termination in this case.

3.1.2 Concatenation

```
match (r₁ ·^s r₂) s k =
  map (reassociate-left {R = _·^s_} (λ inL inL' → _, refl, inL, inL'))
    (match r₁ s (r₂ :: k))
```

In the case for concatenation $r_1 \cdot^s r_2$, we match against r_1 first, and add r_2 to the stack k. Calling match r_1 s (r_2 :: k) will give us a split xs ++ ys ≡ s and derivations of xs ∈L^s r_1 and ys ∈L^k (r_2 :: k). If we unpack the second derivation (using the definition of ∈L^k and the fact that our continuation list contains at least one element, r_2), we will have another split as ++ bs ≡ ys and derivations as ∈L^s r_2 and bs ∈L^k k. The helper function reassociate-left states that if we have such a situation, we can reassociate it to show that xs ++ as matching the entire regular expression $r_1 \cdot^s r_2$. Because we will want to do similar reasoning in the Kleene plus case below, we use a higher-order function that says that if R is a binary operation on regular expressions that respects splitting, then given the first kind of splitting, we can produce the second:

```
reassociate-left : ∀ {r₁ r₂ s k} {R : StdRegExp → StdRegExp → StdRegExp}
  → (f : ∀ {xs as} → xs ∈L^s r₁ → as ∈L^s r₂ → ((xs ++ as) ∈L^s R r₁ r₂))
```

```
  → Σ _ (λ {(xs, ys) → (xs ++ ys ≡ s) × xs ∈L^s r₁
     × Σ _ (λ {(as, bs) → (as ++ bs ≡ ys) × as ∈L^s r₂ × bs ∈L^k k})})
  → (Σ _ (λ {(p, s') → (p ++ s' ≡ s) × (p ∈L^s R r₁ r₂) × s' ∈L^k k}))
```

3.1.3 Alternation

```
match (r₁ ⊕^s r₂) s k =
  (map (change-∈L inj₁) (match r₁ s k)) ||
  (map (change-∈L inj₂) (match r₂ s k))
```

In the alternation case, we match the string with r_1, and if that fails match with r_2 (recall that || handles failure of its first disjunct by trying the second). If the call match r_1 s k succeeds, it will produce a triple splitting s as p ++ s', with a derivation of $p \in L^s\ r_1$ and $s' \in L^k\ k$. However the return type for the alternation case should contain a derivation of type $p \in L^s\ (r_1 \oplus^s r_2)$, so we use the helper function change-∈L, which applies a function to this position of the result triple, to make the appropriate modification:

```
change-∈L : {a b d : List Char → Set} {c : List Char → List Char → Set}
  → (∀ {s} → a s → b s)
  → (Σ _ (λ {(p, s') → (c p s') × (a p) × (d s')}))
  → (Σ _ (λ {(p, s') → (c p s') × (b p) × (d s')}))
change-∈L f (x, eq, inL, rest) = (x, eq, f inL, rest)
```

We use change-∈L to apply inj_1 or inj_2 to the derivation, depending on which part of the alternation successfully matched the string.

3.1.4 Kleene plus

```
match (r +^s) s k =
  (map (change-∈L S+) (match r s k)) ||
  (map (reassociate-left {R = λ r _ → r +^s} (λ inL inL' → C+ refl inL inL'))
    (match r s ((r +^s) :: k)))
```

In the Kleene plus case, we first try to match s with just r, and if that succeeds we apply change-∈L S+ to the derivation since we matched from the single r case. If this fails, then similar to the $\cdot^s$ case, we try to match a prefix of the string to r and then the suffix that follows with the continuation which now includes $r\ ^{+s}$. Just like in the $\cdot^s$ case, we use reassociate-left in order to get that our splitting of s matches the entire starting r.

```
mutual
  match : (r : StdRegExp) (s : List Char) (k : List StdRegExp)
    → Maybe (Σ (List Char × List Char)
                 (λ {(p, s') → (p ++ s' ≡ s) × (p ∈L^s r) × s' ∈L^k k}))
  match ∅^s s k = fail
  match (Lit^s c) [] k = fail
  match (Lit^s c) (x :: xs) k =
    (isEqual x c) >>=
      (λ p → map (λ pf → ((([c], xs), cong (λ x → x :: xs) (sym p), refl, pf)))
        (match-helper k xs)
  match (r₁ ·^s r₂) s k =
    map (reassociate-left {R = _·^s_} (λ inL inL' → _, refl, inL, inL'))
      (match r₁ s (r₂ :: k))
  match (r₁ ⊕^s r₂) s k =
    (map (change-∈L inj₁) (match r₁ s k)) ||
    (map (change-∈L inj₂) (match r₂ s k))
  match (r +^s) s k =
    (map (change-∈L S+) (match r s k)) ||
    (map (reassociate-left {R = λ r _ → r +^s} (λ inL inL' → C+ refl inL inL'))
      (match r s ((r +^s) :: k)))
  match-helper : (k : List StdRegExp) → (s : List Char) → Maybe (s ∈L^k k)
  match-helper [] [] = return refl
  match-helper [] (x :: s) = fail
  match-helper (r :: k') s = match r s k'
```

Figure 1: Defunctionalized intrinsic matcher

3.1.5 Acceptance

From the outside, we can call the generalized matcher with the empty stack to check membership:

```
accepts^s-intrinsic : (r : StdRegExp) → (s : List Char) → Maybe (s ∈L^s r)
accepts^s-intrinsic r s = map ∈L-empty-stack (match r s [])
```

When the match function succeeds on an empty stack, the suffix is in the language of an empty stack and is therefore an empty string, so we use the following lemma to change the result of the function call match r s [] into a derivation over the entire string s.

```
∈L-empty-stack : {r : StdRegExp} {s : List Char}
  → Σ _ (λ {(p, s') → (p ++ s' ≡ s) × (p ∈L^s r) × (s' ≡ [])})
  → s ∈L^s r
```

3.2 Completeness

Though the above matcher is intrinsically sound, it is not intrinsically complete—for example, the function that always fails has the above type. One way to resolve this would be to write a matcher that is intrinsically both sound and complete — ignoring the stack, given s and r, we would like to know (s ∈L^s r) ⊎ ¬ (s ∈L^s r), a type that expresses decidability of matching. However, while there is an efficiency reason to intrinsically compute the the derivation of s ∈L^s r—we will use it to extract matching strings in Section 5.1—there is no computational content to ¬ (s ∈L^s r). Because of this, and because it keeps the matcher code itself simpler, we choose to make completeness extrinsic. In full, completeness says that if we have r, s, k and a split of p ++ s' ≡ s such that there are derivations of type p ∈L^s r and s' ∈L^k k, then we know that our match function does not fail:

```
match-completeness : (r : StdRegExp) (s : List Char) (k : List StdRegExp)
                     → Σ _ (λ {(p, s') → (p ++ s' ≡ s) × (p ∈L^s r) × (s' ∈L^k k)})
                     → isJust (match r s k)
```

That is, when there is a way for the matcher to succeed, it does. We cannot make a stronger claim and say that it returns the same derivation that is given as input, because as we showed before, there can be different derivations for the same s and r, and the given one may not be the one the matcher finds.

The full proof is in the companion code. The base cases $\varnothing^{\mathsf{s}}$ and $\mathsf{Lit}^{\mathsf{s}}$ are easy. Since the Kleene plus case captures the essence of both concatenation and alternation cases, we will explain only this case. The proof proceeds by cases, depending on how the given derivation of $\mathsf{p} \in \mathsf{L}\ (\mathsf{r}^{+})$ was constructed. For the first,

```
match-completeness (r +s) s k ((xs, ys), eq, S+ inL, rest)
  with match r s k | match-completeness r s k ((xs, ys), eq, inL, rest)
... | nothing | ()
... | just _ | _ = tt
```

if the given derivation of $\mathsf{xs} \in \mathsf{L}\ (\mathsf{r}^{+})$ was by the constructor $\mathsf{S+}$, then sitting under the constructor is a derivation of $\mathsf{inL} : \mathsf{xs} \in \mathsf{L}^{\mathsf{s}}\ \mathsf{r}$, and the result follows from the inductive hypothesis on D.

For the second, where the derivation was constructed by $\mathsf{C+}$,

```
match-completeness (r +s) ∘ ((s₁ ++ s₂) ++ ys) k
  ((._, ys), refl, C+ {._} {s₁} {s₂} refl inL1 inL2, rest)
  with match r ((s₁ ++ s₂) ++ ys) k
... | just _ = tt
... | nothing
  with match r ((s₁ ++ s₂) ++ ys) ((r +s) :: k)
     | match-completeness r ((s₁ ++ s₂) ++ ys) ((r +s) :: k)
       (_, append-assoc s₁ s₂ ys, inL1, (_, ys), refl, inL2, rest)
... | nothing | ()
... | just _ | _ = tt
```

we already had a split $\mathsf{xs} +\!\!+ \mathsf{ys} \equiv \mathsf{s}$, so we can replace s with $\mathsf{xs} +\!\!+ \mathsf{ys}$ by pattern matching on the equality proof. The constructor $\mathsf{C+}$ gives us another split $\mathsf{s_1} +\!\!+ \mathsf{s_2} \equiv \mathsf{xs}$, so we can also replace xs with $\mathsf{s_1} +\!\!+ \mathsf{s_2}$. This means now we have to show the goal for $(\mathsf{s_1} +\!\!+ \mathsf{s_2}) +\!\!+ \mathsf{ys}$ instead of s. The definition of the Kleene plus case uses $\_||\_$, which has to try and fail the first case to return the second case. To verify completeness, we first check whether $\mathsf{match\ r}\ ((\mathsf{s_1} +\!\!+ \mathsf{s_2}) +\!\!+ \mathsf{ys})\ \mathsf{k}$ succeeds or fails. If the call succeeds, then we satisfy the first case of $\_||\_$, so the matcher succeeds. If the call fails, then we checker whether the second disjunct fails or succeeds. If it fails, then we obtain a contradiction by the inductive hypothesis, which shows that the recursive call should have succeeded because $\mathsf{s_1}$ matches r and $\mathsf{s_2} +\!\!+ \mathsf{ys}$ matches the continuation (using the associative property of appending lists to show that it is the same string). If it succeeds, then the matcher succeeds, so we have the result.

As a corollary, we get completeness of accepts^s-intrinsic:

```
accepts^s-intrinsic-completeness : (r : StdRegExp) (s : List Char)
                                 → s ∈L^s r
                                 → isJust (accepts^s-intrinsic r s)
```

4 Higher-order intrinsic matcher

In this section, we show that the above intrinsic verification of the first-order matcher scales to a higher-order matcher, written using continuation-passing, which is more similar to Harper's original code. We will use this to explain why the above matcher is a defunctionalization, and why the termination reasoning is more difficult in the higher-order case. To make termination evident to Agda, we will need to use an explicit termination metric that corresponds to well-founded induction on strings/lists. This is represented in Agda by an iterated inductive definition RecursionPermission xs. Visually, you can think of RecursionPermission ys as a tree, where a node for ys has subtrees for each strict suffix of ys. Each of these subtrees is judged smaller by the termination checker, and therefore we will be allowed to recur on any suffix of ys. Such a tree type is defined by the following datatype, which has a higher-order constructor argument:

```
data RecursionPermission {A : Set} : List A → Set where
  CanRec : {ys : List A}
    → ((xs : List A) → Suffix xs ys → RecursionPermission xs)
    → RecursionPermission ys
```

4.1 Definition

The higher-order intrinsic matcher has the following specification:

```
match : (C : Set) (r : StdRegExp) (s : List Char)
      → (k : ∀ {p s'} → p ++ s' ≡ s → p ∈L^s r → Maybe C)
      → RecursionPermission s
      → Maybe C
```

The type variable C stands for the output derivation computed by the matcher on success. Just as Harper's algorithm returns a bool and uses both the continuation and the language's control stack (i.e. it is not fully in CPS),

here both the continuation and the matcher return an option, but the success data can been chosen arbitrarily. In Harper's algorithm, the continuation takes a string that corresponds to s' in the above type. The additional arguments provided here, which are important for justifying termination, say that in a call to match r s k, the domain of the continuation is suffixes s' of s by a prefix that is in the language of r. The final argument is of type RecursionPermission s, and allows recursive calls on strict suffixes of s. The termination measure for this function is lexicographic in first the regular expression r and then the recursion permission tree. The complete code is in Figure 2.

4.1.1 Base cases

The case for the empty regexp fails:

```
match C ∅ˢ s k perm = fail
```

The cases for character literals

```
match C (Litˢ c) [] k perm = fail
match C (Litˢ c) (x :: xs) k perm =
  (isEqual x c) >>= (λ p → k {[x]} {xs} refl (cong (λ q → [q]) p))
```

are as above, except that where we called match-helper to activate the stack k, here we call k itself. The packaging of the result of match-helper (the map in the above code) now happens as *input* to k, because to call k we must show that x :: xs splits as something in the language of Litˢ c and some suffix.

4.1.2 Concatenation

```
match C (r₁ ·ˢ r₂) s k (CanRec f) =
  match C r₁ s
    (λ {p} {s'} eq inL →
      match C r₂ s' (λ {p'} {s''} eq' inL' →
        k {p ++ p'} {s''} (replace-right p s' p' s'' s eq' eq) ((p, p'), refl, inL, inL'))
        (f s' (suffix-after-∈Lˢ eq inL))) (CanRec f)
```

In the concatenation case of the defunctionalized version, we added r_2 to the stack of continuations to be matched later. In the higher-order version, extending the stack with r_2 corresponds to constructing a new continuation function which matches r_2 against the suffix that results from matching r_1

against a prefix—which is exactly what "applying" the stack in match-helper did in the defunctionalized version. The massaging that happens after the recursive call above (the reassociate-left) here happens in the new continuation, which repackages the given derivations inL : p ∈L^s r_1 and inL' : p' ∈L^s r_2 and the given splittings eq : p ++ s' ≡ s and eq' : p' ++ s'' ≡ s' as a splitting (p ++ p') ++ s'' ≡ s and a derivation that p ++ p' ∈ ∈L^s (r_1 ·^{s} r_2). The two recursive calls pass the termination checker because the regular expressions r_1 and r_2 get smaller in each case. To make the inner recursive call, it is necessary to supply a recursion permission for s', i.e. to allow recursive calls on s', and to do this it suffices to show that s' is a suffix of s. The suffix-after-∈L^s lemma

```
suffix-after-∈L^s : ∀ {p s' s r} → (p ++ s' ≡ s) → (p ∈L^s r) → Suffix s' s
```

does this: s' is a non-strict suffix of s by the equality, and because the prefix p is in the language of a *standard* regular expression r and therefore is not empty, it is a strict suffix. (In this case, it would also be sufficient to observe that s' is a non-strict suffix of s, because we do not need the string and recursion permission to get smaller to justify termination, but the argument we just gave will also be used in the Kleene plus case.)

4.1.3 Alternation

```
match C (r₁ ⊕^s r₂) s k perm =
  match C r₁ s (λ eq inL → k eq (inj₁ inL)) perm ||
  match C r₂ s (λ eq inL → k eq (inj₂ inL)) perm
```

The alternation case is similar to the defunctionalized version, except instead of massaging the derivations after the fact with map change-∈L, we modify them before passing them to the continuation.

4.1.4 Kleene plus

```
match C (r +^s) s k (CanRec f) =
  match C r s (λ eq inL → k eq (S+ inL)) (CanRec f) ||
  match C r s (λ {p} {s'} eq inL →
    match C (r +^s) s' (λ {p'} {s''} eq' inL' →
      k (replace-right p s' p' s'' s eq' eq) (C+ refl inL inL'))
      (f s' (suffix-after-∈L^s eq inL))) (CanRec f)
```

The structure of the Kleene plus case is similar to the defunctionalized version, except the continuations are modified analogously to the concatenation and alternation cases. The first recursive call terminates because r gets smaller. For the second recursive call, (r $^{+s}$) stays the same, so it is essential that s' is a *strict* suffix of s, and that the recursion permission tree gets smaller. As in the alternation case, s' is a non-strict suffix of s by the equality eq : p ++ s' ≡ s, and because the prefix p is in the language of a *standard* regular expression r and therefore is not empty, s' is a strict suffix of s by the suffix-after-∈L^{s} lemma. Therefore, applying f to s' and this fact selects a smaller recursion permission subtree, justifying termination.

Termination is trickier for the higher-order matcher than for the defunctionalized matcher because, here, we make the recursive call on r $^{+s}$ in the continuation constructed in the r $^{+s}$ case, so we must argue that *whenever this continuation is applied*, it will be applied to a smaller string. In the defunctionalized matcher, this recursive call is made in match-helper (the apply function for the defunctionalized continuation), which is called from the character literal case, at which point it is syntactically clear that the recursive call *is being made* on a smaller string.

4.1.5 Acceptance

Overall, we can define

```
accepts^s-intrinsic : (r : StdRegExp) → (s : List Char) → Maybe (s ∈L^s r)
accepts^s-intrinsic r s = match _ r s empty-continuation (well-founded s)
```

by choosing an appropriate initial continuation, and by constructing a recursion permission for s (which exists because string suffix is a well-founded relation). The initial continuation

```
empty-continuation : ∀ {p s' s r} → (p ++ s' ≡ s) → (p ∈L^s r) → Maybe (s ∈L^s r)
```

corresponds to the logic for the empty stack [] in match-helper in the defunctionalized version. This function takes a splitting of a string s as p ++ s', as well as a proof that its first part p is in the language of r. It returns either nothing if s' is not empty, or just a witness that (s ∈L^{s} r) if s' is empty, and therefore s ≡ p'.

4.2 Completeness

Completeness is similar to above, and says that the matcher succeeds whenever it should:

```
match : (C : Set) (r : StdRegExp) (s : List Char)
        (k : ∀ {p s'} → p ++ s' ≡ s → p ∈L^s r → Maybe C)
        → RecursionPermission s
        → Maybe C
match C ∅^s s k perm = fail
match C (Lit^s c) [] k perm = fail
match C (Lit^s c) (x :: xs) k perm =
  (isEqual x c) >>= (λ p → k {[x]} {xs} refl (cong (λ q → [q]) p))
match C (r₁ ·^s r₂) s k (CanRec f) =
  match C r₁ s
    (λ {p} {s'} eq inL →
      match C r₂ s' (λ {p'} {s''} eq' inL' →
        k {p ++ p'} {s''} (replace-right p s' p' s'' s eq' eq) ((p, p'), refl, inL, inL'))
        (f _ (suffix-after-∈L^s eq inL))) (CanRec f)
match C (r₁ ⊕^s r₂) s k perm =
  match C r₁ s (λ eq inL → k eq (inj₁ inL)) perm ||
  match C r₂ s (λ eq inL → k eq (inj₂ inL)) perm
match C (r ⁺^s) s k (CanRec f) =
  match C r s (λ eq inL → k eq (S+ inL)) (CanRec f) ||
  match C r s (λ {p} {s'} eq inL →
    match C (r ⁺^s) s' (λ {p'} {s''} eq' inL' →
      k (replace-right p s' p' s'' s eq' eq) (C+ refl inL inL'))
      (f _ (suffix-after-∈L^s eq inL))) (CanRec f)
```

Figure 2: Complete definition of the match function for the higher-order intrinsic matcher.

```
match-completeness : (C : Set) (r : StdRegExp) (s : List Char)
  → (k : ∀ {p s'} → p ++ s' ≡ s → p ∈L^s r → Maybe C)
  → (perm : RecursionPermission s)
  → Σ _ (λ {(p , s') → Σ _ (λ eq → Σ _ (λ inL → isJust (k {p} {s'} eq inL)))})
  → isJust (match C r s k perm)
```

The type can be read as follows: Suppose we have C, r, s, k and perm. Suppose there exists a split of p ++ s' ≡ s such that there exists a derivation of type p ∈L^s r such that the continuation called with those arguments does not return nothing. Then we have to show that the match function does not fail. Like above, we cannot make a stronger claim and say that the calls to the continuation and the match function return the same derivations, because there can be more than one derivation of a string matching a regexp. The proof is in the companion code, and follows the same pattern as the proof of match-completeness for the defunctionalized version.

5 Matching non-standard regular expressions

Both of the above matchers work on syntactically standard regular expressions, which is used to show termination. We now show that this suffices to define a matcher for non-standard regular expressions, which we represent by a type RegExp:

```
data RegExp : Set where
  ∅ : RegExp
  ε : RegExp
  Lit : Char → RegExp
  _·_ : RegExp → RegExp → RegExp
  _⊕_ : RegExp → RegExp → RegExp
  _* : RegExp → RegExp
  G : RegExp → RegExp
```

∅ matches the empty language, ε matches the empty string, Lit is character literals, _·_ is concatenation, _⊕_ is alternation, _*_ is Kleene star, and G is for reporting matching strings.

```
mutual
  _∈L_ : List Char → RegExp → Set
  _ ∈L ∅ = ⊥
  s ∈L ε = s ≡ []
  s ∈L (Lit c) = s ≡ c :: []
```

```
s ∈L (r₁ ⊕ r₂)  =  (s ∈L r₁) ⊎ (s ∈L r₂)
s ∈L (r₁ · r₂)  =
  Σ (List Char × List Char) (λ {(p , q) → (p ++ q ≡ s) × (p ∈L r₁) × (q ∈L r₂)})
s ∈L (r *)  =  s ∈L^x r
s ∈L (G r)  =  s ∈L r

data _∈L^x_ : List Char → RegExp → Set where
  Ex : ∀ {s r} → s ≡ [] → s ∈L^x r
  Cx : ∀ {s s₁ s₂ r} → s₁ ++ s₂ ≡ s → s₁ ∈L r → s₂ ∈L^x r → s ∈L^x r
```

The definition of the language of these regular expressions is similar to above, but with an ε case that requires an empty string, and a base case for $s \in L^x r$ that allows the empty string; note that G does not change the language.

Next, we define a translation from regexps to syntactically standard regexps. The translation uses a helper function that checks if a regular expression accepts the empty string. Instead of giving this function the type $\mathsf{RegExp} \to \mathsf{Bool}$, we give it a more informative type stating that it decides whether the empty string is in the language of its input:

```
δ' : (r : RegExp) → ([] ∈L r) ⊎ (¬ ([] ∈L r))
```

Using δ', we can easily define $\delta : \mathsf{RegExp} \to \mathsf{Bool}$ by forgetting the extra information.

The specification for standardization, which we prove below, is that

$$(\forall s)\ \big[s \in L(r) \iff [(\delta(r) = true \wedge s = []) \vee s \in L(\mathsf{standardize}(r))]\ \big]$$

That is, s is in the language of r if and only if either and r accepts the empty string and the string is empty , or s is in the language of the standardized version of r.

We standardize as follows:

```
standardize : RegExp → StdRegExp
standardize ∅  =  ∅ˢ
standardize ε  =  ∅ˢ
standardize (Lit x)  =  Litˢ x
standardize (r₁ · r₂) with standardize r₁ | standardize r₂ | δ r₁ | δ r₂
... | r₁' | r₂' | false | false  =  r₁' ·ˢ r₂'
... | r₁' | r₂' | false | true  =  r₁' ⊕ˢ (r₁' ·ˢ r₂')
... | r₁' | r₂' | true | false  =  r₂' ⊕ˢ (r₁' ·ˢ r₂')
... | r₁' | r₂' | true | true  =  r₁' ⊕ˢ r₂' ⊕ˢ (r₁' ·ˢ r₂')
standardize (r₁ ⊕ r₂)  =  standardize r₁ ⊕ˢ standardize r₂
```

```
standardize (r *) = (standardize r) ⁺ˢ
standardize (G r) = standardize r
```

The empty string language ε becomes the empty set $\varnothing^s$ and Kleene star becomes Kleene plus, because the emptiness checking is performed outside matching the standardized regexp. For the concatenation case, we write r_1' and r_2' for the standardizations of r_1 and r_2. Because standardize r will not accept the empty string even when r does, it is necessary to check r_1' and r_2' by themselves in the case where the other one accepts the empty string, because otherwise we would miss strings that rely on one component but not the other being empty.

Our definition of the concatenation case is a bit different than Harper's, where δ returns not a boolean, but a regexp $\varnothing$ (if r does not accept the empty string) or ε (if it does), and the clause is as follows:

```
standardize (r₁ · r₂) = ((δ r₁) ·ˢ (standardize r₂)) ⊕ˢ
                        ((standardize r₁) ·ˢ (δ r₂)) ⊕ˢ
                        ((standardize r₁) ·ˢ (standardize r₂))
```

This definition is equivalent to above, using the equivalences that for any r, $\varnothing \cdot^s r = \varnothing = r \cdot^s \varnothing$ and $\varepsilon \cdot^s r = r = r \cdot^s \varepsilon$. For example, when $\delta\ r_1$ is true, Harper's translation gives a $\varepsilon \cdot^s$ (standardize r_2) summand, which is standard but *not* syntactically standard, but we can simplify it to standardize r_2. When $\delta\ r_1$ is false, Harper's translation gives an $\varnothing \cdot^s$ (standardize r_2) summand, which drops out.

This definition of standardization satisfies the above correctness theorem; we have proved

```
∈L-soundness : (s : List Char) (r : RegExp)
               → ((δ r ≡ true) × (s ≡ [])) ⊎ (s ∈Lˢ (standardize r))
               → s ∈L r
∈L-completeness : (s : List Char) (r : RegExp)
                  → s ∈L r
                  → ((δ r ≡ true) × (s ≡ [])) ⊎ (s ∈Lˢ (standardize r))
```

Now that we have a verified standardize function, we can define _accepts_ as follows, where accepts^s-intrinsic can be either of the above matchers:

```
accepts-intrinsic : (r : RegExp) → (s : List Char) → Maybe (s ∈L r)
accepts-intrinsic r s with δ' r
accepts-intrinsic r [] | inj₁ x = just x
accepts-intrinsic r s | _ = map (∈L-soundness s r ∘ inj₂) (acceptsˢ-intrinsic (standardize r) s)
```

If r accepts the empty string, we return true if xs is empty or the standardization of r accepts xs. If r does not accept the empty string, then we only have the latter option. In that case, we call acceptsˢ-intrinsic to get an optional derivation of the type s ∈Lˢ (standardize r) and use that on ∈L-soundness to get an optional derivation of the type s ∈L r.

As usual, we have proved completeness extrinsically:

```
correct-completeness : (r : RegExp) (s : List Char)
                     → s ∈L r
                     → isJust (r accepts s)
```

Finally, we have proved decidability of matching as a corollary of soundness and completeness:

```
decidability : (r : RegExp) (s : List Char) → (s ∈L r) ⊎ (¬ (s ∈L r))
```

5.1 Capturing groups

The "capturing group" constructor G is intended to allow the user to specify parts of a regular expression whose matching strings should be extracted and reported. For example, if our regular expression checks if a string is a valid e-mail address, we might use this to parse the username and domains. If we have a regular expression alphanumeric : RegExp that accepts a single alphanumeric character (this can be defined in the above language as a big ⊕ of character literals), we can define a (naïve) regular expression for e-mail addresses such as

```
e-mail : RegExp
e-mail = G (alphanumeric *) · Lit '@' · G (alphanumeric *) · Lit '.' · G (alphanumeric *)
```

Now, if we match the string "jdoe@wesleyan.edu" with e-mail, we should extract and report "jdoe", "wesleyan" and "edu", because each of those substrings matched a sub-regexp that was marked with G.

This extraction can be computed from the derivation of s ∈L r, which provides a parse tree that says which substring of xs is matched by which part of r. Thus, to report the groups, we do an in-order traversal of the regexp and derivation tree, and collect the strings matching a capturing group to a list. The function to do this can be defined as follows:

```
extract : {r : RegExp} → {xs : List Char} → xs ∈L r → List (List Char)
extract {∅} ()
```

```
extract {ε} refl = []
extract {Lit x} refl = []
extract {r₁ · r₂} ((as, bs), eq, a, b) = extract {r₁} {as} a ++ extract {r₂} {bs} b
extract {r₁ ⊕ r₂} (inj₁ x) = extract {r₁} x
extract {r₁ ⊕ r₂} (inj₂ y) = extract {r₂} y
extract {r *} (Ex refl) = []
extract {r *} (Cx {s} {s₁} {s₂} x x₁ inL) = extract {r} x₁ ++ extract {r *} inL
extract {G r} {xs} inL = xs :: extract {r} inL
```

Base cases $\varnothing$, ε, Lit will return an empty list because if they are captured by a group, the substring is already added to the list in the previous calls to extract. In concatenation, we make two recursive calls and append the results because r_1 and r_2 match different substrings and they may have different capturing groups inside them. In alternation, the entire string matches either r_1 or r_2, so we make one recursive call to the one it matches. The Kleene star case follows the same principles.

Combining this with our intrinsic matcher, we can define an overall function

```
groups : (r : RegExp) (s : List Char) → Maybe (List (List Char))
groups r s = map extract (accepts-intrinsic r s)
```

6 Conclusion

We have studied three variations on Harper's algorithm for regular expression matching, which were inspired by programming and verifying this algorithm using dependent types: defunctionalizing the matcher allows Agda to see termination without an explicit metric, and provides an alternative to Yi's first-order matcher based on state machines; intrinsically verifying soundness allows extracting matching strings; and a syntactic definition of standard regular expressions simplifies the staging of the development. We believe that these variations provide a nice illustration of the benefits of thinking in a dependently typed language, and that they have some pedagogical value for teaching this material in courses on dependently typed programming—or, by porting the observations back to simply-typed languages, on introductory programming.

References

Danielsson, Nils Anders. (2010). Total parser combinators. *Pages 285–296 of: Proceedings of the 15th acm sigplan international conference on functional programming*. ICFP '10. New York, NY, USA: ACM.

Firsov, Denis, & Uustalu, Tarmo. (2013). Certified parsing of regular languages. *Pages 98–113 of: Proceedings of the third international conference on certified programs and proofs - volume 8307*. New York, NY, USA: Springer-Verlag New York, Inc.

Harper, Robert. (1999). Functional pearl. proof-directed debugging – corrigendum. *Journal of functional programming*, **9**(4), 463–469.

Norell, Ulf. (2007). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

Reynolds, John C. (1972). Definitional interpreters for higher-order programming languages. *Pages 717–740 of: Proceedings of the acm annual conference - volume 2*. ACM '72. ACM.

Ridge, Tom. (2011). Simple, functional, sound and complete parsing for all context-free grammars. *Pages 103–118 of: Proceedings of the first international conference on certified programs and proofs*. CPP'11. Berlin, Heidelberg: Springer-Verlag.

Yi, Kwangkeun. (2006). Educational pearl: "proof-directed debugging" revisited for a first-order version. *Journal of functional programming*, **16**(6), 663–670.