JFP **31**, e12, 38 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press
doi:10.1017/S0956796820000295

Proof-directed program transformation: A functional account of efficient regular expression matching

ANDRZEJ FILINSKI

Department of Computer Science, University of Copenhagen, Denmark
(*e-mail:* andrzej@di.ku.dk)

Abstract

We show how to systematically derive an efficient regular expression (regex) matcher using a variety of program transformation techniques, but very little specialized formal language and automata theory. Starting from the standard specification of the set-theoretic semantics of regular expressions, we proceed via a continuation-based backtracking matcher, to a classical, table-driven state machine. All steps of the development are supported by self-contained (and machine-verified) equational correctness proofs.

1 Introduction

1.1 Background

The theory of regular languages is a staple of undergraduate theoretical computer science. It introduces a number of important concepts and methods, such as the notions of regular expressions and finite-state automata (both nondeterministic and deterministic, NFAs/DFAs), and the Thompson, powerset, and partition-refinement constructions. It also serves as a natural context for discussing issues of computability and time/space complexity, including the trade-offs between various representations of a regular language.

However, this theory is typically presented as a semi-isolated body of knowledge, with a collection of bespoke theorems, and specialized algorithms and data structures (Berry & Sethi, 1986; Yi, 2006). In particular, though the theory and practice of *formal* languages evidently forms a cornerstone of *programming* language design and implementation (including for aspects unrelated to lexing/parsing, such as various forms of static analysis), it has been much less influenced by the corresponding aspects of programming languages in general and functional programming in particular. With a few notable exceptions (such as in the textbook by Jones (1997)), the study of semantics, analysis, transformation, and verification of functional programs is often perceived as having little direct relevance for the algorithmic aspects of formal languages.

It was therefore a surprising development that, as discovered by Consel & Danvy (1989), the workhorse Knuth-Morris-Pratt (KMP) algorithm (Knuth *et al.*, 1977) for pattern matching in strings can be obtained as a simple binding-time improvement of a naive,

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

```
datatype re =
    Char of char
  | Eps
  | Seq of re * re
  | Void
  | Alt of re * re
  | Star of re

type ans = bool

(* omatch : re -> (char list -> ans) -> char list -> ans *)
fun omatch (Char c) k [] = false
  | omatch (Char c) k (c' :: s') = c = c' andalso k s'
  | omatch Eps k s = k s
  | omatch (Seq (r1, r2)) k s = omatch r1 (fn s' => omatch r2 k s') s
  | omatch Void k s = false
  | omatch (Alt (r1, r2)) k s = omatch r1 k s orelse omatch r2 k s
  | omatch (r as Star r0) k s =
      k s orelse omatch r0 (fn s' => omatch r k s') s

(* omatchtop : re -> char list -> ans *)
fun omatchtop r s = omatch r (fn [] => true | _ => false) s
```

Fig. 1. Original (divergence-prone) matcher.

quadratic time, but purely functional matcher, followed by a mechanical specialization of the resulting program to a given pattern.

In this article, we will consider a conceptually similar – though not as fully mechanized – approach to the more complex problem of regular expression matching. We will reconstruct (near-)analogs of traditional regex-related constructions from almost first principles, by *systematic* (but not quite *automatic*) program transformations, guided by a few, suitably chosen invariants. As a side benefit, we aim to demonstrate that the framework of functional programming provides an excellent medium for studying and teaching the basic concepts underlying formal-language processing.

1.2 Overview of the constructions

As a starting point, we take the continuation-passing style (CPS) matcher in Figure 1. Except for minor syntactic differences, it is identical to the one considered by Harper (1999). (It is also very similar to the one sketched by Danvy & Filinski (1990), except that the latter encodes nondeterministic choices not by a fixed `orelse`, but by explicitly enumerating all solutions using a second layer of *failure continuations*, which can be further generalized to use any suitable monad (Filinski, 1999) to combine the successes. As we are concentrating on just binary acceptance/rejection here, though, we will not pursue these monadic connections further.)

The traditional syntax of regexes (including one denoting the empty set) is straightforwardly represented as the type `re`. Intuitively, a call `omatch` *r k s* tries to match a string from the language denoted by *r* at the start of *s* and invoke the continuation *k* on the remainder of the string; it accepts when both of these tasks succeed. When there is a choice to be

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

made, both alternatives are tried in sequence, and it is enough that either one succeeds. The initial continuation in `omatchtop` then merely checks that the entire string was consumed.

As it is easily seen, this matcher is prone to looping infinitely when the r_0 in a `Star` r_0 is *nullable*, that is, can accept the empty string. (This problem occurs regardless of the ordering of the disjuncts in `orelse`, since both will be tried if the match ultimately fails for some other reason.) One way of addressing this issue is simply to forbid iteration over nullable sub-regexes, by suitably preprocessing the original regex into one accepting the same language, but not containing the problematic subexpressions. This is the approach pursued by Harper and is in many ways very reasonable. The price, however, is both a moderately more complex overall correctness argument, and a potential blowup in the size of the regex. The latter does not necessarily entail a corresponding reduction in the worst-case performance (which is already not great), but it certainly complicates the complexity analysis.

As an alternative, we may build some kind of loop detection into the matcher itself. The key observation is that, in the second disjunct for `Star` r_0, we may safely require that matching the initial r_0 consumes at least one character from the string, since the case where it consumes none is already covered by the first disjunct. This can, in principle, be expressed quite simply, by replacing the relevant recursive equation with code like this:

```
| omatch (r as Star r0) k s =
    k s orelse
    omatch r0 (fn s' => length s' < length s andalso omatch r k s') s
```

(The calls to `length`, relatively expensive for a linked-list representation of strings, can of course be avoided by redundantly passing around the number of remaining characters, or the number consumed so far, together with `s`.)

However, while the above tweak does indeed correctly fix the divergence issue, it also inadvertently cements the fundamental performance problem of the backtracking matcher, by technically stepping outside of the conceptual machinery of finite-state string recognition, which we could otherwise exploit.

For note that, in the original matcher, every call to `omatch` with some subexpression r' of the top-level regex r occurs with a continuation k' that depends only on the position of r' within r, but *not* on the initial string. This means, in particular, that the set of possible continuations k in any run of the matcher contains only $O(|r|)$ different elements. Further, it is also very easy to see that every argument to a continuation k is some (not necessarily proper) suffix of the original string. Thus, with a simple memoization of all such calls, using at most $O(|r| \cdot |s|)$ space, we can avoid the massive work duplication inherent in the algorithm. (In fact, we will shortly see that we only need $O(|r|)$ such space.)

On the other hand, with the patch, some continuations would close not only over r and k, but also over fragments of the input string s (or at least their lengths). This means that there are now $O(|r| \cdot |s|)$ different continuations, and memoizing their values on all possible input-string suffixes – even ignoring the space cost – only guarantees $O(|r| \cdot |s|^2)$ time.

There is a more frugal way of detecting loops, however: instead of making the continuation in `Star` r_0 include a record of the exact number of characters in the original string s, we only need to check that s' is strictly shorter than s, but not by how much. We will see that, for this, it is enough to maintain a single boolean flag, indicating whether at least one

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

character has been consumed since the flag was last reset. This is the subject of Section 2, where we will prove such a termination-augmented matcher correct with respect to the usual semantics of (unrestricted) regexes.

In the following sections, we will then stepwise transform this matcher into a more efficient algorithm. In Section 3, we apply the idea of *program specialization* (Jones *et al.*, 1993) to statically translate a tree-structured regex into a *program* for a specialized abstract machine, given by a flat, $O(|r|)$-sized list of mutually recursive continuation specifications; this residual representation can then be run on the input string to compute the same result as the original, two-argument matcher. This transformation does not in itself materially affect the asymptotic complexity of the matcher, but it sets the stage for the next step.

In Section 4, we will see that the flat specification is amenable not only to ad hoc *memoization*, but that the results of all continuations can always be computed *eagerly* in a fixed, data-independent order, in the style of *dynamic programming*. In fact, the vector of continuation results for any string depends only on the head character of the string, and the vector of results for its tail; that is, we can compute all the continuation results for the initial string incrementally from the end, using only $O(|r|)$ working space and $O(|s| \cdot |r|)$ time. Effectively, we have a state machine, whose states are boolean vectors, and the computation specification tells us how to get from a state and an input character to the next state.

Finally, in Section 5, we exploit the idea of a *representation change* to obtain a (typically) more efficient representation of the state machine, by enumerating its reachable states and precomputing the transitions between them. This transformation works even for an infinite (or very large) input alphabet, because we only need to consider the machine's behavior on characters that are explicitly mentioned somewhere in the original regex. As with classical DFAs, this gives us an $O(|s|)$-time matcher, but at the cost of an up to $O(2^{|r|})$-sized transition table. We also briefly touch upon the topic of DFA minimization.

Toward the end of each section, we put our results into perspective with respect to related work and possible extensions, and in Section 6 we give some final conclusions and suggestions of topics for further exploration.

1.3 Methodology and notation

A number of the transformations, while informally plausible and natural, still rely on somewhat tricky invariants. A particular focus of this presentation is, therefore, on the formal theorems tying the various stages together in a rigorous way. While some of the proofs in the following are abridged, or omitted entirely, they have all been fully developed and machine-verified. Specifically, we have used the Twelf proof assistant (Pfenning & Schürmann, 1999), based on the Edinburgh Logical Framework (Harper *et al.*, 1993). It must be stressed, however, that the primary purpose of this formalization was simply to gain additional confidence in the correctness of the paper proofs – that is, to guard against clerical errors – as opposed to considering the formalization itself an object of study, or to provide a formal correctness certificate for the actual SML code. As such, while the raw Twelf code is provided for reference as an electronic supplement to this article, it is not necessarily easy to follow and is not intended as a formal artifact of this work.

We will generally not be overly concerned with the performance implications of superficial, low-level representation choices. For example, strings are uniformly represented as

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

(linked) lists of characters, where it would normally be more efficient (at least space-wise) to use pointers/indices into a fixed, linear buffer. Making such a change would be near-trivial but would clutter the presentation. Instead, where particularly relevant, we will just remark on how evident inefficiencies can be eliminated without material changes to the underlying algorithm. We express all code in (the pure fragment of) Standard ML (Milner *et al.*, 1997) to emphasize the continuity with Harper's work, but everything should also work, with only surface syntax adjustments, in Haskell or other functional languages.

Finally, we have a few notes on terminology. For closed SML expressions e and e', we use plain equality ($e = e'$) to express that both sides are defined and equal, and explicit Kleene equality ($e \simeq e'$) to also allow for both sides being undefined (because their evaluations diverge or raise exceptions). In proofs, when reasoning inductively about a recursively defined function f, and showing that $f(a) = b$ implies some relationship $P(a, b)$, we phrase the induction hypothesis in terms of $f(a') = b'$ implying $P(a', b')$ whenever the evaluation of $f(a)$ to b necessarily involves a subevaluation of $f(a')$ to b'. We refer to this as an "induction on the derivation of $f(a) = b$," without necessarily having any specific formal operational semantics of SML in mind (though the one in the Definition (Milner *et al.*, 1997) should do fine). Notation-wise, we write $|l|$ for the length of a list l, and $l_{[i]}$ for its i'th element (zero-based); and following SML, we use $l_1@l_2$ for list concatenation.

2 A continuation-based regex matcher

2.1 Ensuring termination

As noted in the Introduction, it is conceptually quite simple to modify the defining equation for `omatch (Star r0)` to ensure a "progress" property, that is, that every iteration consumes at least one input character. A somewhat harder challenge is to do so in a sufficiently light-handed manner, so as to retain the desirable property that the continuation at every point does not depend on the input string.

As with any continuation-passing program, it is straightforward to include a notion of single-threaded state-passing in the computation, by adding an extra parameter to the continuation. It turns out that we only need a single bit of state, simply keeping track of whether a character has been consumed since the start of the current iteration. Because all iterations are properly nested, this flag bit can be shared by all nesting levels. Informally, the flag is *set* after a successful `Char` c match, *cleared* before matching the body r_0 of an iteration `Star` r_0, and *passed along* unchanged by all other constructs. Also, when an iteration-body match succeeds without setting the flag, it is considered to have *failed*, and no further iterations are attempted. The initial state of the bit does not actually matter, but some of the results can be formulated slightly more uniformly if we set it to true at the beginning.

The augmented matcher can be seen in Figure 2. The one slight oddity is that we have introduced a single-use continuation `k'` in the last equation for `match`. This, of course, makes no discernible difference for correctness or performance of the matcher (indeed, a decent compiler will probably optimize it away entirely), but it makes the code satisfy a syntactic property that will prove important later.

Although we could in principle reason directly about the higher-order matcher, a number of constructions get a bit simpler to express and verify – especially in mechanized

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

```
(* match : re -> (bool -> char list -> ans) -> bool -> char list -> ans *)
fun match (Char c) k b [] = false
  | match (Char c) k b (c' :: s') = c = c' andalso k true s'
  | match Eps k b s = k b s
  | match (Seq (r1, r2)) k b s =
      match r1 (fn b' => fn s' => match r2 k b' s') b s
  | match Void k b s = false
  | match (Alt (r1, r2)) k b s = match r1 k b s orelse match r2 k b s
  | match (Star r0) k b s =
    let fun k' bf ss =
              match r0 (fn b' => fn s' =>
                          b' andalso match (Star r0) k b' s') bf ss
    in k b s orelse k' false s end

(* matchtop : re -> char list -> ans *)
fun matchtop r s = match r (fn b => fn [] => true | _ => false) true s
```

Fig. 2. Termination-augmented matcher (higher-order version).

```
datatype cont =
    CInit
  | CThen of re * cont
  | CStar of re * cont

(* fmatch : re -> cont -> bool -> char list -> ans *)
fun fmatch (Char c) k b [] = false
  | fmatch (Char c) k b (c' :: s') = c' = c andalso apply k true s'
  | fmatch Eps k b s = apply k b s
  | fmatch (Seq (r1, r2)) k b s = fmatch r1 (CThen (r2, k)) b s
  | fmatch Void k b s = false
  | fmatch (Alt (r1, r2)) k b s = fmatch r1 k b s orelse fmatch r2 k b s
  | fmatch (Star r0) k b s =
      apply k b s orelse apply (CThen (r0, CStar (r0, k))) false s
(* apply : cont -> bool -> char list -> ans *)
and apply CInit b [] = true
  | apply CInit b _ = false
  | apply (CThen (r, k)) b s = fmatch r k b s
  | apply (CStar (r, k)) b s = b andalso fmatch (Star r) k b s

(* fmatchtop : re -> char list -> ans *)
fun fmatchtop r s = fmatch r CInit true s
```

Fig. 3. Termination-augmented matcher (first-order version).

form – if we limit ourselves to first-order programs. Therefore, in Figure 3, we present the defunctionalized version (Reynolds, 1972) of the matcher. (In this "first-order" code, functions are still largely written in curried style, but all arguments are of nonfunctional type, and functions are never partially applied.)

The transformation simply consists of enumerating the three ways in which we construct new continuation arguments (in `matchtop`, and in `match` for `Seq` and `Star`) and replacing them with datatype constructors that keep track of the free variables of

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

each continuation body. The actual bodies are then collected in the function `apply`. Note that the constructor `CThen` (r, k), corresponding to `fn b => fn s => match` r k `b s`, is used both in the continuation built for `Seq` and the *outer* continuation for `Star`. The constructor `CStar` (r, k) represents almost the same continuation as `CThen` (`Star` r, k) but also includes a check of the flag.

The higher-order and first-order versions are very closely related:

Theorem 2.1 (Defunctionalization)**.** *For any r, k, b, and s, we have* `fmatch` r k b $s \simeq$ `match` r (`apply` k) b s*, and hence* `fmatchtop` r $s \simeq$ `matchtop` r s*.*

Proof First, we show the correspondence for `fmatch` by straightforward inductions on the derivations of each side. (These derivations are completely isomorphic, as the function closures in the higher-order version correspond 1-1 to the values of type `cont`.) The correspondence for `fmatchtop` then follows immediately. ■

For the correctness proof, let us start by establishing termination of the first-order matcher, by identifying a suitable metric on argument sizes in recursive calls. We first define, in the obvious way, the size $|r|$ of a regex r (which will also be useful for other purposes):

$$\begin{aligned} |\texttt{Char}\ c| &= 1 & \qquad |\texttt{Void}| &= 1 \\ |\texttt{Eps}| &= 1 & |\texttt{Alt}\ (r_1, r_2)| &= 1 + |r_1| + |r_2| \\ |\texttt{Seq}\ (r_1, r_2)| &= 1 + |r_1| + |r_2| & |\texttt{Star}\ r_0| &= 1 + |r_0| \end{aligned}$$

Less obviously, we also assign a *weight* to a pair k and b, $\|k\|_b$, by induction on k:

$$\begin{aligned} \|\texttt{CInit}\|_b &= 0 & \qquad \|\texttt{CStar}\ (r, k)\|_{\texttt{true}} &= 1 + |r| + \|k\|_{\texttt{true}} \\ \|\texttt{CThen}\ (r, k)\|_b &= |r| + \|k\|_b & \|\texttt{CStar}\ (r, k)\|_{\texttt{false}} &= 0 \end{aligned}$$

Theorem 2.2 (Totality of matcher)**.** *For any r, k, b, and s, the results of* `fmatch` r k b s *and* `apply` k b s *are defined. And hence, so is* `fmatchtop` r s*.*

Proof By course-of-values induction (i.e., where the inductive step may assume that the property holds for all lesser natural numbers, not only the immediate predecessor), with the lexicographic order on $(|s|, |r| + \|k\|_b)$, that is, either $|s|$ decreases, or $|s|$ stays the same while the total weight $|r| + \|k\|_b$ decreases (taking $|r| = 0$ for calls to `apply`). The function `fmatch` may call itself or `apply` only on arguments with strictly lower weight than its own, while `apply` may call `fmatch` on arguments with the same weight. As weights are always nonnegative, this ensures that no chain of calls can go on indefinitely. This property is easy to verify for all calls from `fmatch`:

- For `Char` c, in the second equation, $\|k\|_{\texttt{true}}$ may be larger than $1 + \|k\|_b$ (when $b =$ `false`), but $|s'| < |s|$. Everywhere else, $|s|$ stays the same.
- For `Eps`, `Seq` (r_1, r_2), both disjuncts in `Alt` (r_1, r_2), and the left disjunct in `Star` r_0, the total weight $|r| + \|k\|_b$ decreases by at least 1.
- For the right disjunct in `Star` r_0, $|r|$ decreases. The raw size of k itself increases, but $\|\texttt{CStar}\ (r_0, k)\|_{\texttt{false}} = 0 \leq \|k\|_b$, so the total argument weight still decreases.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

And from `apply`:

- For `CThen` (r, k), the arguments have weight $|r| + \|k\|_b$ on both sides.
- For `CStar` (r, k) we only perform a recursive call when $b = \texttt{true}$, in which case $|\texttt{Star}\ r| + \|k\|_b = 1 + |r| + \|k\|_{\texttt{true}} = \|\texttt{CStar}\ (r, k)\|_b$. (In the case $b = \texttt{false}$, had we omitted the "`b andalso`," the total argument weight of the `apply` would be 0, but *increase* to $1 + |r| + \|k\|_{\texttt{false}} > 0$ in the recursive call to `fmatch`.)

Termination of `fmatchtop` $r\ s$ then follows immediately. ■

A closer inspection of the code reveals that, when $|s|$ decreases in the equation for `Char` c, $\|k\|_{\texttt{true}}$ can only increase to at most its original maximal value. Thus, in any invocation `fmatchtop` $r\ s$, the depth of all call chains is at most $O(|r| \cdot |s|)$. However, since the case for `Alt` may make two consecutive such recursive calls, the bound on total running time is potentially as bad as $O(2^{|r|\cdot|s|})$. In fact, this worst case is fairly easy to hit: for example, running the matcher with regex $((\varepsilon \mid \varepsilon)^k \cdot \mathrm{a})^*$ (i.e., k `Seq`-connected copies of `Alt (Eps, Eps)`, followed by a `Char #"a"`, and wrapped in a `Star`) on input $\mathrm{a}^l\mathrm{b}$ (which should fail) appears instantaneous for $k = l = 4$, takes significant time for $k = l = 5$, and becomes completely impractical for $k = l = 6$. Note that the `Star`-body itself is actually non-nullable, so the original matcher would exhibit the same behavior.

2.2 Correctness of matcher

Having established that the augmented matcher always terminates, we now turn our attention to the correctness of its result, that is, that it *only* accepts strings in the language of the regex (soundness), and that it accepts *all* such strings (completeness). For this, we need to formally define the semantics of regular expressions:

Definition 2.3 (Language of regex)**.** *For all languages (sets of strings) L_1 and L_2, we write $L_1 \cdot L_2 = \{s_1@s_2 \mid s_1 \in L_1, s_2 \in L_2\}$; and for all natural numbers n, we define the language L^n by $L^0 = \{[\,]\}$ and $L^{n+1} = L \cdot L^n$. Then the language accepted by a regex r can be defined by structural induction on r:*

$$\begin{aligned} \mathcal{L}(\texttt{Char}\ c) &= \{[c]\} & \qquad \mathcal{L}(\texttt{Void}) &= \emptyset \\ \mathcal{L}(\texttt{Eps}) &= \{[\,]\} & \mathcal{L}(\texttt{Alt}\ (r_1, r_2)) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\ \mathcal{L}(\texttt{Seq}\ (r_1, r_2)) &= \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) & \mathcal{L}(\texttt{Star}\ r_0) &= \textstyle\bigcup_{n\geq 0} \mathcal{L}(r_0)^n \end{aligned}$$

We can then show

Theorem 2.4 (Soundness of `fmatch` and `fmatchtop`)**.** *If* `fmatch` $r\ k\ b\ s = \texttt{true}$*, then there exist (not necessarily unique) s', s'', and b', such that $s'@s'' = s$, $s' \in \mathcal{L}(r)$, and* `apply` $k\ b'\ s'' = \texttt{true}$ *by a subderivation of the given one. Hence, if* `fmatchtop` $r\ s = \texttt{true}$*, then $s \in \mathcal{L}(r)$.*

Proof For the first part, we proceed by induction on the derivation of `fmatch` $r\ k\ b\ s = \texttt{true}$, splitting into cases on r. We show the two most interesting cases; the others are very similar:

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

- Case $r = \texttt{Char}\ c$. We can only have $\texttt{fmatch}\ (\texttt{Char}\ c)\ k\ b\ s = \texttt{true}$ if $s = c :: s_1$ for some s_1 such that $\texttt{apply}\ k\ \texttt{true}\ s_1 = \texttt{true}$. Take $s' = [c]$, $s'' = s_1$, and $b' = \texttt{true}$. Then $[c]@s_1 = s$, $[c] \in \mathcal{L}(\texttt{Char}\ c)$, and $\texttt{apply}\ k\ \texttt{true}\ s_1 = \texttt{true}$ is a subderivation.
- Case $r = \texttt{Star}\ r_0$. Let $k_0 = \texttt{CStar}\ (r_0, k)$. For the match to succeed, at least one of the two disjuncts must be true:
  - Case $\texttt{apply}\ k\ b\ s = \texttt{true}$. Take $s' = [\,]$, $s'' = s$ and $b' = b$. Then, $[\,]@s = s$, $[\,] \in \mathcal{L}(r_0)^0 \subseteq \mathcal{L}(r_0)^* = \mathcal{L}(\texttt{Star}\ r_0)$, and we have the required subderivation.
  - Case $\texttt{fmatch}\ r_0\ k_0\ \texttt{false}\ s = \texttt{true}$. By IH on this subderivation, we obtain s_0, s_0', and b_0, such that $s_0@s_0' = s$, $s_0 \in \mathcal{L}(r_0)$, and $\texttt{apply}\ k_0\ b_0\ s_0' = \texttt{true}$; and thus $\texttt{fmatch}\ (\texttt{Star}\ r_0)\ k\ b_0\ s_0' = \texttt{true}$ by a smaller derivation. By IH on this derivation, we get that there exist s_1, s_1', and b_1 such that $s_1@s_1' = s_0'$, $s_1 \in \mathcal{L}(\texttt{Star}\ r_0)$, and $\texttt{apply}\ k\ b_1\ s_1' = \texttt{true}$ by an even smaller derivation. Take $s' = s_0@s_1$, $s'' = s_1'$, and $b' = b_1$; then $s_0@s_1@s_1' = s_0@s_0' = s$. And since $s_1 \in \mathcal{L}(\texttt{Star}\ r_0) = \mathcal{L}(r_0)^*$, we must have $s_1 \in \mathcal{L}(r_0)^n$ for some n, and thus $s_0@s_1 \in \mathcal{L}(r_0) \cdot \mathcal{L}(r_0)^n = \mathcal{L}(r_0)^{n+1} \subseteq \mathcal{L}(r_0)^* = \mathcal{L}(\texttt{Star}\ r_0)$.

Assume now that $\texttt{true} = \texttt{fmatchtop}\ r\ s = \texttt{fmatch}\ r\ \texttt{CInit}\ \texttt{true}\ s$. By the above argument, this means that, for some s', s'', and b', we have $s = s'@s''$, $s' \in \mathcal{L}(r)$, and $\texttt{apply}\ \texttt{CInit}\ b'\ s'' = \texttt{true}$. But the latter can only happen when $s'' = [\,]$, so we must have $s = s'@[\,] = s'$, and thus $s \in \mathcal{L}(r)$ as required. ■

Note that, for soundness of `fmatch`, the parameter b played no significant role in the result or its proof. For completeness, on the other hand, it is crucial that matching does not fail too often, and we need finer control over the value of b. That is, we must codify that $\texttt{fmatch}\ r\ k\ b\ s$ will invoke $\texttt{apply}\ k\ b'\ s'$ with $b' = \texttt{true}$ if either b was already `true`, or the matching of r consumed a non-empty prefix s of the input string to obtain s':

Theorem 2.5 (Completeness of `fmatch` and `fmatchtop`)**.** *If* $s \in \mathcal{L}(r)$ *and* $\texttt{apply}\ k\ (b \vee s \neq [\,])\ s' = \texttt{true}$, *then also* $\texttt{fmatch}\ r\ k\ b\ (s@s') = \texttt{true}$. *Hence, if* $s \in \mathcal{L}(r)$, *then* $\texttt{fmatchtop}\ r\ s = \texttt{true}$.

Proof The first part follows by a simple induction on r. Again, we show only the key cases here:

- Case $r = \texttt{Char}\ c$. Since $s \in \mathcal{L}(\texttt{Char}\ c)$, we must have $s = [c]$. Then, evidently $\texttt{fmatch}\ (\texttt{Char}\ c)\ k\ b\ ([c]@s') = \texttt{apply}\ k\ \texttt{true}\ s' = \texttt{apply}\ k\ (b \vee ([c] \neq [\,]))\ s' = \texttt{true}$.
- Case $r = \texttt{Seq}\ (r_1, r_2)$. From $s \in \mathcal{L}(\texttt{Seq}\ (r_1, r_2))$, we get that $s = s_1@s_2$ for some $s_1 \in \mathcal{L}(r_1)$ and $s_2 \in \mathcal{L}(r_2)$. Let $k_1 = \texttt{CThen}\ (r_2, k)$. We have $\texttt{fmatch}\ r\ k\ b\ (s@s') = \texttt{fmatch}\ r_1\ k_1\ b\ (s_1@s_2@s')$, which must be `true` by IH on r_1, if we can show that $\texttt{apply}\ k_1\ b_1\ (s_2@s') = \texttt{true}$, where $b_1 = (b \vee s_1 \neq [\,])$. That is, we must show $\texttt{fmatch}\ r_2\ k\ b_1\ (s_2@s') = \texttt{true}$, which we get from IH on r_2, if $\texttt{apply}\ k\ b_2\ s' = \texttt{true}$, where $b_2 = (b_1 \vee s_2 \neq [\,])$. But $b_2 = (b \vee s_1 \neq [\,] \vee s_2 \neq [\,]) = (b \vee (s_1@s_2) \neq [\,]) = (b \vee s \neq [\,])$, so we can directly use the assumption on k.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

- Case $r = \texttt{Star}\ r_0$. Since $s \in \mathcal{L}(r_0)^*$, we must have $s \in \mathcal{L}(r_0)^n$ for some n. We proceed by an inner induction on n:
  - If $n = 0$, we have $s \in \mathcal{L}(r_0)^0$, that is, $s = [\,]$. Then, it suffices to show that the left part of the disjunction in $\texttt{fmatch}\ r\ k\ b\ ([\,]@s')$ succeeds, for which we have $\texttt{apply}\ k\ b\ s' = \texttt{apply}\ k\ (b \vee [\,] \neq [\,])\ s' = \texttt{true}$, by the assumption on k.
  - If $n > 0$, we have $s = s_0@s_1$, where $s_0 \in \mathcal{L}(r_0)$ and $s_1 \in \mathcal{L}(r_0)^{n-1}$. If $s_0 = [\,]$, then $s = s_1$, and we can just use the inner IH. Otherwise, we may assume $s_0 \neq [\,]$, and we just need to prove that the second part of the disjunction succeeds (because the first part always at least terminates by the first part of Theorem 2.2). That is, taking $k_0 = \texttt{CStar}\ (r_0, k)$, we want to show that $\texttt{fmatch}\ r_0\ k_0\ \texttt{false}\ (s_0@s_1@s') = \texttt{true}$. By the (outer) IH on r_0, it suffices to show that $\texttt{apply}\ k_0\ (\texttt{false} \vee s_0 \neq [\,])\ (s_1@s') = \texttt{true}$, and by using the assumption that $s_0 \neq [\,]$, this means we need to show $\texttt{fmatch}\ (\texttt{Star}\ r_0)\ k\ \texttt{true}\ (s_1@s') = \texttt{true}$. But since $s_1 \in \mathcal{L}(r_0)^{n-1}$, we get this directly by the inner IH.

To complete the proof, take $k = \texttt{CInit}$, $b = \texttt{true}$, and $s' = [\,]$. Then, regardless of s, $\texttt{apply}\ k\ (b \vee s \neq [\,])\ s' = \texttt{true}$, and thus, by the above result, we have $\texttt{fmatchtop}\ r\ s = \texttt{fmatch}\ r\ k\ b\ (s@[\,]) = \texttt{true}$. ■

Summarizing the results:

Corollary 2.6 (Correctness of `matchtop`)**.** *For any r and s, there exists an a such that* $\texttt{matchtop}\ r\ s = a$, *and* $a = \texttt{true} \Leftrightarrow s \in \mathcal{L}(r)$.

Proof Follows directly from Theorems 2.1, 2.2, 2.4, and 2.5. ■

However, while the matcher is formally correct and has decent performance on a range of common examples, it also easily exhibits the exponential worst-case behavior mentioned after Theorem 2.2, even for sensibly written regexes. So there is still room for improvement.

2.3 Perspectives

Adding flag-based cycle detection to the original CPS matcher requires only a very minor modification to the code and correctness proof; this is arguably simpler than preprocessing the regex, even if we do not care about performance at all. Perhaps more significantly, in practical applications, we often want regex matching to not only return a boolean accept/reject answer, but also some kind of parse result, recording *how* the string was matched, and in particular how the nondeterministic choices were made in alternatives and iterations (Frisch & Cardelli, 2004; Nielsen & Henglein, 2011). From this extra information, it is easy to recover exactly which parts of the input string were matched by particular sub-regexes. For such applications, it may be a significant complication if the regex itself has been substantially rewritten before the matching proper (even if the set of matched strings is ultimately unchanged), whereas merely pruning away any unproductive paths in the matcher is unproblematic. It should thus be fairly straightforward to generalize the type `ans = bool` (and the combining forms `orelse` and `andalso`) in the augmented matcher to one carrying additional information on successes.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

3 Specializing the matcher to a regex

3.1 Staging the computation

As the first step of transformation, we will *stage* the matching process by translating the tree-structured regular expression into an equivalent, flat collection of mutually recursive continuation definitions, which can subsequently be evaluated against an actual input string. The correctness of this transformation is independent of what language the matcher is meant to recognize, or even its termination properties.

Informally, the translation can be seen as a simple, syntax-directed compilation schema: we traverse the regular expression in a way similar to the matcher, but instead of passing the continuation around in its entirety, we generate abstract machine code for any newly constructed continuations and pass only the code label (as a *continuation number*). Similarly, instead of applying the continuation, we just generate code for a call.

This translation process is completely independent of the input string. The boolean flag will also only be known at runtime, as the same continuation may be invoked with different flags at different times during matching. Thus, we will need to generate code to update and test the flag at the appropriate points.

The target code definition and compiler are shown in Figure 4. The intuitive meanings of the various computations (continuation bodies) in `comp` are as follows:

- `AtEnd`: the computation that succeeds iff there are no characters left in the string.
- `Expect` (c, i): the computation that checks that the first character (if any) of the string is equal to c, and invokes continuation i on the remainder of the string, with the flag set to true; otherwise, the computation fails.
- `Cont` (p, i): the computation that invokes continuation i on the current string. If p is false, the flag bit is set to false first; otherwise, it is passed along unmodified.
- `Fail`: the computation that always fails.
- `Or` (f_1, f_2): the computation that succeeds if at least one of f_1 or f_2 succeeds, and fails otherwise.

A *program* then consists of, for each continuation position i in a list, a pair $g = (u, f)$, with the boolean u specifying whether this computation is *unconditional*. An unconditional computation is always evaluated, whereas a conditional one is only evaluated if the character-consumption flag is currently true, and fails immediately otherwise. Additionally, the program explicitly identifies the *main* continuation, that is, the one that has to accept for the whole string to be accepted. (In the matcher, this would correspond to defining `fmatchtop` $r\ s =$ `apply` k_{m} `true` s, where $k_{\mathrm{m}} =$ `CThen` $(r,$ `CInit`$)$.)

The function `trans` takes as arguments a regex r, the continuation (number) i to be invoked on success, and an allocation counter (i.e., the first unused continuation number) n; it returns a triple consisting of the computation f representing the regex, a (possibly empty) list gs of continuation definitions generated, and the updated allocation counter n', where $n' = n + |gs|$. Finally, `transtop` defines continuation number 0 to be the *initial* continuation (i.e., the one that checks that nothing is left) and also generates an explicit definition for the main continuation, so that it too can be referenced by a number.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

```
datatype contno = CN of int

datatype comp =
    AtEnd
  | Expect of char * contno
  | Cont of bool * contno
  | Fail
  | Or of comp * comp

type ccomp = bool * comp
type pgm = ccomp list * contno

(* trans : re -> contno -> int -> comp * ccomp list * int *)
fun trans (Char c) i n = (Expect (c, i), [], n)
  | trans Eps i n = (Cont (true, i), [], n)
  | trans (Seq (r1, r2)) i n =
      let val (f2, gs2, n2) = trans r2 i n
          val (f1, gs1, n1) = trans r1 (CN n2) (n2+1)
      in (f1, gs2 @ [(true, f2)] @ gs1, n1) end
  | trans Void i n = (Fail, [], n)
  | trans (Alt (r1, r2)) i n =
      let val (f1, gs1, n1) = trans r1 i n
          val (f2, gs2, n2) = trans r2 i n1
      in (Or (f1, f2), gs1 @ gs2, n2) end
  | trans (Star r0) i n =
      let val (f0, gs0, n0) = trans r0 (CN n) (n+1)
          val f1 = Or (Cont (true, i), Cont (false, CN n0))
      in (f1, [(false, f1)] @ gs0 @ [(true, f0)], n0+1) end

(* transtop : re -> pgm *)
fun transtop r =
  let val (f, gs, n) = trans r (CN 0) 1
  in ([(true, AtEnd)] @ gs @ [(true, f)], CN n) end;
```

Fig. 4. The regex compiler.

Theorem 3.1 (Totality of `transtop`)**.** *For any r, the result of* `transtop` *r is defined.*

Proof By a straightforward structural induction on r, we see that `trans` r i n is defined for all i and n. Definedness of `transtop` r follows immediately. ■

Note that the translation, for simplicity, is expressed in terms of repeated list appends, which nominally makes it run in $O(|r|^2)$ time. This inefficiency is straightforward to avoid: since the newly generated continuation definitions are never subsequently inspected (by the translation itself), we could just incrementally prepend them to a globally threaded accumulator (like in general tree flattening); alternatively, we can replace all the "@"s with a node constructor for a binary tree of definition lists and subsequently flatten the resulting *rope* to a proper list in a single pass. We keep the current presentation for conciseness in stating and proving the correctness properties.

It is also easy to see that the overall *size* (not merely *length*) of the program generated for a regex r is $O(|r|)$, because in each recursive equation for `trans`, the total size of the

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

returned computation and any continuation definitions generated is linear in the size of r. (Note that, in the equation for `Star` r_0, the duplicated computation f_1 has a constant size, independent of r_0.) Summarizing the contributions of the various regex forms (and the top-level translator) to (1) the number of continuations generated and (2) the number of *references* to continuations (by `Expect` and `Cont`), we have

Regex	Conts	Refs
(top)	2	0
`Char` −	0	1
`Eps`	0	1
`Seq` (−, −)	1	0

Regex	Conts	Refs
`Void`	0	0
`Alt` (−, −)	0	0
`Star` −	2	4

To cement our intuitions about the intuitive behavior of the translation, consider the example in Figure 5. (While the regex in question and its translation are moderately large, the individual parts can largely be understood independently of each other.) The top left of the figure shows the tree representation of the regex, where we have explicitly labeled all the subexpression nodes for ease of reference. Character data is drawn as boxed nodes; the other regex forms are circles.

On the top right is a tabular representation of all the calls made to `trans` $r\ i\ n$ (once each) during the translation, as well as a summary of their results: the new allocation counter n' (as advanced by the regex node itself and/or its children), and the continuation definitions (with contents to be detailed below) and references *produced* (not merely propagated) by the node. A labeled edge $k_{i_1} \xrightarrow{\ell} k_{i_2}$ means that the computation f_{i_1} (i.e., number i_1) includes a reference `Expect` (c, i_2) (where $\ell = c$) or `Cont` (p, i_2) ($\ell = \varepsilon$). Note that, while the edge's *target* i_2 is always either i or a continuation internally generated by the node, its *source* i_1 may not even have been generated yet, as the f returned by the translation represents a collection of incoming edges from elsewhere. The first row in the table represents the contributions of `transtop`, that is, the nodes for the initial and main continuations.

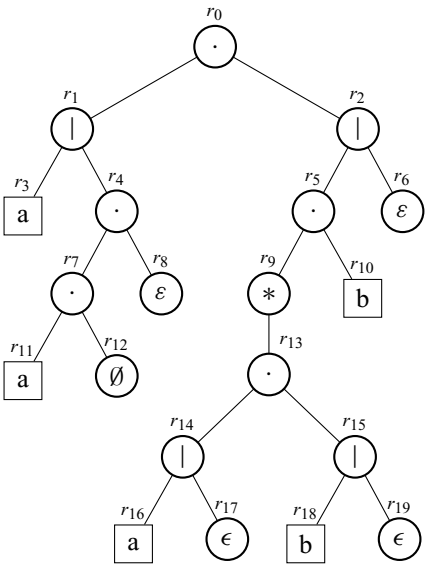
The bottom of the figure is a graphic representation of the program generated from the regex. Each numbered continuation node is labeled by the continuation it represents, with `CInit` shown compactly as "✓", `CThen` (r, k) as "r; k", and `CStar` (r, k) as "$r^{\diamond}$; k". (These labels are for reference only; they are not explicitly represented in the data structure.)

The *main* continuation is indicated with an unlabeled incoming edge, while the *initial* one is marked with a double circle. Also, conditional computations are shown with a dashed node border. Similarly, edges labeled with a character c represent `Expect` (c, i)-references, while the (thinner) ones labeled by an ε correspond to `Cont` (p, i); the edge is drawn dashed if $p =$ `false`.

As suggested by the familiar-looking notation, a string belongs to the language of the regex iff it is possible to traverse the graph from main to initial continuation, while consuming the corresponding string characters on character-labeled edges, and following ε-labeled ones freely. This picture is slightly complicated by the loop avoidance flag: initially, the flag is true; it is set to false when following a *dashed ε-edge*, and back to true on following any *character edge*. Moreover, when the flag is currently false, it is not possible to follow either kind of ε-edge to a *dashed node*. We will shortly formalize this interpretation and its correctness.

Note in particular that `Void`-regexes may cause the graph to contain unreachable (k_6) and/or dead-end (k_7) nodes, but neither will require any special treatment in the

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press



r	i	n	n'	Conts	Refs
top	–	(0)	(9)	k_0, k_8	
r_0	k_0	1	8	k_5	
r_1	k_5	6	8		
r_2	k_0	1	5		
r_3	k_5	6	6		$k_8 \xrightarrow{a} k_5$
r_4	k_5	6	8	k_6	
r_5	k_0	1	5	k_1	
r_6	k_0	5	5		$k_5 \xrightarrow{\varepsilon} k_0$
r_7	k_6	7	8	k_7	
r_8	k_5	6	6		$k_6 \xrightarrow{\varepsilon} k_5$
r_9	k_1	2	5	k_2, k_4	$k_2 \xrightarrow{\varepsilon} k_1$, $k_5 \xrightarrow{\varepsilon} k_1$ $k_2 \xrightarrow{\varepsilon} k_4$, $k_5 \xrightarrow{\varepsilon} k_4$
r_{10}	k_0	1	1		$k_1 \xrightarrow{b} k_0$
r_{11}	k_7	8	8		$k_8 \xrightarrow{a} k_7$
r_{12}	k_6	7	7		
r_{13}	k_2	3	4	k_3	
r_{14}	k_3	4	4		
r_{15}	k_2	3	3		
r_{16}	k_3	4	4		$k_4 \xrightarrow{a} k_3$
r_{17}	k_3	4	4		$k_4 \xrightarrow{\varepsilon} k_3$
r_{18}	k_2	3	3		$k_3 \xrightarrow{b} k_2$
r_{19}	k_2	3	3		$k_3 \xrightarrow{\varepsilon} k_2$

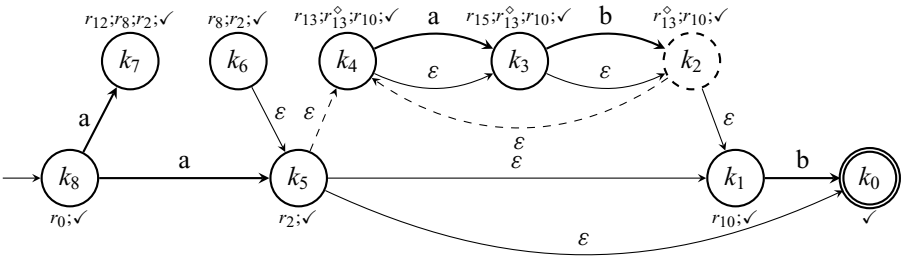


Fig. 5. Visualization of compilation of regex $\big(\mathrm{a} \mid \mathrm{a} \cdot \emptyset \cdot \varepsilon\big) \cdot \big(((\mathrm{a} \mid \varepsilon) \cdot (\mathrm{b} \mid \varepsilon))^* \cdot \mathrm{b} \mid \varepsilon\big)$.

following. Also, the body of the star expression in the example is evidently nullable, so it non-trivially exercises the loop-avoidance features of the construction. In fact, the general pattern $r = (c_1 \mid \varepsilon) \cdots (c_n \mid \varepsilon)$, especially when represented as an arbitrarily *balanced* Seq-tree (as opposed to a left- or right-degenerate one), highlights an awkward case for the *standardization* approach discussed by Harper (1999) (i.e., to statically convert the body regex r to an r^- with $\mathcal{L}(r^-) = \mathcal{L}(r) \setminus \{[\,]\}$): if the given, syntax-directed construction of r^- is implemented naively, it potentially causes an exponential blowup in the size of the regexes, because they are conceptually trees, not DAGs; recognizing opportunities for *factoring* the transformed regex, so as to properly share common subexpressions, while probably feasible, could significantly complicate the algorithm and its proof.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

```
val nth = List.nth

(* interp : ccomp list -> comp -> bool -> char list -> ans *)
fun interp gs AtEnd b [] = true
  | interp gs AtEnd b _ = false
  | interp gs (Expect (c, i)) b [] = false
  | interp gs (Expect (c, i)) b (c' :: s') =
      c = c' andalso interpi gs i true s'
  | interp gs (Cont (true, i)) b s = interpi gs i b s
  | interp gs (Cont (false, i)) b s = interpi gs i false s
  | interp gs Fail b s = false
  | interp gs (Or (f1, f2)) b s = interp gs f1 b s orelse interp gs f2 b s
(* interpc : ccomp list -> ccomp -> bool -> char list -> ans *)
and interpc gs (true, f) b s = interp gs f b s
  | interpc gs (false, f) b s = b andalso interp gs f b s
(* interpi : ccomp list -> contno -> bool -> char list -> ans *)
and interpi gs (CN n) b s = interpc gs (nth (gs, n)) b s

(* irun : pgm -> char list -> ans *)
fun irun (gs, i1) s = interpi gs i1 true s

(* imatchtop : re -> char list -> ans *)
fun imatchtop r s = irun (transtop r) s
```

Fig. 6. The backtracking interpreter.

3.2 Correctness of staging

We will now show that the splitting of the matcher into a compile-time and a runtime part is meaning-preserving. Specifically, in Figure 6, we define a simple interpreter that can run a compiled regex on an input string.

To show the equivalence of `imatchtop` to `fmatchtop`, we first define the notion of *embedding* a sequence of continuation definitions in a complete program:

Definition 3.2 (Continuation slice embedding)**.** *We write* $gs \vdash n \hookrightarrow gs'$ *if* gs' *is a slice of* gs *starting at* n*, that is, if* $gs = gs_l @ gs' @ gs_r$ *for some* gs_l *and* gs_r*, where* $|gs_l| = n$*. Evidently,* $gs \vdash n \hookrightarrow gs_1 @ gs_2$ *iff* $gs \vdash n \hookrightarrow gs_1$ *and* $gs \vdash n + |gs_1| \hookrightarrow gs_2$*.*

Definition 3.3 (Representation of continuation in program)**.** *We say that continuation* k *is* represented *as computation number* $i = \texttt{CN}\ n_i$ *in* gs*, written* $gs \vdash k \sim i$*, if* $gs \vdash n_i \hookrightarrow [g]$*, and (by induction on* k*):*

- *If* $k = \texttt{CInit}$*, then* $g = (\texttt{true}, \texttt{AtEnd})$*.*
- *If* $k = \texttt{CThen}\ (r, k')$*, then* $g = (\texttt{true}, f)$ *for some* f*,* i'*,* n*,* n'*, and* gs'*, where* $gs \vdash k' \sim i'$*,* `trans` $r\ i'\ n = (f, gs', n')$*, and* $gs \vdash n \hookrightarrow gs'$*.*
- *If* $k = \texttt{CStar}\ (r, k')$*, then* $g = (\texttt{false}, f)$ *for some* f*,* i'*,* n*,* n'*, and* gs'*, where* $gs \vdash k' \sim i'$*,* `trans` $(\texttt{Star}\ r)\ i'\ n = (f, gs', n')$*, and* $gs \vdash n \hookrightarrow gs'$*.*

(Note that this is a purely syntactic property, with no references to b or s.)

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Lemma 3.4 (Correctness of continuation representations). *If $gs \vdash k \sim i$, then for all b and s:*

a. $\texttt{apply}\ k\ b\ s \simeq \texttt{interpi}\ gs\ i\ b\ s$.
b. If $\texttt{trans}\ r\ i\ n = (f, gs', n')$ *and* $gs \vdash n \hookrightarrow gs'$, *then* $n' = n + |gs'|$ *and* $\texttt{fmatch}\ r\ k\ b\ s \simeq \texttt{interp}\ gs\ f\ b\ s$.

Proof Assume $gs \vdash k \sim i$, that is, $i = \texttt{CN}\ n_i$, $gs \vdash n_i \hookrightarrow [g]$, and $g = (u, f)$ satisfies the conditions of Definition 3.3; and let b and s be given. Formally, we prove the Kleene equalities by separate (mutual) inductions on the derivations of either side, but since the arguments are virtually identical in both directions, we present them here together in equational style.

For part (a), we need to show that $\texttt{apply}\ k\ b\ s \simeq \texttt{interpc}\ gs\ (u, f)\ b\ s$. The most complex case is

- Case $k = \texttt{CStar}\ (r, k')$, so $u = \texttt{false}$, $gs \vdash k' \sim i'$, and for some n, n', and gs', we have $\texttt{trans}\ (\texttt{Star}\ r)\ i'\ n = (f, gs', n')$, with $gs \vdash n \hookrightarrow gs'$. If $b = \texttt{false}$, we simply have $\texttt{apply}\ (\texttt{CStar}\ (r, k'))\ \texttt{false}\ s \simeq \texttt{false} \simeq \texttt{interpc}\ gs\ (\texttt{false}, f)\ \texttt{false}\ s$. And if $b = \texttt{true}$, then $\texttt{apply}\ (\texttt{CStar}\ (r, k'))\ \texttt{true}\ s \simeq \texttt{fmatch}\ (\texttt{Star}\ r)\ k'\ \texttt{true}\ s \simeq^{\text{IH(b)}} \texttt{interp}\ gs\ f\ \texttt{true}\ s \simeq \texttt{interpc}\ gs\ (\texttt{false}, f)\ \texttt{true}\ s$. (Here, we have explicitly indicated which of the equalities relies on the induction hypothesis.)

For part (b), the key cases for r are

- Case $r = \texttt{Char}\ c$. Then $f = \texttt{Expect}\ (c, i)$, $gs' = [\,]$, and $n' = n$, so clearly $n' = n + |gs'|$. If $s = [\,]$, then $\texttt{fmatch}\ (\texttt{Char}\ c)\ k\ b\ [\,] \simeq \texttt{false} \simeq \texttt{interp}\ gs\ (\texttt{Expect}\ (c, i))\ b\ [\,]$. If $s = c' :: s'$ for some $c' \neq c$, then again both sides are immediately equal to $\texttt{false}$. And if $s = c :: s'$, then $\texttt{fmatch}\ (\texttt{Char}\ c)\ k\ b\ (c :: s') \simeq \texttt{apply}\ k\ \texttt{true}\ s' \simeq^{\text{IH(a)}} \texttt{interpi}\ gs\ i\ b\ s' \simeq \texttt{interp}\ gs\ (\texttt{Expect}\ (c, i))\ b\ (c :: s')$.
- Case $r = \texttt{Star}\ r_0$. Then, we have $f = \texttt{Or}\ (\texttt{Cont}\ (\texttt{true}, i), \texttt{Cont}\ (\texttt{false}, \texttt{CN}\ n_0))$, $gs' = [(\texttt{false}, f)]@gs_0@[(\texttt{true}, f_0)]$, and $n' = n_0 + 1$, where $\texttt{trans}\ r_0\ (\texttt{CN}\ n)\ (n + 1) = (f_0, gs_0, n_0)$. Further, $n_0 = n + 1 + |gs_0|$, so $n + |gs'| = n + 1 + |gs_0| + 1 = n_0 + 1 = n'$. And $gs \vdash n \hookrightarrow [(\texttt{false}, f)]$, $gs \vdash n + 1 \hookrightarrow gs_0$, and $gs \vdash n_0 \hookrightarrow [(\texttt{true}, f_0)]$. With $k_0 = \texttt{CStar}\ (r_0, k)$ and $i_0 = \texttt{CN}\ n$, we immediately have $gs \vdash k_0 \sim i_0$. Thus,
  $\texttt{fmatch}\ (\texttt{Star}\ r_0)\ k\ b\ s \simeq \texttt{apply}\ k\ b\ s\ \vee\ \texttt{fmatch}\ r_0\ k_0\ \texttt{false}\ s$
  $\simeq^{\text{IH(a,b)}} \texttt{interpi}\ gs\ i\ b\ s\ \vee\ \texttt{interp}\ gs\ f_0\ \texttt{false}\ s$
  $\simeq \texttt{interp}\ gs\ (\texttt{Cont}\ (\texttt{true}, i))\ b\ s\ \vee\ \texttt{interp}\ gs\ (\texttt{Cont}\ (\texttt{false}, \texttt{CN}\ n_0))\ b\ s$
  $\simeq \texttt{interp}\ gs\ f\ b\ s$. ■

Theorem 3.5 (Correctness of staging). *For any r and s,* $\texttt{imatchtop}\ r\ s \simeq \texttt{fmatchtop}\ r\ s$.

Proof Let $(gs, i_1) = \texttt{transtop}\ r$ (always defined by Theorem 3.1). Then, $gs = [(\texttt{true}, \texttt{AtEnd})]@gs'@[(\texttt{true}, f)]$ and $i_1 = \texttt{CN}\ n'$, where $(f, gs', n') = \texttt{trans}\ r\ (\texttt{CN}\ 0)\ 1$. Thus, $gs \vdash 0 \hookrightarrow [(\texttt{true}, \texttt{AtEnd})]$, $gs \vdash 1 \hookrightarrow gs'$, and $gs \vdash n' \hookrightarrow [(\texttt{true}, f)]$. For $k_0 = \texttt{CInit}$ and $i_0 = \texttt{CN}\ 0$, clearly $gs \vdash k_0 \sim i_0$, and thus, using Lemma 3.4(b), $\texttt{fmatchtop}\ r\ s \simeq \texttt{fmatch}\ r\ k_0\ \texttt{true}\ s \simeq \texttt{interp}\ gs\ f\ \texttt{true}\ s \simeq \texttt{interpi}\ gs\ i_1\ \texttt{true}\ s \simeq \texttt{imatchtop}\ r\ s$. ■

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Note that the staging in itself does not materially affect the runtime of the matching process, because the interpreted program essentially still executes in lockstep with the backtracking matcher. (In fact, the two-phase version may be slightly slower, because we have used a linear list for representing the program, making lookups of high-numbered continuations potentially slow. It should be clear, however, that `transtop` could convert the final list of continuation definitions into an SML `vector` (read-only array); and correspondingly, `interpi` could use constant-time vector indexing instead of `nth`.) However, this refinement is not actually needed, as naively interpreting the instructions in `imatchtop` is merely a stepping stone for proving correctness, and will be completely eliminated in the next transformation stage, anyway.

3.3 Perspectives

In many ways, the compilation step is analogous to Thompson's construction (Thompson, 1968), which converts a tree-structured regex into a general NFA graph. It is a bit more parsimonious in node allocation for alternatives, as a single choice node may have an arbitrary combination of outgoing ε- and/or character-labeled edges; this makes especially the representation of regexes with character classes (e.g., `[a-z0-9_]` in common notation) more compact, without any explicit post-processing.

The interpreter then corresponds to a backtracking traversal of the NFA, that is, "nondeterminstically" choosing which outgoing edge to follow and reconsidering the choice if it leads to a failure.

A key difference to classical NFAs, however, is that the compile-time loop prevention ensures that there are no *entirely undashed* ε-cycles in the graph; that is, if we also correctly maintain and test the flag bit at runtime, any sequence of nondeterministic choices must eventually consume an input character or fail finitely. We can thus simulate the NFA directly, without explicitly keeping track of all the visited nodes.

A more subtle difference is that, while NFA vertices are normally considered just an unordered set, the *numbering* of the continuations in a program will turn out to be very significant for the constructions in the next section.

4 From lazy to eager interpretation

The usual informal interpretation of an NFA is that it specifies a (nondeterministic) *control flow* graph: an edge from k_i to $k_{i'}$ (i.e., node number i containing a reference to i' as one the `Or`-tree leaves in the computation f_i) means that, during execution, the program can choose to transition from node i to node i', with a potentially shorter remaining string. However, looking at the code of the interpreter, we see that it actually treats the program more as a *data flow* graph, where an edge from k_i to $k_{i'}$ means that the result of k_i may *depend* functionally on that of $k_{i'}$, again on a potentially shorter string.

For standard NFAs (i.e., with possibly *unguarded* ε-cycles), such an interpretation is perhaps less natural, as a dependency loop (also known as a *black hole*) in the graph would normally be considered a global *error*, rather than a recoverable, local *failure*. However,

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

with the flag-based cycle prevention, we are effectively evaluating a *DAG*, not a general graph, so the issue does not arise.

The original, top-down interpreter evaluates this DAG completely naively, ignoring all sharing and recomputing the same intermediate results over and over. (On the other hand, it will not evaluate the right child of an `Or`-computation at all, if the left one succeeds.) We can think of this as a *lazy* (in the by-*name* sense) evaluation strategy. An immediate improvement would thus be to *memoize* the results, that is, to remember and reuse the results of all continuation invocations, corresponding to a by-*need* evaluation.

But it is easy to see that, for the top-level matcher to return a `false` result, every single `orelse`-calculation in the tree/DAG must do likewise. Thus, unless we have a usage scenario in which a substantial majority of matcher invocations can be expected to succeed, we might as well assume the worst case and *eagerly* (or by-*value*) evaluate *all* continuation results for the string and its suffixes, in a bottom-up fashion. While this may perform some nominally unnecessary calculations, it makes the overall computation process much more streamlined and regular than opportunistic memoization, and thus both more efficiently implementable and easier to analyze.

In fact, the dependency graph has some additional structure that makes an eager strategy particularly appealing: the DAG consists of an $|s|$-length stack of *blocks*, with all dependency links going either between result slots in the same block (`Cont`-references) or to the one immediately below (`Expect`-references). The cross-block dependencies are determined by the character c at that position in the string and correspond exactly to the `Expect` (c, i')-references in the program; the `Expect` (c', i')-references with $c' \neq c$ contribute nothing for this string position.

Within each block, there are actually twice as many result slots as there are numbered continuations in the program, corresponding to the flag argument to the continuation being either `false` or `true`. All dependency links from one block to the one below go only to the `true` slots of the latter (because an `Expect`-reference sets the flag to `true`); links within each block go either to a *lower-numbered* slot for the *same* flag value, or from a `true`-flag slot to an arbitrary `false`-flag one. The result of the `false` slot for a *conditional* computation is fixed as `false`, without any further dependencies. It is fairly easy to see (informally, at least) that this makes the dependency graph cycle-free.

At the very bottom of the dependency DAG, there is a special block of slots, in which those corresponding to computations containing an `AtEnd`-pseudoreference are forced to `true`, `Expect` (c, i)-references are ignored (as returning `false`), and the intra-block links are handled as above. Thus, to compute the results for the topmost block of slots, and especially the slot corresponding to the main continuation being invoked with a `true` flag, we start at the bottom of the DAG (corresponding to the *end* of the string) and work upward to the root, computing the results for each block in sequence (first all the `false` slots, then the `true` ones, in numerical order). Between each pair of consecutive blocks, we only need to retain the values of the `true`-slots from the lower block.

As a possible runtime optimization, if the results from some block are *all* false, we know that the overall computation can never recover, and we may safely fail early. (For simplicity, our code will not exploit this property, however.)

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

```
type state = ans list  (* only if complete *)

type 'a nextc = (char * 'a) option

(* aevali : contno -> ans list -> ans *)
fun aevali (CN n) v = nth (v, n)

(* aeval : comp -> bool -> ans list -> ans list -> state nextc -> ans *)
fun aeval AtEnd _ _ _ NONE = true
  | aeval AtEnd _ _ _ (SOME _) = false
  | aeval (Expect _) _ _ _ NONE = false
  | aeval (Expect (c, i)) _ _ _ (SOME (c', v')) = c = c' andalso aevali i v'
  | aeval (Cont (true, i)) true vf vt cv = aevali i vt
  | aeval (Cont (true, i)) false vf vt cv = aevali i vf
  | aeval (Cont (false, i)) _ vf vt cv = aevali i vf
  | aeval Fail _ _ _ cv = false
  | aeval (Or (f1, f2)) b vf vt cv =
      aeval f1 b vf vt cv orelse aeval f2 b vf vt cv

(* aevalc : ccomp -> bool -> ans list -> ans list -> state nextc -> ans *)
fun aevalc (true, f) b vf vt cv = aeval f b vf vt cv
  | aevalc (false, f) b vf vt cv = b andalso aeval f b vf vt cv

(* astep : pgm -> state nextc -> state *)
fun astep (gs, _) cv =
  let fun asf [] vf = vf
        | asf (g :: gs') vf = asf gs' (vf @ [aevalc g false vf [] cv])
      fun ast [] vf vt = vt
        | ast (g :: gs') vf vt = ast gs' vf (vt @ [aevalc g true vf vt cv])
  in ast gs (asf gs []) [] end

(* aobs : pgm -> state -> ans *)
fun aobs (_, i1) v = aevali i1 v

(* arun : pgm -> char list -> ans *)
fun arun pgm s =
  let val v = foldr (fn (c,v) => astep pgm (SOME (c,v))) (astep pgm NONE) s
  in aobs pgm v end

(* amatchtop : re -> char list -> ans *)
fun amatchtop r s = arun (transtop r) s
```

Fig. 7. The eager, computation-sharing interpreter.

4.1 An eager interpreter

The code realizing the above-described approach is in Figure 7. The function `aeval` finds the value of a computation, given the value of the flag, the current character (unless we are at the very end of the string), and the results of all other continuations it may depend on. (Attempts to reference an unavailable continuation result, including a direct self-reference, will cause `aevali` (via `nth`) to raise an exception; the correctness proof must establish that this will not happen.)

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

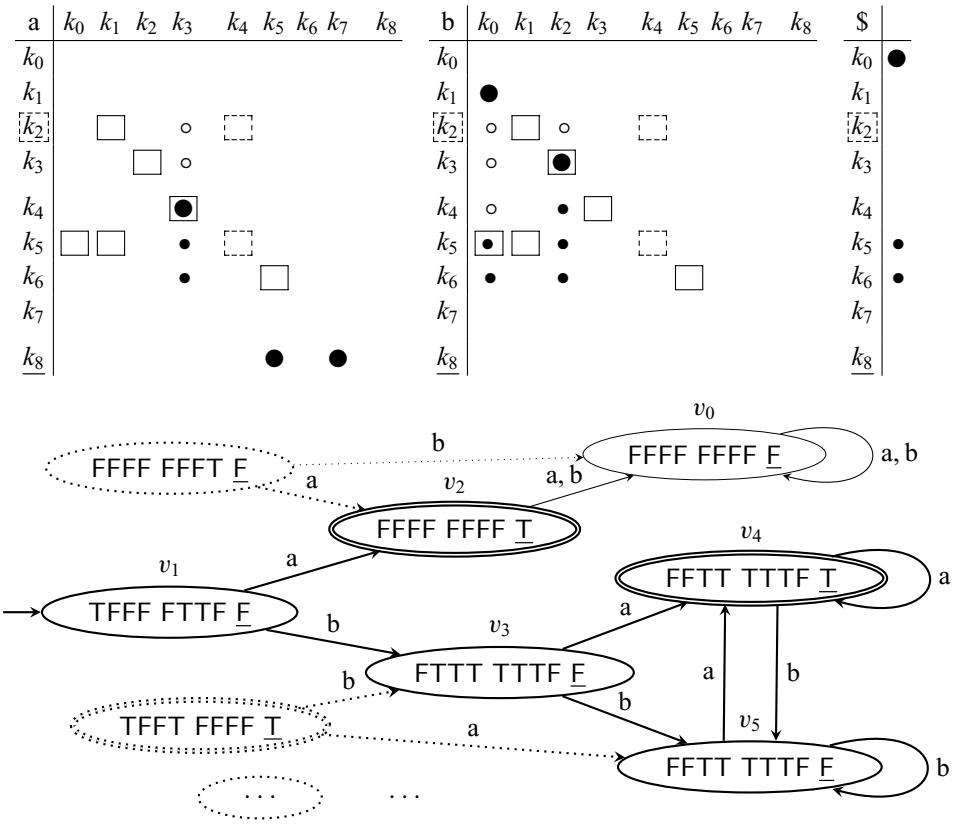


Fig. 8. Visualization of transition functions and (fragment of) state space for running example.

Then `aevalc` evaluates individual continuation definitions, and `astep` computes, in two passes, the results of all continuations, given the current character and the corresponding results for the remainder of the string (except if we are at the end). Finally, `aobs` simply extracts the result of the main continuation from the vector, and `arun` iterates *backward* through all the characters of the string before reading off the final answer.

As coded, the incremental construction of the next-state vector in the loops `asf` and `ast` of `astep` is somewhat inefficient because we need to both append to the vector and access random existing elements. This would be easy to achieve efficiently with mutable, single-threaded arrays, as we never modify the contents of a cell after initializing it. Alternatively, at a modest performance cost ($O(\log n)$, instead of $O(1)$, time per lookup), we could use random-access lists (Okasaki, 1995) with a purely functional implementation.

The state machine corresponding to the regex from Figure 5 is visualized in Figure 8. The top part of the figure shows an *extensional* representation of the transition functions for each character. In the table for a character c, there is a full mark ● in row i, column i' whenever there is a c-labeled edge from node k_i to $k_{i'}$. Similarly, there is a (dashed) box around each such position whenever there is a (dashed) ε-transition between the two nodes. (Note that the boxes are in the same positions for all characters c.) Finally, the row label k_i is written in a dashed box if it corresponds to a dashed node in the graph, and the *main* continuation (i.e., with the unlabeled incoming edge) is underlined. (The 4-wise

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

grouping with extra spacing is merely a visual aid for easier counting, with no semantic significance.) This table will in general be fairly sparse, with the number of non-blank positions linear in the size of the original regex.

Additionally, the table elements corresponding to the (pre)-ε-closure of the character transitions, derived using the boxes and already filled entries, are marked with a smaller bullet. Because of the ordering properties, this closure can always be uniformly computed in two passes per column, with the filled bullets ($\bullet$) corresponding to those entries marked during the first pass in `astep`, and the open ones ($\circ$) in the second one. (The kinds of mark do not matter for the actual transition function; they merely illustrate *when* the information is computed.)

The top rightmost, single-column table similarly marks the directly accepting node ($\boxed{\bullet}$) and those from which this node is reachable by ε-edges ($\bullet$), with the latter again computed in two linear passes (in fact, one pass would suffice here). It can be seen as the transition function for the end-of-input token, that is, the *initial* state of the machine when processing the input from right to left.

As suggested by the notation, the state transition function corresponds precisely to a boolean multiplication of the transition matrix and the state (column) vector. That is, the new state will have a true value in position i precisely if there exists an i' such that there is a mark in matrix position (i, i'), and i' is true in the old state. We stress, however, that the transition matrices are *not* explicitly precomputed, which would actually make them slower to apply, since they can in general be quite dense with all the closure marks added. For example, the a-matrix for regex $(\mathrm{a} \mid \varepsilon)^n$ has n $\boxed{\bullet}$-entries in the lower diagonal, and $\bullet$ -entries everywhere below that.

The bottom part of the figure visualizes the *state space* of the machine. Each node corresponds to a boolean vector, with the node drawn with a double border precisely when its result for the (underlined) main continuation k_8 is true. Every node has exactly one outgoing edge per character in the alphabet. The initial state from above is labeled v_1.

The state labeled v_0, the all-false vector, is the *canonical* dead state; clearly, no nonzero state can be reached from it by pre-multiplication with any transition matrix, whether derived from a regex or not. It is drawn thinner, as are the transitions to it.

Note that the state-space diagram conceptually includes a node for all 512 possible state vectors, with corresponding transition edges between them. However, the vast majority of these (in our example, though not always) are *unreachable* from the starting state. A few such nodes, and the transitions from them, are drawn *dotted* in the figure. The nine state vectors with exactly one true position (e.g., the one for k_7 in top left part of the diagram) form a *spanning set*: column i of each transition matrix for a character (or a sequence of such characters) corresponds to the value of transition function (or their composition) on the i'th singleton vector.

4.2 Correctness of eager evaluation

We start by formalizing the above-observed ordering constraints on references:

Definition 4.1 (Well-formed and ordered programs)**.** *We say that a program* $(gs, \texttt{CN}\ n_1)$ *with* $|gs| = n$ *is* well formed *if* $0 \leq n_1 < n$*, and whenever* $gs \vdash n_i \hookrightarrow [(u, f)]$*, then all*

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

occurrences of `Cont` $(p, \texttt{CN}\ n_{i'})$ *and* `Expect` $(c, \texttt{CN}\ n_{i'})$ *in* f *satisfy* $0 \leq n_{i'} < n$. *Further, a well formed program is (topologically)* ordered *if the occurrences of* `Cont` $(p, \texttt{CN}\ n_{i'})$ *in* f *also satisfy* $n_{i'} < n_i$, unless $u = p = \texttt{false}$.

(Programs constructed by `transtop` also satisfy that references `Expect` $(c, \texttt{CN}\ n_{i'})$ at position n_i are backward-only. That is an accidental property, which is not required by the constructions in the following.)

A well-formed program can be executed by the lazy interpreter in Figure 6 without `nth` aborting, but it might still recurse infinitely because of circularities, and/or the subcomputation dependencies at runtime might vary based on the input string. On the other hand, for an ordered program, we will show that we can always safely evaluate the computations using two left-to-right passes. As already indicated, the translation does indeed produce such programs:

Lemma 4.2 (Orderedness of compiler output)**.** *If* `transtop` $r = pgm$, *then* pgm *is ordered.*

Proof It is an easily checked invariant that, in any call to `trans` r $(\texttt{CN}\ n_i)$ n, whether top-level in `transtop`, or recursive from inside `trans`, we have $n_i < n$; and that if `trans` r $(\texttt{CN}\ n_i)$ $n = (f, gs, n')$, then the well-formedness and ordering constraints are satisfied by f (when placed in any position $\geq n'$) and by gs (when placed at precisely position n).

The only place where the "unless" clause in the definition of ordering is needed is in the translation of `Star` r_0, in which the conditional computation $[(\texttt{false}, f_1)]$ is generated for position n. (All other continuation definitions have $u = \texttt{true}$.) Here, in $f_1 = \texttt{Or}\ (\texttt{Cont}\ (\texttt{true}, \texttt{CN}\ n_i), \texttt{Cont}\ (\texttt{false}, \texttt{CN}\ n_0))$, the first disjunct does satisfy $n_i < n$ (as it needs to), but the second one does not, because of the "forward reference" to $n_0 = n + |gs_0| \not< n$. ■

Lemma 4.3 (Totality of `aeval` and `aevalc`)**.** *Suppose* $|gs| = n$; *cv is either* `NONE` *or* `SOME` (c, v) *with* $|v| = n$; *and either* $b = \texttt{false}$, $|v_{\mathrm{f}}| = n_i$, *and* $|v_{\mathrm{t}}| = 0$; *or* $b = \texttt{true}$, $|v_{\mathrm{f}}| = n$, *and* $|v_{\mathrm{t}}| = n_i$. *Then, when* $gs[n_i] = (u, f)$ *satisfies the ordering constraints, we have*

1. `aeval` $f\ b\ v_{\mathrm{f}}\ v_{\mathrm{t}}$ *cv is defined, unless* $b = u = \texttt{false}$
2. `aevalc` $(u, f)\ b\ v_{\mathrm{f}}\ v_{\mathrm{t}}$ *cv is always defined.*

Proof Part (1) follows by a simple structural induction on f, where we just need to check that all calls to `aevali` succeed. The one for `Expect` (c, i') is immediate by the assumption on cv. And for `Cont` (p, i'), the only problematic case is when $p = u = \texttt{false}$ (so the ordering constraint only guarantees $n_{i'} < n$) and $b = \texttt{false}$ (so $|v_{\mathrm{f}}| = n_i$, and we may not have $n_{i'} < n_i$); but this situation is specifically ruled out. For part (2), we then immediately verify that `aevalc` does not invoke `aeval` in the forbidden case. ■

Lemma 4.4 (Totality of `astep`)**.** *If* pgm *is ordered and of length n, then*

1. `astep` $pgm\ \texttt{NONE} = v_1$ *for some* v_1 *with* $|v_1| = n$.
2. *For all c, and v with* $|v| = n$, `astep` $pgm\ (\texttt{SOME}\ (c, v)) = v'$ *for some* v' *with* $|v'| = n$.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Proof For each of the two loops `asf` and `ast`, we easily verify that they call `aevalc` with precisely the vector lengths stipulated by Lemma 4.3; thus, definedness follows immediately from part (2) of that lemma. ■

Theorem 4.5 (Totality of `arun`)**.** *For any ordered pgm and s,* `arun` *pgm s is defined.*

Proof Let $(gs, i_1) = pgm$. By induction on s, using Lemma 4.4, we immediately get that $v =$ `foldr (...) (...) s` is defined; and since $i_1 < |gs| = |v|$, so is `aobs` $pgm\ v =$ `aevali` $i_1\ v$. ■

Having established that the eager interpreter always returns a result, let us next verify that it agrees with the lazy/backtracking one.

Definition 4.6 (Tabulation of continuation values)**.** *We first define what it means for v to (partially) tabulate the continuation results for gs on b and s:*

$$gs \vdash v \Leftarrow_b s \iff \forall n < |v|.\ v_{[n]} = \texttt{interpi}\ gs\ (\texttt{CN}\ n)\ b\ s$$

We also define $gs \vdash cv \Leftarrow^{\text{ht}} s$*, which returns (if possible) the head character of s and the complete vector of continuation results for the tail:*

$$gs \vdash \texttt{NONE} \Leftarrow^{\text{ht}} [\,]$$
$$gs \vdash \texttt{SOME}\ (c, v') \Leftarrow^{\text{ht}} (c :: s') \iff gs \vdash v' \Leftarrow_{\texttt{true}} s' \wedge |v'| = |gs|$$

Lemma 4.7 (Soundness of eager evaluation)**.** *Suppose we have* $gs \vdash v_{\text{f}} \Leftarrow_{\texttt{false}} s$*,* $gs \vdash v_{\text{t}} \Leftarrow_{\texttt{true}} s$*, and* $gs \vdash cv \Leftarrow^{\text{ht}} s$*. Then,*

1. *If* `aeval` $f\ b\ v_{\text{f}}\ v_{\text{t}}\ cv = a$*, then also* `interp` $gs\ f\ b\ s = a$*.*
2. *If* `aevalc` $(u, f)\ b\ v_{\text{f}}\ v_{\text{t}}\ cv = a$*, then also* `interpc` $gs\ (u, f)\ b\ s = a$*.*

Proof Part (1) can be shown by a simple structural induction on f:

- Case $f =$ `AtEnd`. If $s = [\,]$ (and hence $cv =$ `NONE`), both sides simplify to $a =$ `true`; otherwise, to $a =$ `false`.
- Case $f =$ `Expect` (c, i), where $i =$ `CN` n. If $s = [\,]$, we likewise have $cv =$ `NONE`, both sides simplify to `false`, and we are done. Otherwise, $s = c' :: s'$ and $cv =$ `SOME` (c', v'), where $gs \vdash v' \Leftarrow_{\texttt{true}} s'$. If $c' \neq c$, both sides again immediately simplify to `false`. Thus, it remains to consider the case $c' = c$, for which, using the property of v', we get `aeval` $f\ b\ v_{\text{f}}\ v_{\text{t}}$ (`SOME` $(c', v')) =$ `aevali` $i\ v' = v'[n] =$ `interpi` $gs\ i$ `true` $s' =$ `interp` $gs\ f\ b\ s$.
- Case $f =$ `Cont` (p, i), where $i =$ `CN` n. In the subcase where $p = b =$ `true`, we have $a =$ `aeval` f `true` $v_{\text{f}}\ v_{\text{t}}\ mc =$ `aevali` $i\ v_{\text{t}} = v_{\text{t}[n]}$, and so by assumption on v_{t}, also `interp` gs (`Cont` (`true`, i)) `true` $s =$ `interpi` $gs\ i$ `true` $s = a$. Otherwise (i.e., if $p =$ `false` and/or $b =$ `false`), we analogously get that both sides are equal to $v_{\text{f}[n]}$.
- Case $f =$ `Fail`: immediate, as both sides are `false`.
- Case $f =$ `Or` (f_1, f_2): follows straightforwardly from the IH on f_1 and f_2.

Part (2) follows immediately from part (1), for both possible values of u. ■

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Lemma 4.8 (Soundness of `astep`). *If* $gs \vdash cv \Leftarrow^{\mathrm{ht}} s$ *and* $\texttt{astep}\ (gs, i_1)\ cv = v$, *then* $gs \vdash v \Leftarrow_{\texttt{true}} s$

Proof From Lemma 4.7(2) we get that, whenever $gs \vdash v_\mathrm{f} \Leftarrow_{\texttt{false}} s$, $|v_\mathrm{f}| = n$, $gs \vdash n \hookrightarrow [g]$, and $\texttt{aevalc}\ g\ \texttt{false}\ v_\mathrm{f}\ [\,]\ cv = a$, then $a = \texttt{interpc}\ gs\ g\ \texttt{false}\ s = \texttt{interpi}\ gs\ (\texttt{CN}\ n)\ \texttt{false}\ s$, and thus also $gs \vdash v_\mathrm{f}@[a] \Leftarrow_{\texttt{false}} s$. Since (vacuously) $gs \vdash [\,] \Leftarrow_{\texttt{false}} s$, by induction on the derivation of $v_\mathrm{f} = \texttt{asf}\ gs\ [\,]$, we get $gs \vdash v_\mathrm{f} \Leftarrow_{\texttt{false}} s$. And hence, by an analogous argument for $v = \texttt{ast}\ gs\ v_\mathrm{f}\ [\,]$, we conclude $gs \vdash v \Leftarrow_{\texttt{true}} s$. ■

We also note, for future reference:

Lemma 4.9 (Dead states). *If* v_0 *is an all-*`false` *vector, and* `astep` *pgm* $(\texttt{SOME}\ (c, v_0)) = v$, *then* $v = v_0$.

Proof A couple of straightforward inductions. ■

Theorem 4.10 (Soundness of `arun`). *If* `arun` *pgm* $s = a$, *then also* `irun` *pgm* $s = a$.

Proof Let $pgm = (gs, i_1)$, where $i_1 = \texttt{CN}\ n_1$. We have $a = \texttt{aobs}\ pgm\ v = \texttt{aevali}\ i_1\ v = v_{[n_1]}$, where $v = \texttt{foldr}\ \cdots\ s$. By induction on s, using Lemma 4.8 for the both the base case ($s = []$, $cv = \texttt{NONE}$) and the step ($s = c :: s'$, $cv = \texttt{SOME}\ (c, v')$, $gs \vdash v' \Leftarrow_{\texttt{true}} s'$), we get that $gs \vdash v \Leftarrow_{\texttt{true}} s$. And from that, using Definition 4.6, we immediately get $\texttt{irun}\ pgm\ s = \texttt{interpi}\ gs\ i_1\ \texttt{true}\ s = v_{[n_1]} = a$. ■

Corollary 4.11. *For any* r *and* s, $\texttt{amatchtop}\ r\ s = \texttt{imatchtop}\ r\ s$.

Proof Follows immediately from Theorems 3.1, 4.5, and 4.10. ■

4.3 Perspectives

The transformation in this section effectively corresponds to the powerset construction (Rabin & Scott, 1959) for transforming NFAs to DFAs, but it is motivated by concerns of efficiency (avoiding recomputations), rather than decidability. As a notable difference, however, instead of computing the ε-closure for the *target* node(s) of each character transition (i.e., which NFA nodes can we get to after following a c-labeled edge from a given node?), we compute it on the *source* node(s) (i.e., from which nodes can we get to the given one by following a c-labeled edge?), corresponding to processing the input string from the end.

Also, the state machine we construct is not (yet) a classical transition-table automaton, in which each state corresponds to a *concrete* graph vertex in a data structure. Rather, determining the next state from the current one and the input character requires a full evaluation of all the computations in the program, at a per-step cost of $O(|r|)$. (On the other hand, the cost is not sensitive to the size of the input alphabet Σ, other than by assuming that two characters can be compared for equality in $O(1)$ time.)

Using memoization (or dynamic programming) for parsing-related tasks is not a new idea (Frost & Szydlowski, 1996), though it is more commonly used for parsing languages

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

more complicated than regular ones. In the present setting, however, the actual code needed to properly share continuation results becomes particularly simple and uniform.

As mentioned, the eager interpreter effectively processes the input string backward. This is very evident here, as `arun` uses `foldr`, not `foldl`, to traverse the string. To obtain a more direct correspondence to a traditional left-to-right automaton, we can reverse the input string first, exploiting the identity `foldr` $f\ a\ s =$ `foldl` $f\ a$ (`rev` s). If reversing the string is inconvenient or impractical, for example, because it is being incrementally read from a file, or produced in a streaming fashion, we can instead exploit that `rev` $s \in \mathcal{L}(r)$ iff $s \in \mathcal{L}(\overleftarrow{r})$, where the regex $\overleftarrow{r}$ is obtained by recursively swapping the order of the arguments r_1 and r_2 in all occurrences of `Seq` (r_1, r_2) in r. And this transformation can easily be merged into `transtop`, by simply exchanging r_1 and r_2 on the RHS of the defining equation for `trans` (`Seq` (r_1, r_2)).

5 Tabulating transitions

5.1 Changing state machine representations

The running time of the matcher so far is evidently linear in the length of the input string. However, processing each string character (i.e., evaluation of `astep`) takes time proportional to the size of the program (or the original regex), for a total running time of $O(|r| \cdot |s|)$.

There is one more speedup opportunity that we have not exploited yet. The type `state` of possible machine states, that is, $O(|r|)$-length boolean vectors, is evidently finite (though potentially quite large). Since the input alphabet $\Sigma =$ `char` is also finite (and, in SML, fairly small), this means that the state transition function $\Sigma \times$ `state` $\rightarrow$ `state` has a finite domain and can thus be precomputed as a lookup table.

In fact, we do not even need Σ to be finite: we can show that any input character that does not occur explicitly in the regex-derived program as an argument of at least one `Expect` (i.e., is not *relevant*) will cause the machine to transition to a dead state. Since there can be at most $O(|r|)$ relevant characters in an r, we do not have to worry about the underlying character set being potentially large (such as Unicode) or even infinite.

A collection of definitions for a table-driven state machine are shown in Figure 9. We introduce a distinct type `stateno` for *state numbers*, with `SN` 0 specifically representing the canonical dead state.

The function `pgmchars` *pgm* determines the set (represented as an ordered, duplicate-free list) of characters c occurring as `Expect` (c, i) anywhere in *pgm*. As a slight nod to efficiency, we calculate it by incrementally adding characters to an accumulator, rather than by repeated unions.

The type `charmap` represents finite maps from characters to state numbers, here represented for simplicity as plain association lists. The function `lookc` looks up a character in a map, returning `SN` 0 if not found; conversely, `mkcharmap` constructs a character map from equal-length lists of characters and corresponding state numbers, with only mappings to nonzero states explicitly included. Clearly, this minimal implementation is not practically efficient, as lookups take time linear in the domain of the map. A minimal

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

```
datatype stateno = SN of int

type charset = char list (* ordered, duplicate-free *)
type charmap = (char * stateno) list

type dfa = (charmap * ans) list * stateno

(* pgmchars : pgm -> charset *)
fun pgmchars (fs, _) =
  let fun insert c [] = [c]
        | insert c (cs as c' :: cs') =
            if c > c' then c' :: insert c cs'
            else if c = c' then cs else c :: cs
      fun inschars (Expect (c, _)) cs = insert c cs
        | inschars (Or (f1, f2)) cs = inschars f2 (inschars f1 cs)
        | inschars _ cs = cs
  in foldl (fn ((_, f), cs) => inschars f cs) [] fs end

(* lookc : char -> charmap -> stateno *)
fun lookc (c:char) [] = SN 0
  | lookc c ((c', j) :: cm) = if c = c' then j else lookc c cm

(* mkcharmap : charset -> stateno list -> charmap *)
fun mkcharmap [] [] = []
  | mkcharmap (c :: cs) (SN 0 :: js) = mkcharmap cs js
  | mkcharmap (c :: cs) (j :: js) = (c, j) :: mkcharmap cs js


(* dstep : dfa -> stateno nextc -> stateno *)
fun dstep (_, j1) NONE = j1
  | dstep (es, _) (SOME (c, SN m)) =
      let val (cm, _) = nth (es, m) in lookc c cm end

(* dobs : dfa -> stateno -> ans *)
fun dobs (es, _) (SN m) = let val (_, a) = nth (es, m) in a end

(* drun : dfa -> char list -> ans *)
fun drun dfa s =
  let val j = foldr (fn (c,j) => dstep dfa (SOME (c,j))) (dstep dfa NONE) s
  in dobs dfa j end
```

Fig. 9. Table-driven state machine.

realistic alternative would be to take `charmap = (char * stateno) vector`, with the characters ordered, so that `lookc` could perform a binary, instead of linear, search.

With those preliminary definitions out of the way, the actual state machine definition is tiny. A `dfa` consists of a state table (again inefficiently represented as a list, for uniformity), where each entry contains a character map and a boolean value indicating whether this state is accepting. Also, the machine description includes a designated starting state (typically, but not necessarily, `SN 1`). The functions `dstep`, `dobs`, and `drun` are then the direct analogs of `astep`, `aobs`, and `arun` for the new representation.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

5.2 Correctness of representation change

Let us first summarize what we will need to know about relevant characters and character maps.

Lemma 5.1 (Correctness of relevant-character collection)**.** *For any pgm,* `pgmchars` *pgm returns the set (as an ordered list) of all characters c occurring as* `Expect` (c, i) *anywhere in pgm.*

Proof We first show, by induction on *cs*, that the call `insert` *c cs* correctly adds *c* to the ordered list *cs*; then the result follows by straightforward structural induction over *pgm*. ■

Lemma 5.2 (Correctness of character maps)**.** *Assume cs is strictly increasing, and* $|js| = |cs|$*. Then cm* = `mkcharmap` *cs js is defined; and for any c, if* $c = cs[k]$ *for some (necessarily unique) k, then* `lookc` *c cm* $= js[k]$*; otherwise (i.e., if c is not in cs),* `lookc` *c cm* = `SN` 0*.*

Proof Straightforward induction on the structure of *cs*. ■

Lemma 5.3 (Failure for irrelevant characters)**.** *If c does not occur in pgm, and* `astep` *pgm* (`SOME` $(c, v)) = v'$*, then* $v' = v_0$ *(the all-false vector).*

Proof Straightforward induction on the evaluation derivation, noting that since we are evidently not at the end of the string, the only way for `aeval` to return a `true` value (possibly to be subsequently propagated to other vector positions) is when *c* is the same as a c_0 occurring as an `Expect` (c_0, i) in *pgm*. ■

For the more interesting parts, we note that state tabulation is merely an extreme case of *representation change* for state machines.

Definition 5.4 (State machine)**.** *A* state machine *with input alphabet* Σ *and output set A is a quadruple* $\mathcal{M} = (Q, q_1, \delta, \alpha)$*. Here Q is the* state set*;* $q_1 \in Q$ *is the* initial state*;* $\delta : \Sigma \times Q \to Q$ *is the* transition function*; and* $\alpha : Q \to A$ *is the* observation function*. Any such a machine evidently determines a function* $run_{\mathcal{M}} : \Sigma^* \to A$*, given by* $run_{(Q,q_1,\delta,\alpha)}\ [c_1, ..., c_n] = \alpha($`foldr` $\delta\ q_1\ [c_1, ..., c_n]) = \alpha(\delta(c_1, ..., \delta(c_n, q_1)))$.

The set A can be just the booleans, representing acceptance/rejection, but it could also be a more involved observation, such as a parse tree or other output. We do not need to require that Q and Σ are finite. Note that, for consistency with the development so far, we are still processing the input string right-to-left.

Definition 5.5 (State machine simulations)**.** *A* simulation *of a state machine* $\mathcal{M}$ *by another* $\mathcal{M}'$ *(with the same input alphabet and output set) is witnessed by a partial function* $h : Q \rightharpoonup Q'$ *such that:*

1. $h(q_1) = q'_1$.
2. *If* $h(q) = q'$*, then for all* $c \in \Sigma$, $h(\delta(c, q)) = \delta'(c, q')$.
3. *If* $h(q) = q'$*, then* $\alpha(q) = \alpha'(q')$.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Note that h need not be injective; that is, multiple states in Q can correspond to the same state in Q'. Also, not every state in Q needs to have a counterpart in Q'; however, all the *reachable* ones do.

Lemma 5.6 (Correctness of simulation)**.** *If there exists a simulation of $\mathcal{M}$ by $\mathcal{M}'$, then for any $s \in \Sigma^*$, $run_{\mathcal{M}}\ s = run_{\mathcal{M}'}\ s$.*

Proof This follows directly from Definition 5.5. Let $q = \texttt{foldr}\ \delta\ q_1\ s$. By induction on s (using property (1) for [ ] and (2) for ::), we get that $\texttt{foldr}\ \delta'\ q_1'\ s = q'$ for some q' with $h(q) = q'$. Then, by (3), we get $run_{\mathcal{M}}\ s = \alpha(q) = \alpha'(q') = run_{\mathcal{M}'}\ s$. ■

In concrete applications, we will generally not work with arbitrary transition and observation functions, but only with ones that are derived uniformly from some *finitary* description of the machine. We also fix $\Sigma = \texttt{char}$ and $A = \texttt{ans} = \texttt{bool}$. Then, for our two state machine families, we take

Definition 5.7 (Concrete state machines)**.** *For any ordered $pgm = (gs, i_1)$, we define the machine $\mathcal{A}(pgm) = (V, v_1, \delta, \alpha)$, where $V = \{v \in \texttt{state} \mid |v| = |gs|\}$, $v_1 = \texttt{astep}\ pgm\ \texttt{NONE}$, $\delta(c, v) = \texttt{astep}\ pgm\ (\texttt{SOME}\ (c, v))$, and $\alpha(v) = \texttt{aobs}\ pgm\ v$. Similarly, for any $dfa = (es, j_1)$, we take $\mathcal{D}(dfa) = (J, j_1, \delta', \alpha')$, where $J = \{\texttt{SN}\ j \in \texttt{stateno} \mid 0 \leq j < |es|\}$, and δ' and α' are derived analogously from* `dstep` *and* `dobs`*.*

Note that the description-interpreting functions only need to be defined for elements of the corresponding state set, not all elements of its containing SML type. For example, `aobs` *pgm v* is allowed to crash if v has the wrong length wrt. *pgm*.

5.3 Constructing transition tables

Our final task will be to obtain a lookup-table-based state machine, with associated simulation, from the boolean-vector one. The basic idea is to incrementally construct the components of $\mathcal{M}'$ and h together, starting from the initial state of $\mathcal{M}$, and extending to all states reachable from it.

The code is in Figure 10. The key data structure here is D `trie`, which compactly represents partial maps from boolean vectors to D using binary trees. To look up a vector v in a trie t, we examine v's bits in order, traversing into the t_t or t_f subtrie (for `true` and `false`, respectively) of $t = \texttt{Node}\ (t_t, t_f)$, until we either encounter a `Leaf` d with the result, or `Empty`. Note that we will only use a t for vectors all of the same length n, so all `Leaf` d-leaves will be found at depth n in t (where the root is at depth 0). On the other hand, we may encounter an `Empty` leaf after only traversing a prefix of the vector, if the trie contains no mappings for that prefix; in particular, a proper trie will never contain a subtree of the form `Node (Empty, Empty)`, which is equivalent to just `Empty`.

The function `accesst` is used for both trie lookups and updates: `accesst` $t\ v\ d$ returns either `Found` d', if t contains a mapping of v to d'; or `Added` t' if t did not contain a mapping for v, but t' extends t with a mapping of v to d.

The function `trav` then constructs, in a depth-first manner, the transition table, as well as the trie witnessing the simulation. A typical call has the form $(j, es, t', m') =$

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

```
datatype 'd trie = Empty | Leaf of 'd | Node of 'd trie * 'd trie

datatype 'd accres = Found of 'd | Added of 'd trie

(* accesst : 'd trie -> ans list -> 'd -> 'd accres *)
fun accesst Empty [] d = Added (Leaf d)
  | accesst Empty (a :: v) d = upd (accesst Empty v d) a Empty
  | accesst (Leaf d') [] d = Found d'
  | accesst (Node (tt, tf)) (true :: v) d = upd (accesst tt v d) true tf
  | accesst (Node (tt, tf)) (false :: v) d = upd (accesst tf v d) false tt
(* upd : 'd accres -> bool -> 'd trie -> 'd accres *)
and upd (Found d') _ _ = Found d'
  | upd (Added tt') true tf = Added (Node (tt', tf))
  | upd (Added tf') false tt = Added (Node (tt, tf'))

(* trav : charset -> pgm -> state -> stateno trie -> int ->
           stateno * (charmap * ans) list * stateno trie * int *)
fun trav cs pgm v t m =
  case accesst t v (SN m) of
    Found j => (j, [], t, m)
  | Added t0 =>
      let val vs = map (fn c => astep pgm (SOME (c, v))) cs
          val (js, es1, t1, m1) = travl cs pgm vs t0 (m+1)
      in (SN m, (mkcharmap cs js, aobs pgm v) :: es1, t1, m1) end
(* travl : charset -> pgm -> state list -> stateno trie -> int ->
            stateno list * (charmap * ans) list * stateno trie * int *)
and travl cs pgm [] t m = ([], [], t, m)
  | travl cs pgm (v :: vs) t m =
     let val (j, es1, t1, m1) = trav cs pgm v t m
         val (js, es2, t2, m2) = travl cs pgm vs t1 m1
     in (j :: js, es1 @ es2, t2, m2) end

(* mkdfa : pgm -> dfa *)
fun mkdfa pgm =
  let val cs = pgmchars pgm
      val v0 = map (fn _ => false) (#1 pgm)
      val v1 = astep pgm NONE
      val ([SN 0, j1], es, t, m) = travl cs pgm [v0, v1] Empty 0
  in (es, j1) end

(* dmatchtop : re -> char list -> ans *)
fun dmatchtop r s = drun (mkdfa (transtop r)) s
```

Fig. 10. Conversion to table-driven machine.

`trav` *cs pgm v t m*. Here, the input *cs* is the set (ordered list) of relevant characters to consider for character maps, and *pgm* is the state machine (these two never change); *v* is some state; *t* is the current trie; and *m* is an allocation counter for state numbers. In the output tuple, *j* is the state number corresponding to *v*; *es* is a list of new entries to be added to the table from position *m*; t' is the possibly updated trie; and m' is the new allocation counter, so that $m' = m + |es|$. If the given state *v* was already present in *t*, `trav` just returns its corresponding state number; but if *v* needs to be added, we allocate a new

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

state number for it, compute its character map (enumerating all states reachable from v by a single `astep` with a character from cs), and generate a table entry for it.

The auxiliary function `travl` is simply the "effectful" list extension of `trav`: instead of a single vector v, it takes a list vs; and instead of a single result j, it returns a list js, with $|js| = |vs|$. Moreover, t and m are threaded through the computation in a state-like manner, while the output es is accumulated by concatenation.

Finally, `mkdfa` converts a complete state machine to its transition-table representation. It first allocates the permanent-failure state as `SN 0`, and then finds the state number of the initial state, which will normally be `SN 1` (because the initial continuation accepts the empty input and thus the state contains at last one true bit), but as a "side effect" it also computes the transition table for all states reachable from the initial one. Then, `dmatchtop` is the final matcher.

The state labels v_j in Figure 8 represent precisely order in which the DFA entries are created. In particular, v_2 is created before v_3, and v_4 before v_5, because a < b. The dotted nodes are never reached in the traversal, and so are not assigned state numbers.

5.4 Correctness of conversion

We first need some simple results about tries.

Definition 5.8 (Trie extension)**.** *A trie t'* extends *t, written $t \sqsubseteq t'$, if t' arises by replacing some* `Empty` *nodes in t with potentially larger subtries. More formally, $\sqsubseteq$ is the least relation on tries satisfying*

1. $\texttt{Empty} \sqsubseteq t'$.
2. $\texttt{Leaf}\ d \sqsubseteq \texttt{Leaf}\ d$.
3. $\texttt{Node}\ (t_\mathrm{t}, t_\mathrm{f}) \sqsubseteq \texttt{Node}\ (t_\mathrm{t}', t_\mathrm{f}')$ *if* $t_\mathrm{t} \sqsubseteq t_\mathrm{t}'$ *and* $t_\mathrm{f} \sqsubseteq t_\mathrm{f}'$.

It is immediate to verify that $\sqsubseteq$ is a preorder (reflexive and transitive).

In order to show that we cannot keep extending a trie forever, we define

Definition 5.9 (Trie free space)**.** *For an n-level trie t, the count $\|t\|_n$ of unused slots in t is defined by induction on n:*

$$\begin{array}{rclcrcl} \|\texttt{Empty}\|_0 & = & 1 & \qquad & \|\texttt{Empty}\|_{n+1} & = & 2 \cdot \|\texttt{Empty}\|_n \\ \|\texttt{Leaf}\ d\|_0 & = & 0 & & \|\texttt{Node}\ (t_\mathrm{t}, t_\mathrm{f})\|_{n+1} & = & \|t_\mathrm{t}\|_n + \|t_\mathrm{f}\|_n \end{array}$$

(Note that there are no equations for $\|\texttt{Leaf}\ d\|_{n+1}$ or $\|\texttt{Node}\ (t_\mathrm{t}, t_\mathrm{f})\|_0$, as those would not be well-formed n-level tries.)

The trie-access function then satisfies a couple of straightforward properties:

Lemma 5.10 (Correctness of tries)**.** *Let t be an n-level trie, and v a length-n vector. Then:*

1. $\texttt{accesst}\ t\ v\ d$ *is defined.*
2. *If* $\texttt{accesst}\ t\ v\ d = \texttt{Found}\ d'$ *and* $t \sqsubseteq t'$, *then also* $\texttt{accesst}\ t'\ v\ d'' = \texttt{Found}\ d'$.
3. *If* $\texttt{accesst}\ t\ v\ d = \texttt{Added}\ t'$, *then* $t \sqsubseteq t'$, *and* $\texttt{accesst}\ t'\ v\ d' = \texttt{Found}\ d$.
4. *If* $\texttt{accesst}\ t\ v\ d = \texttt{Added}\ t'$, *then* $\|t'\|_n = \|t\|_n - 1$.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Proof Each part can be shown by a simple induction on t. ■

A trie t determines a partial function $t^\dagger$ from states to trie data, given by $t^\dagger(v) = d$ if `accesst` $t\ v\ d_0 =$ `Found` d (for some fixed d_0), and $t^\dagger(v)$ undefined otherwise. (By property (2), the choice of d_0 does not matter.)

Lemma 5.11 (Invariants for conversion)**.** *If* `trav` *cs pgm* v t $m = (j, es, t', m')$, *or* `travl` *cs pgm vs* t $m = (js, es, t', m')$, *then*

1. $m' = m + |es|$
2. $t \sqsubseteq t'$
3. *If* $t^\dagger$ *is injective and onto* m *(i.e., its range is the set* $\{\texttt{SN}\ m_j \mid 0 \leq m_j < m\}$*), then* $t'^\dagger$ *is injective and onto* m'.

Proof Each part can be proved by a straightforward (mutual) induction on the evaluation derivations. Briefly:

1. We see that m is incremented precisely when `trav` generates a new table entry; `travl` does not emit any table entries on its own, but only through `trav`.
2. We use Lemma 5.10(3), as well as reflexivity and transitivity of $\sqsubseteq$.
3. In `trav`, for the call `accesst` $t\ v$ (`SN` m), we have assumed that `SN` m is not in the range of $t^\dagger$, so that if t gets extended to t_0, $t_0^\dagger$ will still be injective, but now onto $m + 1$. The function `travl` only modifies t through `trav`. ■

Note that we do not in general need $t^\dagger$ to be injective, or to have any particular range, for it to witness a simulation, but we will use those properties for showing correctness of our simple conversion algorithm.

Definition 5.12 (Entry soundness)**.** *Let cs, pgm, and t be given. A transition table entry* $e = (cm, a)$ *is t-sound for state number j if there exists some v, with* $t^\dagger(v) = j$*, such that:*

1. `aobs` *pgm* $v = a$.
2. *For all c in cs, if* `lookc` $c\ cm = j'$*, then* `astep` *pgm* (`SOME` (c, v)) $= v'$ *for some* v' *with* $t^\dagger(v') = j'$.

A list es is t-sound for position m if its constituent entries are sound for state numbers `SN` m, `SN` $(m + 1)$, ..., `SN` $(m + |es| - 1)$*, respectively. It is immediate to check that, if* es_1 *is sound for m, and* es_2 *is sound for* $m + |es_1|$*, then* $es_1 @ es_2$ *is also sound for m.*

Lemma 5.13 (Soundness of `trav` and `travl`)**.** *The traversal functions are correct in the following sense:*

a. If `trav` *cs pgm* v t $m = (j, es, t', m')$*, and* $t' \sqsubseteq t''$*, then es is* t''*-sound for position m, and* $t''^\dagger(v) = j$.
b. If `travl` *cs pgm vs* t $m = (js, es, t', m')$*, and* $t' \sqsubseteq t''$*, then es is* t''*-sound for position m, and* `map` $t''^\dagger$ $vs = js$.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Proof The two parts are proved by mutual induction on the evaluation derivations:

a. There are two subcases. If `accesst` t v (`SN` m) = `Found` j, then $es = [\,]$, which is vacuously t''-sound; and since $t' = t$, we have $t \sqsubseteq t''$, so also $t''^{\dagger}(v) = j$ by Lemma 5.10(2).

   Otherwise, `accesst` t v (`SN` m) = `Added` t_0, where $t \sqsubseteq t_0$, and we have $j =$ `SN` m and $es = (cm,$ `aobs` pgm $v) :: es_1$, where $cm =$ `mkcharmap` cs js, $(js, es_1, t', m') =$ `travl` cs pgm vs t_0 $(m+1)$, and for all $k < |cs|$, $vs_{[k]} =$ `astep` pgm (`SOME` $(cs_{[k]}, v)$). Since $t_0 \sqsubseteq t'$ (Lemma 5.11(2)) and $t_0^{\dagger}(v) = j$ (Lemma 5.10(3)), also $t''^{\dagger}(v) = j$. From IH(b) on the `travl`-evaluation, we get that es_1 is t''-sound for $m+1$, so it remains to show that (`mkcharmap` cs js, `aobs` pgm v) is t''-sound for m. The second component satisfies the requirement by definition; and for the first, let $c = cs_{[k]}$ for some $k < |cs|$. If `lookc` c $cm = j'$, then by Lemma 5.2, we have $j' = js_{[k]} =$ (`map` $t''^{\dagger}$ $vs)_{[k]} = t''^{\dagger}(vs_{[k]}) = t''^{\dagger}$(`astep` pgm (`SOME` (c, v))), again as required.

b. Follows by pointwise application of IH(a) for the elements of vs, using transitivity of $\sqsubseteq$ and the concatenation property of t''-sound entry sequences. ■

Theorem 5.14 (Correctness of conversion)**.** *For any ordered pgm, we have* `mkdfa` $pgm =$ *dfa for some dfa such that* $\mathcal{D}(dfa)$ *simulates* $\mathcal{A}(pgm)$.

Proof Let $pgm = (gs, i_1)$ and $n = |gs|$, We first note that $cs =$ `pgmchars` pgm is always defined (Lemma 5.1), as is `astep` pgm v for all v with $|v| = n$ (Lemma 4.4). Next, we observe that the mutually recursive functions `trav` and `travl` necessarily terminate by the measure $\|t\|_n$ on their t-parameters, because `trav` either returns immediately or invokes `travl` with a t_0 that has one less free slot than t (Lemma 5.10(4)); and that function simply invokes `trav` back exactly $|cs|$ times, threading the trie sequentially through all calls. Thus, the computation terminates after at most $\|\texttt{Empty}\|_n \cdot |cs| = 2^n \cdot |cs| = O(2^{|r|} \cdot \min(|\Sigma|, |r|))$ steps.

In particular, we can safely compute $(js, es, t, m) =$ `travl` cs pgm $[v_0, v_1]$ `Empty` 0, where v_0 is all-false of length n, and $v_1 =$ `astep` pgm `NONE`. By Lemma 5.13(b) (and reflexivity of $\sqsubseteq$), we get that es is t-sound for position 0, and $js = [$`SN` $0, j_1]$, where $j_1 = t^{\dagger}(v_1)$. Thus, the pattern-matching `val`-binding in `mkdfa` will succeed, and $dfa = (es, j_1)$, so condition (1) of Definition 5.5 (with $h = t^{\dagger}$) is immediate.

For the remaining two, let v be given such that $t^{\dagger}(v) = j =$ `SN` m_j. Then by t-soundness of es, we have $es[m_j] = (cm, a)$, where the entry (cm, a) is t-sound for m_j. By injectivity of $t^{\dagger}$ (Lemma 5.11(3), since `Empty`† is clearly injective and onto 0), the v_j in the definition of t-soundness must be the same as v, so in particular, `aobs` pgm $v = a =$ `dobs` dfa j, giving us condition (3).

For condition (2), let c be arbitrary, and let $v' =$ `astep` pgm (`SOME` (c, v)) and $j' =$ `dstep` dfa (`SOME` (c, j)) = `lookc` c cm. We must show that $t^{\dagger}(v') = j'$. If c is in cs, this follows from the condition on cm in t-soundness; and otherwise, by Lemma 5.3, we know that $v' = v_0$; and by Lemma 5.2 that $j' =$ `SN` 0, and so again $t^{\dagger}(v_0) =$ `SN` $0 = j'$. ■

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Corollary 5.15. *For any r and s,* `dmatchtop` *r s* = `amatchtop` *r s.*

Proof On both sides, we compute *pgm* = `transtop` *r*, which we know is defined (Theorem 3.1), and *pgm* is ordered (Lemma 4.2). Further, on the LHS, we have a *dfa* = `mkdfa` *pgm*, and there is a simulation of $\mathcal{A}$(*pgm*) by $\mathcal{D}$(*dfa*) (Theorem 5.14). Finally, by Lemma 5.6, `amatchtop` *r s* = `arun` *pgm s* = `drun` *dfa s* = `dmatchtop` *r s*. ■

5.5 DFA minimization

An optional final step in a regex-matching implementation consists of *minimizing* the constructed DFA, by collapsing indistinguishable states. For example, in the DFA of Figure 8, we observe that the states labeled v_3 and v_5 are equivalent, in that both are non-accepting, and both transition to v_4 on a and to one of each other on b.

A number of algorithms for DFA minimization are known, notably those of Brzozowski (1962) and Hopcroft (1971). However, while a minimization step may make the DFA more compact, sometimes significantly so, it does not materially affect the runtime efficiency of the matcher (other than by second-order effects due to better cache utilization). Thus, we will not formally consider minimization further here.

Still, it is worth remarking that, with our representations of programs and automata, Brzozowski's construction in particular can be expressed exceptionally concisely. Recall that, in a traditional presentation, it consists of *reversing* all the edges in the DFA (yielding, in general, an NFA, because a DFA node may still have multiple *incoming* edges labeled by the same character), making the initial state of the DFA an accepting state of the NFA, and making all the DFA accepting states initial in the NFA (by adding suitable ε-transitions from a single initial state, if necessary). Not very surprisingly, this yields an NFA for the reversed language; that is, a string is accepted by the newly constructed NFA precisely when its reverse is accepted by the original DFA. The unexpected part is that converting this NFA back to a DFA (using the usual powerset construction) actually yields a *minimal* DFA for the reversed language; that is, any two distinct states demonstrably lead to different acceptance/rejection outcomes for at least one string, and so cannot be collapsed.

The same construction turns out to work very smoothly for our analogs of NFAs and DFAs: because the conversion from matcher program to state machine inherently reverses the conceptual direction of the edges, we can *directly* convert a character map in the tabulated automaton to a computation with an `Or`-linked list of `Expect`-nodes. The endpoint of each such list becomes `AtEnd` if the automaton entry was accepting, and `Fail` otherwise. DFA transitions into the designated dead state, which are not explicitly represented in the character maps, are not included in the program, either. No `Cont`-references are needed, and all computations are unconditional. Finally, the initial state of the DFA becomes the *main* continuation of the program.

The resulting code is shown in Figure 11. The function `mkpgm` performs the conversion, and `minrevdfa` implements the full pipeline from regex to minimal automaton. Note that the returned automaton, nominally for the reversed language of the regex, can actually be seen as a conventional, left-to-right (i.e., `foldl`-based) DFA for the original language.

An example of the construction is shown in Figure 12. Here, we have taken the (non-dotted part of) the DFA in Figure 8 and viewed it directly as a program, with the same interpretation of the graph nodes as in Figure 5. From the graph, we can directly read off

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

```
(* mkpgm : dfa -> pgm *)
fun mkpgm (es, SN n1) =
  let fun mkcomp (cm, a) =
            foldl (fn ((c, SN n), f) => Or (Expect (c, CN n), f))
                  (if a then AtEnd else Fail)  cm
  in (map (fn e => (true, mkcomp e)) es, CN n1) end

(* minrevdfa : re -> dfa *)
fun minrevdfa r = mkdfa (mkpgm (mkdfa (transtop r)))

(* eqvre, subre : re -> re -> bool *)
fun eqvre r1 r2 = minrevdfa r1 = minrevdfa r2
fun subre r1 r2 = eqvre (Alt (r1,r2)) r2
```

Fig. 11. Constructing minimal DFAs by Brzozowski's method.

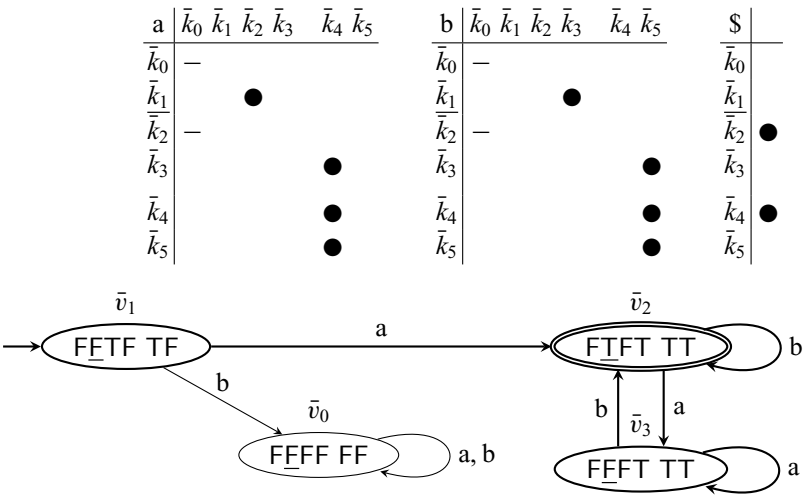


Fig. 12. Transition matrices, (reachable) state space, and DFA labeling for running example.

the transfer matrices for the individual characters, and the initial vector, as shown in the top part of the figure, where we have adopted the position numbering $\bar{k}_i = v_i$. (The matrix entries marked with a – represent the transitions to the canonical dead state v_0 in the DFA. They can be safely omitted in the conversion from character maps to computations, as the continuation-result vector will always have a false entry in position 0.) Note that the computations contain no `Cont`-references and are hence trivially ordered. (However, unlike for the directly regex-derived transfer matrices, the ●-entries are *not* confined to lie strictly below the main diagonal.)

The (reachable) nodes of the state space and the transitions between them are then shown in the bottom part of the figure, and the labels $\bar{v}_0$ through $\bar{v}_3$ reflect what the depth-first, alphabetical-order traversal assigns in the tabulation. Inspecting the resulting DFA, we see that it precisely recognizes, in an evidently minimal way, the language of the original RE: a single a, followed by zero or more a's and/or b's, but where the last one must be a b.

However, the arguably main significance of DFA minimization lies not in the marginal improvement in efficiency or space savings it may provide, but in that the minimal DFA

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

for any regular language is actually *unique* up to labeled graph isomorphism (including the edge labels, and the initial and accepting nodes). That is, two regular expressions denote the same language precisely when their minimal automata can simulate each other. And since our tabulation algorithm traverses the state space of an automaton in a canonical way (depth-first, with outgoing edges processed in alphabetical order), the tabulated representations of isomorphic graphs are actually *identical*.

This is exploited in the function `eqvre`, which decides whether two regexes denote the same language. (The reversal inherent to the construction is not a problem, because *both* languages are reversed before the comparison.) Using the equivalence tester, we can also easily decide whether one regex's language is *included* in the other's, by exploiting the set-theoretic equivalence $L_1 \subseteq L_2 \Leftrightarrow L_1 \cup L_2 = L_2$), this is codified in function `subre`.

Still, since we have not rigorously proved the correctness of the construction (which, despite the deceptive simplicity of the code, is not a trivial undertaking), we will not state any of the above as formal theorems at this time. It would be interesting to consider, however, whether the correctness of Brzozowski's algorithm (or another, possibly more efficient one) can also usefully be argued from a functional-programming/semantic perspective, rather than from a classical automata-theoretic one.

5.6 Perspectives

The incremental construction of a transition-table DFA from a more implicit specification of the transitions between the states is a common idiom, though here we particularly emphasize that the set of observations can be more complex than the booleans (corresponding to accepting/rejecting states), and that an explicitly tabulated version is just an extreme representation of the state machine, for minimizing transition times. Usually there is a trade-off between runtime performance, and the work (and space) needed for precomputation, and the general formulation of the simulation easily accounts for hybrid representations, such as caching (in runtime-mutable memory) the character maps for oft-visited DFA states, and possibly also evicting them as needed, if the cache grows too large.

We have picked a very simple representation of character maps, which (with the already mentioned binary-search improvement) is fine for applications where `Char` *c* regexes only match specific characters – but probably not larger character *classes*, such as all digits or uppercase letters, for which explicitly listing all members of a class in character maps may incur a substantial space penalty. In many such cases, a slight generalization of the maps to also allow compact representation of contiguous character *ranges* mapping to the same state should suffice, though. The refinements to regexes with general character-class leaves considered by Owens *et al.* (2009) in the context of Brzozowski derivatives might also be applicable here.

6 Conclusions

6.1 Summary and potential directions for future work

We have seen how to derive an efficient, table-driven matcher for regular expressions, by a series of relatively simple, meaning-preserving program transformations starting from

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Harper's (incomplete) CPS-based matcher – using general techniques such as cycle detection, defunctionalization, specialization, change of evaluation strategy, and tabulation of finite functions. While the net result we obtain is essentially the same as with classical methods rooted in formal languages and automata theory, the abstractions we use are somewhat different, and many standard constructions are "sliced" differently.

To keep the development focused, we have explicitly only considered the problem of deciding membership in the language of a regex. However, there is reason to hope that, by exposing some of the underlying algorithmic and program-transformation principles, the results would prove more robust, modular, and easily generalizable than existing, highly tuned solutions for specific problems. For example, we might consider one or more of:

- generalizing the task from yes/no recognition of whole strings to various notions of regex parsing (disambiguated as, e.g., greedy) or transduction.
- generalizing the language description from regular expressions to, for example, regular grammars, or extended regular expressions (with intersection and/or complementation).
- generalizing the languages recognized from regular to larger classes, such as (fragments of) Perl-compatible regular expressions (including, e.g., backreferences), parsing expression grammars (PEGs), (possibly limited) context-free grammars, or beyond; or considering approximate or error-tolerant matching.
- generalizing the input data type from linear strings to trees or graphs.
- generalizing the computation model from sequential to control and/or data parallel, for improving performance on modern hardware.

Many of these are already readily covered by existing methods and results, but often varying only one parameter at a time, not several at once; and it is often far from obvious how to combine *multiple* generalizations along different dimensions, when their underlying algorithmic principles are obscured.

The development also hopefully serves to attest to the naturalness and power of functional programming as a framework for reasoning about algorithm derivation: we were able to express, concisely and uniformly, all the relevant invariants and relationships formally connecting the set-theoretic semantics of regular expressions to the efficient, table-driven recognizer; and the relevant theorems can all be shown by elementary means.

6.2 Notes on the formalization

As mentioned, all theorems and their proofs have been fully formalized and verified in Twelf. While computations on strings are not an ideal application domain for this proof assistant (in particular, we are not playing to its strengths in reasoning about variable bindings and substitutions, while still suffering its weaknesses in expressing equational-reasoning arguments concisely), it proved quite adequate for the task. All the required lemmas and theorems are (or can easily be restated as) Π^0_1-formulas (i.e., of the form $\forall...\exists...$) about finite trees, and the proofs are by simple inductions with (sometimes lexicographic) subtree orderings. A few lemmas, especially the ones involving the higher-order CPS matcher, could probably be formulated and proved a bit more smoothly with deeper quantifier alternations and/or higher-order logic features, as readily available in many current proof assistants, but we easily manage even without that.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

As Twelf developments are effectively "blank-slate," explicitly defining everything (including basic arithmetic in Peano style) from scratch, and with essentially no automation of the proof itself (beyond *checking* that all cases are covered, and that inductions are well founded with respect to a subterm ordering), one might be concerned about the size of the formalized proofs. And indeed, the full formalization is of non-negligible, but still very manageable size. The executable content of the constructions, some 250 lines of SML code in this article, expands to about 900 lines of Twelf code, largely due to the (near-mechanical) transcription of the functional code into a significantly more verbose logic-programming style (but *not* exploiting backtracking or two-sided unification), and the need to hand-implement a number of primitive types and functions. The correctness proofs come to about 3400 additional lines, again including a number of trivial and generic/boilerplate lemmas about list concatenation being associative, character comparisons being decidable, etc., all of which are routinely taken for granted in even a semi-formal proof.

Almost certainly, essentially the same proofs could be expressed significantly more compactly in Coq or Agda, especially with some rudimentary automation of the routine tasks. (Or, alternatively, the development itself could be expressed in a dependently typed language, optionally followed by extraction of the executable content into a separate program, for efficient execution.) On the other hand, the raw size of the formalization does give a useful indication of the intrinsic conceptual complexity of the constructions.

Acknowledgments

I would like to sincerely thank the referees for their insightful comments, and the editorial team behind the Bob Harper Festschrift issue for their patience.

Also, it has been a pleasant surprise how much of Bob's research and teaching has – directly or indirectly – proved influential or inspirational in preparing this article. This includes not only his suggestion of the CPS-based regex matcher as an object of formal study, but also his work on the theoretical underpinnings of the tools used (Standard ML and Twelf), and his general, infectious enthusiasm for functional programming and applied semantics (Harper, 2012). I am very happy to (much belatedly) congratulate him on his 60th birthday!

Conflicts of Interest

None.

References

Berry, G. & Sethi, R. (1986) From regular expressions to deterministic automata. *Theor. Comput. Sci.* **48**, 117–126.

Brzozowski, J. A. (1962) Canonical regular expressions and minimal state graphs for definite events. In *Proceedings of the Symposium on Mathematical Theory of Automata*. MRI Symposia Series 12. Polytechnic Institute of Brooklyn, pp. 529–561.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press

Consel, C. & Danvy, O. (1989) Partial evaluation of pattern matching in strings. *Inform. Process. Lett.* **30**(2), 79–86.

Danvy, O. & Filinski, A. (1990) Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160.

Filinski, A. (1999) Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* pp. 175–188.

Frisch, A. & Cardelli, L. (2004) Greedy regular expression matching. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004*. LNCS 3142, pp. 618–629.

Frost, R. A. & Szydlowski, B. (1996) Memoizing purely functional top-down backtracking language processors. *Sci. Comput. Program.* **27**(3), 263–288.

Harper, R. (1999) Proof-directed debugging. *J. Function. Program.* **9**(4), 463–469.

Harper, R. (2012) *Practical Foundations for Programming Languages*. Cambridge University Press.

Harper, R., Honsell, F. & Plotkin, G. (1993) A framework for defining logics. *J. ACM* **40**(1), 143–184.

Hopcroft, J. E. (1971) An $n \log n$ algorithm for minimizing the states in a finite automaton. In *The Theory of Machines and Computations*, Kohavi, Z. (ed). Academic Press, pp. 189–196.

Jones, N. D. (1997) *Computability and Complexity: From a Programming Perspective*. MIT Press.

Jones, N. D., Gomard, C. K. & Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation*. Prentice Hall International.

Knuth, D. E., Morris, J. H. & Pratt, V. R. (1977) Fast pattern matching in strings. *SIAM J. Comput.* **6**(2), 323–350.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997) *The Definition of Standard ML*. Revised edn. The MIT Press.

Nielsen, L. & Henglein, F. (2011) Bit-coded regular expression parsing. In *International Conference on Language and Automata Theory and Applications, LATA 2011*. LNCS 6638, pp. 402–413.

Okasaki, C. (1995) Purely functional random-access lists. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture, FPCA 1995*. ACM, pp. 86–95.

Owens, S., Reppy, J. & Turon, A. (2009) Regular-expression derivatives re-examined. *J. Function. Program.* **19**(2), 173–190.

Pfenning, F. & Schürmann, C. (1999) System description: Twelf — a meta-logical framework for deductive systems. In *Automated Deduction – CADE-16*. LNCS 1632, pp. 202–206.

Rabin, M. O. & Scott, D. (1959) Finite automata and their decision problems. *IBM J. Res. Dev.* **3**(2), 114–125.

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference* pp. 717–740. Reprinted, with a foreword, in *Higher-Order and Symbolic Computation* **11**(4), 363–397.

Thompson, K. (1968) Programming techniques: Regular expression search algorithm. *Commun. ACM* **11**(6), 419–422.

Yi, K. (2006) 'Proof-directed debugging' revisited for a first-order version. *J. Function. Program.* **16**(6), 663–670.

https://doi.org/10.1017/S0956796820000295 Published online by Cambridge University Press