AUTOMATED PATTERN DETECTION IN HASKELL

CARL S. MCTAGUE AND JAMES P. CRUTCHFIELD

Below are complete implementations of Algorithms 2 and 4 from the paper Automated Pattern Detection: An Algorithm for Constructing Optimally Synchronizing Multi-Regular Language Filters [1]. We use the programming language Haskell, which represents the state of the art in polymorphically typed, lazy, purely functional programming language design [2]. Haskell compilers and interpreters are freely available for almost any computer (see www.haskell.org).


We will require the following standard modules:

module APD where
import List
import Maybe
import Array

REPRESENTING AUTOMATA

We emulate our exposition in [1] by representing an automaton as a list of starting states, a list of transitions, and a list of final states, and by representing a transducer as an automaton whose alphabet consists of pairs of symbols:

```haskell
data FA s i = FA{faStarts :: [s], faTrans :: [(s, i, s)], faFinals :: [s]}
deriving (Show, Read, Eq)
type Transducer s i o = FA s i
```

The following simple functions compute the list of symbols and states present in an automaton:

- `faAlphabet :: Eq i ⇒ FA s i → [i]`
- `faAlphabet fa = nub [a | (_, a, _) ← faTrans fa]`

The following functions, `faDet` and `faIntersect`, implement Lemmas 2 and 3 (of [1]). They return automata whose states are lists and pairs of the argument automata’s states, thus intrinsically encoding the Lemmas’ natural injections. Their implementations rely on the function `faTransFromDelta`, which constructs an explicit list of transitions from an algorithmically specified transition function. (Note that far more efficient implementations of `faDet` are possible.)

```haskell
faDet :: (Ord s, Eq i) ⇒ FA s i → FA [s] i
faDet fa = FA starts’ trans’ finals’
  where starts’ = [sort $ faStarts fa]
    trans’ = faTransFromDelta starts’ delta $ faAlphabet fa
    delta detState a | detState’ ≡ [] = Nothing
                      otherwise = Just detState’
    where detState’ = sort $ nub [s” | s ← detState, (s, a’, s”) ← faTrans fa, s’’ ≡ s, a’ ≡ a] finals’ = filter containsFinal (starts’ ‘union’ transStates trans’)
    containsFinal detState = or [s ∈ faFinals fa | s ← detState]

faIntersect :: (Eq s, Eq s’, Eq i) ⇒ FA s i → FA s’ i → FA (s, s’) i
faIntersect fa fa’ =
  FA starts” trans” finals”
  where starts” = [(s, s’) | s ← faStarts fa, s’ ← faStarts fa’]
    alphabet = faAlphabet fa ‘union’ faAlphabet fa’
```

LEMMAS 2 AND 3

Note that far more efficient implementations of `faDet` are possible.
The function \( \text{faDisjointUnion} \) forms the disjoint union of automata. It returns an automaton whose states are pairs \((i, s)\) where \(s\) is a state of the \(i\)th argument automaton \(D_i\).

\[
\text{faDisjointUnion} :: \text{[FA s i]} \to \text{FA (Int, s) i}
\]

\[
\text{faDisjointUnion} \text{fas} = \text{concatFas} \text{zipWith renameStates} [1 \ldots] \text{fas}
\]

\[
\text{concatFas } [] = \text{FA } [] \ldots \text{FA }
\]

\[
\text{concatFas} \text{(FA starts trans finals : fas)} = \text{FA (starts + starts') (trans + trans') (finals + finals')}
\]

\[
\text{where } \text{FA starts' trans' finals'} = \text{concatFas} \text{fas}
\]

We use these representations frequently in the following implementation of Algorithm 2.

**Algorithm 2**

\[
\text{transducerFilterFromDomains :: (Eq s, Ord s, Eq i) \to \text{[FA s i]} \to \text{Transducer [(Int, s)] i Int}}
\]

\[
\text{transducerFilterFromDomains faDs =}
\]

\[
\text{FA (faStarts faA) (baseTTrans + newTTrans) (faFinals faA)}
\]

\[
\text{where faA = faDet \$ \text{faDisjointUnion faDs} -- } A := \text{Det}(D_1 \cup \cdots \cup D_n)
\]

\[
\text{baseTTrans = \{ (s, (a, f s'), s') | (s, a, s') \to \text{faTrans faA} \}}
\]

\[
\text{where } f ss | \text{length is } 1 = \text{head is} -- \text{transition ending in } D_i
\]

\[
\text{where is = \text{nub } map \text{fst } fss}
\]

\[
\text{forbiddenPairs = \{ (s, a) | s \to \text{faStates faA, a \to \text{faAlphabet faA}} \}}
\]

\[
\text{newTTrans = map \text{newTransition forbiddenPairs}}
\]

\[
\text{newTransition (s, a) = (s, (a, o), s')}
\]

\[
\text{where faAsa = (FA (f : faStates faA) ((s, a, f) : \text{faTrans faA}) [f]) -- the automaton } A^{s,a}
\]

\[
f = [(1 + \text{length faDs}, \text{head } s \text{\starts } \text{head faDs})] -- \text{the fresh state } f \text{ used to build } A_{s,a}
\]

\[
\text{faDetAsaCapA} = \text{faDet faAsa} \text{"faIntersect" faA} -- \text{the automaton } \text{Det}(A_{s,a} \cap A)
\]

\[
\text{faZ = \text{faDet } zero \text{faDetAsaCapA} -- the automaton } \text{Det}(Z[\text{Det}(A_{s,a} \cap A)])
\]

\[
\text{reachableStateSeq = \text{take } (\text{length } \text{faStates faZ}) -- \text{the states } s_1, \ldots, s_{m+m'}
\]

\[
\text{(iterate nextState } \text{head } s \text{\starts faZ)}
\]

\[
\text{where nextState s = head } [s'' | (s', s'') \to \text{faTrans faZ, s' }\equiv s]
\]

\[
\text{sStarLs = map } (\text{map snd o} -- \text{the sets } \{S_{d,e}\}_{e=1}^{m+m'}
\]

\[
\text{intersect } \text{\text{faFinals faDetAsaCapA}} \text{reachableStateSeq}
\]

\[
\text{sdStars = [\text{nub } -- \text{the sets } \{S_{d,s}\}_{s=1}^n}
\]

\[
\text{[s | s \to \text{map snd } s \text{\starts faDetAsaCapA, d }\equiv \text{length } (\text{nub } (\text{map fst } s))]
\]

\[
\text{| d \equiv [1 \ldots \text{length faDs}]}
\]

\[
\text{sdls = concatMap (\text{ldsStar } \to \text{map } \text{intersect } \text{\text{sdStar} sStarLs}) \text{sdStars -- the sets } \{S_{d,e}\}
\]

\[
\text{s' = head } \text{\filter } (\text{\text{sdStar } \to \text{length } \text{sdls }\equiv 1}) \text{sdls -- the state } s' \text{ to which to synchronize}
\]

\[
o = -1 -- \text{domain break}
\]

\[
\text{zero fa = FA (faStarts fa) [(s, 0, s') | (s, s', s') \to \text{faTrans fa} (faFinals fa)}
\]
THE DISJOIN ALGORITHM

To implement Algorithm 4, we need considerably more automata-theoretic machinery. First of all, \texttt{faDistjoin} tests whether the languages (of positive-length strings) recognized by two given automata are disjoint:

\[
\text{faDistjoin} : (\text{Ord } s, \text{Ord } s', \text{Eq } i) \Rightarrow \text{FA } s i \rightarrow \text{FA } s' i \rightarrow \text{Bool}
\]

\[
\text{faDistjoin } fa \ fa' = \text{faNull } \triangleright \text{fa 'faIntersect' fa'}
\]

Here \texttt{faNull} tests whether an automaton accepts any (positive-length) strings:

\[
\text{faNull} : (\text{Ord } s, \text{Eq } i) \Rightarrow \text{FA } s i \rightarrow \text{Bool}
\]

\[
\text{faNull } fa \mid \text{null } \triangleright \text{faFinals } faD = \text{True}
\]

\[
\mid \text{\lnot } \triangleright \text{null } [s \mid (s, \_ s') \leftarrow \text{faTrans } faD, s' \in \text{faStarts } faD] = \text{False}
\]

\[
\mid \text{otherwise } = \text{True}
\]

where \texttt{faD} = \texttt{faDet} \texttt{fa}

We will also need to compute the “difference” of automata:

\[
\text{faDifference} : (\text{Eq } s, \text{Eq } i, \text{Ord } s') \Rightarrow \text{FA } s i \rightarrow \text{FA } s' i \rightarrow \text{FA } (s, [s']) i
\]

\[
\text{faDifference } fa \ fa' = \text{fa 'faIntersect' faComplement } \text{alphabet } fa'
\]

where \texttt{alphabet} = \texttt{faAlphabet} \texttt{fa} ‘union’ \texttt{faAlphabet} \texttt{fa}'

Here the function \texttt{faComplement} gives an automaton that accepts precisely those strings that its argument does not:

\[
\text{faComplement} : (\text{Ord } s, \text{Eq } i) \Rightarrow [i] \rightarrow \text{FA } s i \rightarrow [s] i
\]

\[
\text{faComplement } \text{alphabet } fa = \text{FA starts trans finals}
\]

where \texttt{faD} = \texttt{faDet} \texttt{fa}

\[
\text{starts } = \text{faStarts } faD
\]

\[
\text{trans } = (\text{faTrans } faD)
\]

\[
\triangleright [[s, a, []] \mid (s, a) \leftarrow \text{forbiddenPairs } \text{alphabet } faD, \text{\lnot } \triangleright \text{null } s]
\]

\[
\triangleright [[[], a, []] \mid a \leftarrow \text{alphabet}]
\]

\[
\text{finals } = ([]) : \text{faFinals } faD \andel \text{faFinals } faD
\]

\[
\text{forbiddenPairs} : (\text{Eq } s, \text{Eq } i) \Rightarrow [i] \rightarrow \text{FA } s i \rightarrow [(s, i)]
\]

\[
\text{forbiddenPairs } \text{alphabet } fa =
\]

\[
[(s, a) \mid s \leftarrow \text{faStates } fa, a \leftarrow \text{alphabet}] \andel [(s, a) \mid (s, a, \_) \leftarrow \text{faTrans } fa]
\]

We can now implement the \texttt{Disjoin} (−) algorithm:

\[
\text{faDisjoin} : (\text{Ord } s, \text{Eq } i) \Rightarrow [\text{FA } s i] \rightarrow [\text{FA Int } i]
\]

\[
\text{faDisjoin } [] = []
\]

\[
\text{faDisjoin } (fa : fas) = \text{filter } (\text{\lnot } \triangleright \text{faNull} (\text{faMinusRest } \text{disjoinedRestMinusFa})
\]

where \texttt{faMinusRest} = \texttt{faIS} \texttt{fa 'faDifference' (faIS } \texttt{faDisjointUnion fas})

\[
\text{disjoinedRestMinusFa } = \text{concatMap } f (\text{faDisjoin fas})
\]

\[
f fa' \mid \text{fa 'faDisjoint' fa'} = \text{[fa']}
\]

\[
\mid \text{otherwise } = \text{[faIS } \text{fa 'faIntersect' fa',
\]

\[
\text{faIS } \text{fa 'faDifference' fa]}
\]

Here \texttt{faIS} “integerizes” the states of an automaton; that is, it produces an equivalent automaton whose states are integers. This simplifies type signatures whenever the output of functions such as \texttt{faDet} and \texttt{faIntersect} are fed into one another. In fact, \texttt{faIS} will be required in order for several algorithms below to have type signatures at all, as they apply functions such as \texttt{faDet} a nonconstant number of times.

\[
\text{faIntegerizeStates}, \text{faIS} : \text{Eq } s \Rightarrow \text{FA } s i \rightarrow \text{FA Int } i
\]

\[
\text{faIS } = \text{faIntegerizeStates}
\]

\[
\text{faIntegerizeStates } fa =
\]

\[
\text{FA starts trans finals}
\]

where \texttt{table} = \texttt{zip} (\texttt{faStates } fa) [1..]

\[
\text{starts } = \text{catMaybe } \text{[lookup } s \text{ table } \mid s \leftarrow \text{faStarts } fa]
\]

\[
\text{trans } = ([i, a, j] \mid (s, a, s') \leftarrow \text{faTrans } fa,
\]

\[
\text{let } \text{Just } i = \text{lookup } s \text{ table,}
\]

\[
\text{let } \text{Just } j = \text{lookup } s' \text{ table}
\]

\[
\text{finals } = \text{catMaybe } \text{[lookup } s \text{ table } \mid s \leftarrow \text{faFinals } fa]
\]

EPSILON MOVES AND AUTOMATON CONCATENATION

We use the machinery of “ɛ-moves” (see [3]) to concatenate automata:
data Epsilon a = Epsilon | Letter a

deriving Show

faRemoveEpsilons :: Eq s ⇒ FA s (Epsilon i) → FA s i
faRemoveEpsilons fa = FA (faStarts trans finals)
where starts = epsilonClosure fa $ faStarts fa
finals = reverseEpsilonClosure fa $ faFinals fa
trans = concatMap inducedTrans $ faTrans fa
inducedTrans (s, Letter a, s') =
[(s'', a, s''') | s'' ← reverseEpsilonClosure fa [s],
 s''' ← reverseEpsilonClosure fa [s']]

epsilonClosure, reverseEpsilonClosure :: Eq s ⇒ FA s (Epsilon i) → [s] → [s]
epsilonClosure fa states = fixedPointBy (≡) $ iterate epsilonAway states
where epsilonAway ss = nub $ ss ⊕ [(s | (s, Epsilon, s′) ← FATrans fa, s ∈ ss]
reverseEpsilonClosure = epsilonClosure ⊙ faReverse

faReverse :: FA s i → FA s i
faReverse fa = FA (faFinals fa) trans (faStarts fa)
where trans = [(s′, a, s) | (s, a, s′) ← FATrans fa]

fixedPointBy :: (a → a → Bool) → [a] → a → a
-- Assumes that [s] has a fixed point
fixedPointBy eqFunc (a1 : a2 : as) | a1 `eqFunc` a2 = a1
| otherwise = fixedPointBy eqFunc (a2 : as)

faConcat :: (Ord s, Eq i) ⇒ FA s i → FA s i → FA [(Int, s)] i
faConcat fa fa' = faDet $ faRemoveEpsilons (FA (faStarts trans finals)
where starts = [(1, s) | s ← faStarts fa]
finals = [(2, s) | s ← faFinals fa']
trans = [(1, s), Letter i, (1, s')] | (s, i, s') ← FATrans fa
where inducedTrans (s, Letter i, (2, s')) | s ← faFinals fa, s' ← FAStarts fa'

ALGORITHMS 3 AND 4

The first major step of Algorithm 4 is to construct the initial function Γ (see [1]). Given a list of domains \( \{D_i\} \), we produce for each \( D_i \) the association list (or “graph”) of Γ, \( \{(s, Γ(s))\}_{s ∈ S(D_i)} \). We do this in a slightly more efficient way than in the text. Rather than computing \( \text{Det}(A^+, s) \cap A_s \) repeatedly for each \( s \in S(A) \)—most of them will be empty anyway—, we compute \( \text{Det}(A^+, s) \cap A \) once, and then examine its states to determine which \( s \)’s actually matter.

initialGamma :: (Ord s, Eq i) ⇒ [FA s i] → [[[s, [FA Int i]]]]
initialGamma faDs =
[([(s, gamma [(i, s)]) | s ← faStates (faDs !! (i - 1))] | i ← [1..length faDs]]
where faA = faDet $ faDisjointUnion faDs
alphabet = faAlphabet faA
gamma s | null forbiddenAs = [faSigmaStar alphabet]
| otherwise = map (faIS o faMin)$
\begin{align*}
faDisjoin \ [faSigmaStar alphabet \ ‘forbiddenAs’ \ faBisas’ \ | \ a ← forbiddenAs, faBisas’ ← gamma’ s a]
\end{align*}
where forbiddenAs = alphabet \ \{ a | (s’, a, ‾) ← FATrans faA, s’ ∈ S(A) \}
gamma’ s a = [faIS s FA (faStarts faDetAsaCapA) (faTrans faDetAsaCapA) \[s'' | (s'', _ , s''') ← FATrans faDetAsaCapA, s''' ∈ s’\] \[s | s' ← resyncStates]}

where faAsa = (FA \( f : faStates faA \) \( (s, a, f) : FATrans faA \) \( f \))
f = [(1 + length faDs, head \$ faStarts $ head faDs)] -- the fresh state used to build \( A^+ \)
faDetAsaCapA = (faDet faAsa) \( ‘\text{faIntersect’ \ faA \) -- the automaton \( \text{Det}(A^+, s) \cap A \)
resyncStates = nub \[s’ | (s’, s’) ← FAFinals faDetAsaCapA]\ -- the states \( s’ \) that actually matter

faPullBackFinalsBy :: (Eq s, Eq i) ⇒ i → FA s i → FA s i
faPullBackFinalsBy a fa = FA (faStarts fa) (faTrans fa) finals
where \(\text{finals} = [s \mid (s, a', s') \rightarrow \text{faTrans} \ fa, a' \equiv a, s' \in \text{faFinals} \ fa]\)

\[
\text{faMin} :: [i] \rightarrow \text{FA Int} \ i
\]

\[
\text{faMin} \ \text{alphabet} = \text{FA} \ \left[ (1, a, 1) \mid a \leftrightarrow \text{alphabet} \right] \ [1]
\]

Here \(\text{faMin}\) "minimizes" an automaton; that is, it produces an equivalent automaton having as few states as possible (see [3]). Note that we do this here for the sake of tidiness alone. A concise, but extremely inefficient, implementation of \(\text{faMin}\) is:

\[
\text{faMin} :: (\text{Ord} \ s, \text{Eq} \ i) \Rightarrow \text{FA} \ s \ i \rightarrow \text{FA} \ [(s)] \ i
\]

\[
\text{faMin} = \text{faDet} \circ \text{faReverse} \circ \text{faDet} \circ \text{faReverse}
\]

The second major step of Algorithm 4 is to refine \(\Gamma\) to make it compatible with the transition structures of the \(D_i\),—a process called Algorithm 3 in [1]. The function \(\text{optimalGamma}\) ultimately produces the same output as Algorithm 3 of the paper, but it requires fewer refinement steps because it refines \(\text{transition by transition}\), rather than state by state, and the refinements thus made are immediately passed along to the following transitions within a single refinement step. Not only does this reduce the total number of refinement steps, but by refining a single transition at a time we can avoid excessive \(\text{Disjoin}(\-)\) operations. We also use Haskell arrays to further boost performance.

\[
\text{refineGammaForSingleD} :: (\text{Ix} \ s, \text{Eq} \ i) \Rightarrow \text{FA} \ s \ i \rightarrow [(s, \text{FA Int} \ i)] \rightarrow [(s, \text{FA Int} \ i)]
\]

\[
\text{refineGammaForSingleD} \ \text{faDs} =
\]

\[
\text{where} \ \text{initialGammaArray} = \text{array} \ \text{arrayBounds} \ \text{associations}
\]

\[
\text{arrayBounds} = (\text{minimum} \ \text{faStates} \ \text{faD}, \text{maximum} \ \text{faStates} \ \text{faD})
\]

\[
\text{refineArrayWithTransition} \ \text{array} \ (s, a, s') = \text{accum} \ \langle \text{curr \ snd} \rangle \ \text{array} \ [(s, \text{gamma}')]
\]

\[
\text{where} \ \text{gamma}' = \text{map} \ (\text{faIS} \circ \text{faMin}) \ \text{filter} \ \langle \text{\neg} \circ \text{faNull} \rangle
\]

\[
\text{faPullBackFinalsBy} \ a \ \text{$\text{iterate}$} \ (\text{zipWith} \ \text{optimalDomainFromGamma} \ \text{faDs}) \ \text{initialGamma} \ \text{faDs}
\]

\[
\text{where} \ \text{cardinality} \ aa = \text{sum} \ \langle \text{\text{length} \ \text{gammaOfs}} | \ \text{associations} \ \leftarrow \ aa, \langle \_{\_} \rangle, \text{gammaOfs} \rangle \ \text{associations}
\]

\[
\text{cardinalityEq} \ aa \ aa' = \text{cardinality} \ aa \equiv \text{cardinality} \ aa'
\]

Now that \(\Gamma\) is compatible with the transition structures of the \(D_i\), we can construct the automata \(\{D'_i\} = \text{Optimize}(\{D_i\})\):

\[
\text{optimizeDomains} :: (\text{Ix} \ s, \text{Ord} \ s, \text{Eq} \ i) \Rightarrow \text{FA} \ s \ i \rightarrow [\text{FA Int} \ s]
\]

\[
\text{optimizeDomains} \ \text{faDs} =
\]

\[
\text{where} \ \text{optimalDomainFromGamma} \ \text{faDs} = \text{faIS} \ \text{faStarts} \ \text{trans} \ \text{finals}
\]

\[
\text{where} \ \text{starts} = [s, \text{faE}] \ s \leftrightarrow \text{faStarts} \ \text{faD}, \text{faE} \leftrightarrow \text{gamma} \ s
\]

\[
\text{trans} = \left( [[(s, \text{faE}), a, (s', \text{faE}')), (s, a, s') \rightarrow \text{faTrans} \ \text{faD}, \text{faE} \leftrightarrow \text{gamma} \ s, \text{faE'} \leftrightarrow \text{gamma} \ s']ight)
\]

\[
\text{finals} = \left( [s, \text{faE}] \mid (s, a) \rightarrow \text{faFinals} \ \text{faD}, \text{faE} \leftrightarrow \text{gamma} \ s
\]

\[
\gamma \ s \ = \text{fromJust} \ \text{lookup} \ s \ \text{associations}
\]

\[
\text{faSubset} :: (\text{Ord} \ s, \text{Eq} \ i) \Rightarrow \text{FA} \ s \ i \rightarrow \text{FA} \ s \ i \rightarrow \text{Bool}
\]

\[
\text{faSubset} \ \text{fa} \ fa' = \text{faNull} \ \text{fa} \ \text{faDifference} \ \text{fa}'
\]

This concludes the implementation of Algorithm 4.

EXAMPLE

Consider the automata in FIG. 7 of [1]:

\[
\text{fig7Domains} = [\text{FA} \ [1, 2] \ \{(1, 0, 2), (1, 1, 2), (2, 0, 1)\} \ [1, 2],
\text{FA} \ [3 \ldots 6] \ \{(3, 1, 4), (4, 1, 5), (5, 0, 6), (6, 0, 3), (6, 1, 3)\} \ [3 \ldots 6]]
\]

The above defined function \(\text{optimizeDomains}\) when applied to \(\text{fig7Domains}\) gives:

\[
[\text{FA} \ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13],
[1, 0, 9), (2, 0, 10), (3, 0, 11), (4, 0, 12), (5, 0, 13), (1, 1, 6), (2, 1, 7), (3, 1, 8), (4, 1, 7),
(5, 1, 7), (6, 0, 1), (7, 0, 1), (8, 0, 1), (9, 0, 2), (10, 0, 2), (11, 0, 2), (12, 0, 2), (13, 0, 2)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13],
\]
\[ FA \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\}, \\
[(1, 1, 6), (2, 1, 6), (3, 1, 6), (4, 1, 7), (5, 1, 7), (6, 1, 12), (7, 1, 12), (8, 1, 12), \\
(9, 1, 12), (10, 1, 12), (11, 1, 13), (12, 0, 16), (13, 0, 16), (14, 0, 16), (15, 0, 17), (16, 0, 1), (17, 0, 2), \\
(18, 0, 3), (16, 1, 4), (17, 1, 4), (18, 1, 5)]\]

(Compare FIG. 8 of \[1\]).

REFERENCES

