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 polymorphicly typed, lazy, purely functional programming language design [2]. Haskell compilers and interpreters are freely available for almost any computer (see www.haskell.org).

The raw source code of what follows is available at http://cse.ucdavis.edu/~cmg/compmech/pubs/apd.htm.

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:

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

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

$faAlphabet :: Eq\ i \Rightarrow FA\ s\ i \rightarrow [i]$
$faAlphabet\ fa = nub\ [a \mid (\_, a, \_) \leftarrow faTrans\ fa]$

$faStates :: Eq\ s \Rightarrow FA\ s\ i \rightarrow [s]$
$faStates\ fa = faStarts\ fa\ \text{`}union\text{`}\ (transStates\ \$\ faTrans\ fa)\ \text{`}union\text{`}\ faFinals\ fa$

$transStates :: Eq\ s \Rightarrow [(s,i,s)] \rightarrow [s]$
$transStates\ trans = nub\ \$\ foldl\ (\lambda ss\ (s, \_, s') \rightarrow s : s' : ss)\ []\ trans$

Lemmas 2 and 3

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.)

$faDet :: (Ord\ s, Eq\ i) \Rightarrow FA\ s\ i \rightarrow FA\ [s]\ i$
$faDet\ fa = FA\ starts'\ trans'\ finals'$
  where $starts' = [sort\ \$\ faStarts\ fa]$
    $trans' = faTransFromDelta\ starts'\ delta\ \$\ faAlphabet\ fa$
    $delta\ detState\ a \mid detState' \equiv [] = Nothing$
      $\mid otherwise = Just\ detState'$
     where $detState' = sort\ \$\ nub\ [s'' \mid s \leftarrow detState,$
        $(s', a', s'') \leftarrow faTrans\ fa, s' \equiv s, a' \equiv a]$
    $finals' = filter\ containsFinal\ (starts'\ \text{`}union\text{`}\ transStates\ trans')$
    $containsFinal\ detState = or\ [s \in faFinals\ fa \mid s \leftarrow detState]$

$faIntersect :: (Eq\ s, Eq\ s', Eq\ i) \Rightarrow$ -- assumes fa and fa' are semi-deterministic
   $FA\ s\ i \rightarrow FA\ s'\ i \rightarrow FA\ (s,s')\ i$
$faIntersect\ fa\ fa' =$
  $FA\ starts''\ trans''\ finals''$
  where $starts'' = [(s,s') \mid s \leftarrow faStarts\ fa, s' \leftarrow faStarts\ fa']$
    $alphabet = faAlphabet\ fa\ \text{`}union\text{`}\ faAlphabet\ fa'$

```
        trans'' = faTransFromDelta starts'' delta alphabet
        delta (s, s') a | null ss ∨ null ss' = Nothing
                        | otherwise = Just (head ss, head ss')
          where ss  = [s''' | (s'', a', s''') ← faTrans fa, s'' ≡ s, a ≡ a']
                ss' = [s''' | (s'', a', s''') ← faTrans fa', s'' ≡ s', a ≡ a']
        finals'' = filter bothFinal $ union starts'' $ transStates trans''
        bothFinal (s, s') = s ∈ faFinals fa ∧ s' ∈ faFinals fa'

faTransFromDelta :: (Eq s, Eq i) ⇒ [s] → (s → i → Maybe s) → [i] → [(s, i, s)]
faTransFromDelta starts delta alphabet = trans
    where f states | null (states' \\ states) = states
                   | otherwise = f $ nub $ states ++ states'
                  where states' = nub $ catMaybes [delta s i | s ← states, i ← alphabet]
          states = f starts
          trans = [(s, i, fromJust s') | s ← states, i ← alphabet, let s' = delta s i, isJust s']
```

The function *faDisjointUnion* forms the disjoint union of automata. It returns an automaton whose states are pairs (i, s) where s is a state of the ith argument automaton $\mathcal{D}_i$.

```
faDisjointUnion :: [FA s i] → FA (Int, s) i
faDisjointUnion fas = concatFas $ zipWith renameStates [1 ..] fas
  where renameStates n fa = FA starts' trans' finals'
          where starts' = [(n, s) | s ← faStarts fa]
                trans' = [((n, s), a, (n, s')) | (s, a, s') ← faTrans fa]
                finals' = [(n, s) | s ← faFinals fa]
        concatFas [] = FA [] [] []
        concatFas (FA starts trans finals : fas) =
          FA (starts ++ starts') (trans ++ trans') (finals ++ finals')
          where FA starts' trans' finals' = concatFas fas
```

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

ALGORITHM 2

```
transducerFilterFromDomains :: (Eq s, Ord s, Eq i) ⇒ [FA s i] → Transducer [(Int, s)] i Int
transducerFilterFromDomains faDs =
  FA (faStarts faA) (baseTTrans ++ newTTrans) (faFinals faA)
  where faA = faDet $ faDisjointUnion faDs      -- $\mathcal{A} := \mathrm{Det}(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$
        baseTTrans = [(s, (a, f s'), s') | (s, a, s') ← faTrans faA]
          where f ss | length is ≡ 1 = head is      -- transition ending in $\mathcal{D}_i$
                     | otherwise = 0      -- synchronization, $\lambda$
                    where is = nub $ map fst ss
        forbiddenPairs = [(s, a) | s ← faStates faA, a ← faAlphabet faA] \\ [(s, a) | (s, a, _) ← faTrans faA]
        newTTrans = map newTransition forbiddenPairs
        newTransition (s, a) = (s, (a, o), s')
          where faAsa = (FA (f : faStates faA) ((s, a, f) : faTrans faA) [f])      -- the automaton $\mathcal{A}^{s,a}$
                f = [(1 + length faDs, head $ faStarts $ head faDs)]      -- the fresh state $f$ used to build $\mathcal{A}^{s,a}$
                faDetAsaCapA = (faDet faAsa) `faIntersect` faA      -- the automaton $\mathrm{Det}(\mathcal{A}^{s,a} \cap \mathcal{A})$
                faZ = faDet $ zero faDetAsaCapA      -- the automaton $\mathrm{Det}(\mathcal{Z}[\mathrm{Det}(\mathcal{A}^{s,a} \cap \mathcal{A})])$
                reachableStateSeq = take (length $ faStates faZ)      -- the states $s_1, \ldots, s_{m+m'}$
                                         (iterate nextState $ head $ faStarts faZ)
                  where nextState s = head [s'' | (s', _, s'') ← faTrans faZ, s' ≡ s]
                sStarLs = map ((map snd) ∘      -- the sets $\{S_{*,\ell}\}_{\ell=1}^{m+m'}$
                               (intersect $ faFinals faDetAsaCapA)) reachableStateSeq
                sdStars = [nub      -- the sets $\{S_{d,*}\}_{i=1}^{n}$
                             [s | s ← map snd $ faFinals faDetAsaCapA, d ≡ length (nub (map fst s))]
                          | d ← [1 .. length faDs]]
                sdls = concatMap (λsdStar → map (intersect sdStar) sStarLs) sdStars      -- the sets $\{S_{d,\ell}\}$
                [s'] = head $ filter (λsdl → length sdl ≡ 1) sdls      -- the state $s'$ to which to synchronize
                o = −1      -- domain break
        zero fa = FA (faStarts fa) [(s, 0, s') | (s, _, s') ← faTrans fa] (faFinals fa)
```

The Disjoin Algorithm

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

$$\begin{aligned}&\mathit{faDisjoint} :: (\mathit{Ord}\ s, \mathit{Ord}\ s', \mathit{Eq}\ i) \Rightarrow \mathit{FA}\ s\ i \to \mathit{FA}\ s'\ i \to \mathit{Bool}\\&\mathit{faDisjoint}\ \mathit{fa}\ \mathit{fa}' = \mathit{faNull}\ \$\ \mathit{fa}\ \text{`}\mathit{faIntersect}\text{`}\ \mathit{fa}'\end{aligned}$$

Here *faNull* tests whether an automaton accepts any (positive-length) strings:

$$\begin{aligned}&\mathit{faNull} :: (\mathit{Ord}\ s, \mathit{Eq}\ i) \Rightarrow \mathit{FA}\ s\ i \to \mathit{Bool}\\&\mathit{faNull}\ \mathit{fa} \mid \mathit{null}\ \$\ \mathit{faFinals}\ \mathit{faD} = \mathit{True}\\&\qquad\qquad \mid \neg\ \$\ \mathit{null}\ (\mathit{faFinals}\ \mathit{faD} \setminus\setminus \mathit{faStarts}\ \mathit{faD}) = \mathit{False}\\&\qquad\qquad \mid \neg\ \$\ \mathit{null}\ [s \mid (s, \_, s') \leftarrow \mathit{faTrans}\ \mathit{faD}, s' \in \mathit{faStarts}\ \mathit{faD}] = \mathit{False}\\&\qquad\qquad \mid \mathit{otherwise} = \mathit{True}\\&\quad \mathbf{where}\ \mathit{faD} = \mathit{faDet}\ \mathit{fa}\end{aligned}$$

We will also need to compute the "difference" of automata:

$$\begin{aligned}&\mathit{faDifference} :: (\mathit{Eq}\ s, \mathit{Eq}\ i, \mathit{Ord}\ s') \Rightarrow \mathit{FA}\ s\ i \to \mathit{FA}\ s'\ i \to \mathit{FA}\ (s, [s'])\ i\\&\mathit{faDifference}\ \mathit{fa}\ \mathit{fa}' = \mathit{fa}\ \text{`}\mathit{faIntersect}\text{`}\ \mathit{faComplement}\ \mathit{alphabet}\ \mathit{fa}'\\&\quad \mathbf{where}\ \mathit{alphabet} = \mathit{faAlphabet}\ \mathit{fa}\ \text{`}\mathit{union}\text{`}\ \mathit{faAlphabet}\ \mathit{fa}'\end{aligned}$$

Here the function *faComplement* gives an automaton that accepts precisely those strings that its argument does not:

$$\begin{aligned}&\mathit{faComplement} :: (\mathit{Ord}\ s, \mathit{Eq}\ i) \Rightarrow [i] \to \mathit{FA}\ s\ i \to \mathit{FA}\ [s]\ i\\&\mathit{faComplement}\ \mathit{alphabet}\ \mathit{fa} = \mathit{FA}\ \mathit{starts}\ \mathit{trans}\ \mathit{finals}\\&\quad \mathbf{where}\ \mathit{faD} = \mathit{faDet}\ \mathit{fa}\\&\qquad\quad \mathit{starts} = \mathit{faStarts}\ \mathit{faD}\\&\qquad\quad \mathit{trans} = (\mathit{faTrans}\ \mathit{faD})\\&\qquad\qquad +\!\!+\ [(s, a, [\,]) \mid (s, a) \leftarrow \mathit{forbiddenPairs}\ \mathit{alphabet}\ \mathit{faD}, \neg\ \$\ \mathit{null}\ s]\\&\qquad\qquad +\!\!+\ [([\,], a, [\,]) \mid a \leftarrow \mathit{alphabet}]\\&\qquad\quad \mathit{finals} = ([\,] : \mathit{faStates}\ \mathit{faD}) \setminus\setminus \mathit{faFinals}\ \mathit{faD}\end{aligned}$$

$$\begin{aligned}&\mathit{forbiddenPairs} :: (\mathit{Eq}\ s, \mathit{Eq}\ i) \Rightarrow [i] \to \mathit{FA}\ s\ i \to [(s, i)]\\&\mathit{forbiddenPairs}\ \mathit{alphabet}\ \mathit{fa} =\\&\quad [(s, a) \mid s \leftarrow \mathit{faStates}\ \mathit{fa}, a \leftarrow \mathit{alphabet}] \setminus\setminus [(s, a) \mid (s, a, \_) \leftarrow \mathit{faTrans}\ \mathit{fa}]\end{aligned}$$

We can now implement the $\mathrm{Disjoin}(-)$ algorithm:

$$\begin{aligned}&\mathit{faDisjoin} :: (\mathit{Ord}\ s, \mathit{Eq}\ i) \Rightarrow [\mathit{FA}\ s\ i] \to [\mathit{FA}\ \mathit{Int}\ i]\\&\mathit{faDisjoin}\ [\,] = [\,]\\&\mathit{faDisjoin}\ (\mathit{fa} : \mathit{fas}) = \mathit{filter}\ (\neg \circ \mathit{faNull})\ (\mathit{faMinusRest} : \mathit{disjoinedRestMinusFa})\\&\quad \mathbf{where}\ \mathit{faMinusRest} = \mathit{faIS}\ \$\ \mathit{fa}\ \text{`}\mathit{faDifference}\text{`}\ (\mathit{faIS}\ \$\ \mathit{faDisjointUnion}\ \mathit{fas})\\&\qquad\quad \mathit{disjoinedRestMinusFa} = \mathit{concatMap}\ f\ (\mathit{faDisjoin}\ \mathit{fas})\\&\qquad\quad f\ \mathit{fa}' \mid \mathit{fa}\ \text{`}\mathit{faDisjoint}\text{`}\ \mathit{fa}' = [\mathit{fa}']\\&\qquad\qquad \mid \mathit{otherwise} = [\mathit{faIS}\ \$\ \mathit{fa}\ \text{`}\mathit{faIntersect}\text{`}\ \mathit{fa}',\\&\qquad\qquad\qquad\qquad\quad \mathit{faIS}\ \$\ \mathit{fa}'\ \text{`}\mathit{faDifference}\text{`}\ \mathit{fa}]\end{aligned}$$

Here *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 *faDet* and *faIntersect* are fed into one another. In fact, *faIS* will be required in order for several algorithms below to have type signatures at all, as they apply functions such as *faDet* a nonconstant number of times.

$$\begin{aligned}&\mathit{faIntegerizeStates}, \mathit{faIS} :: \mathit{Eq}\ s \Rightarrow \mathit{FA}\ s\ i \to \mathit{FA}\ \mathit{Int}\ i\\&\mathit{faIS} = \mathit{faIntegerizeStates}\\&\mathit{faIntegerizeStates}\ \mathit{fa} =\\&\quad \mathit{FA}\ \mathit{starts}\ \mathit{trans}\ \mathit{finals}\\&\quad \mathbf{where}\ \mathit{table} = \mathit{zip}\ (\mathit{faStates}\ \mathit{fa})\ [1\,..]\\&\qquad\quad \mathit{starts} = \mathit{catMaybes}\ [\mathit{lookup}\ s\ \mathit{table} \mid s \leftarrow \mathit{faStarts}\ \mathit{fa}]\\&\qquad\quad \mathit{trans} = [(i, a, j) \mid (s, a, s') \leftarrow \mathit{faTrans}\ \mathit{fa},\\&\qquad\qquad\qquad \mathbf{let}\ \mathit{Just}\ i = \mathit{lookup}\ s\ \mathit{table},\\&\qquad\qquad\qquad \mathbf{let}\ \mathit{Just}\ j = \mathit{lookup}\ s'\ \mathit{table}]\\&\qquad\quad \mathit{finals} = \mathit{catMaybes}\ [\mathit{lookup}\ s\ \mathit{table} \mid s \leftarrow \mathit{faFinals}\ \mathit{fa}]\end{aligned}$$

Epsilon Moves and Automaton Concatenation

We use the machinery of "ϵ-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 starts trans finals
    where starts = epsilonClosure fa $ faStarts fa
          finals = reverseEpsilonClosure fa $ faFinals fa
          trans = concatMap inducedTrans $ faTrans fa
          inducedTrans (_, Epsilon, _) = [ ]
          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       -- 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 starts 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]
               ++ [((1, s), Epsilon, (2, s′)) | s ← faFinals fa, s′ ← faStarts fa′]
               ++ [((2, s), Letter i, (2, s′)) | (s, i, s′) ← faTrans 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 $\{\mathcal{D}_i\}$, we produce for each $\mathcal{D}_i$ the association list (or "graph") of Γ, $\{(s, \Gamma(s))\}_{s \in S(\mathcal{D}_i)}$. We do this in a slightly more efficient way than in the text. Rather than computing $\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}_{s'}$ repeatedly for each $s' \in S(\mathcal{A})$—most of them will be empty anyway—, we compute $\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}$ once, and then examine its states to determine which 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 ∘ faMin)$
                      faDisjoin [faSigmaStar alphabet ‘faConcat‘ faBsas′
                                          | a ← forbiddenAs, faBsas′ ← gamma′ s a]
                      -- let Γ(s) := Disjoin({Σ* · B(s, a, s′) : (s, a, s″) ∉ T(A), s′ ∈ S(A)})
              where forbiddenAs = alphabet \\ [a | (s′, a, _) ← faTrans faA, s′ ≡ s]
          gamma′ s a = [faIS $ FA (faStarts faDetAsaCapA) (faTrans faDetAsaCapA)
                              [s″ | (s″, _, (_, s‴)) ← faTrans faDetAsaCapA, s‴ ≡ s′]
                              | s′ ← resyncStates]
              where faAsa = (FA (f : faStates faA) ((s, a, f) : faTrans faA) [f])       -- the automaton A^{s,a}
                    f = [(1 + length faDs, head $ faStarts $ head faDs)]       -- the fresh state used to build A^{s,a}
                    faDetAsaCapA = (faDet faAsa) ‘faIntersect‘ faA       -- the automaton Det(A^{s,a}) ∩ A
                    resyncStates = nub [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
```

$$\textbf{where } \mathit{finals} = [\, s \mid (s, a', s') \leftarrow \mathit{faTrans}\ \mathit{fa},\ a' \equiv a,\ s' \in \mathit{faFinals}\ \mathit{fa} \,]$$

$$\mathit{faSigmaStar} :: [\,i\,] \rightarrow \mathit{FA}\ \mathit{Int}\ i$$
$$\mathit{faSigmaStar}\ \mathit{alphabet} = \mathit{FA}\ [1]\ [(1, a, 1) \mid a \leftarrow \mathit{alphabet}]\ [1]$$

Here 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 faMin is:

$$\mathit{faMin} :: (\mathit{Ord}\ s, \mathit{Eq}\ i) \Rightarrow \mathit{FA}\ s\ i \rightarrow \mathit{FA}\ [[s]]\ i$$
$$\mathit{faMin} = \mathit{faDet} \circ \mathit{faReverse} \circ \mathit{faDet} \circ \mathit{faReverse}$$

The second major step of Algorithm 4 is to refine Γ to make it compatible with the transition structures of the $\mathcal{D}_i$—a process called Algorithm 3 in [1]. The function $\mathit{optimalGamma}$ ultimately produces the same output as Algorithm 3 of the paper, but it requires fewer refinement steps because it refines *transition by transion*, 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 expensive $\mathrm{Disjoin}(-)$ operations. We also use Haskell arrays to further boost performance.

$$\begin{array}{l}
\mathit{refineGammaForSingleD} :: (\mathit{Ix}\ s, \mathit{Eq}\ i) \Rightarrow \mathit{FA}\ s\ i \rightarrow [(s, [\mathit{FA}\ \mathit{Int}\ i])] \rightarrow [(s, [\mathit{FA}\ \mathit{Int}\ i])] \\
\mathit{refineGammaForSingleD}\ \mathit{faD}\ \mathit{associations} = \\
\quad \mathit{assocs}\ \$\ \mathit{foldl}\ \mathit{refineArrayWithTransition}\ \mathit{initialGammaArray}\ \$\ \mathit{faTrans}\ \mathit{faD} \\
\quad \textbf{where}\ \mathit{initialGammaArray} = \mathit{array}\ \mathit{arrayBounds}\ \mathit{associations} \\
\qquad\quad \mathit{arrayBounds} = (\mathit{minimum}\ \$\ \mathit{faStates}\ \mathit{faD}, \mathit{maximum}\ \$\ \mathit{faStates}\ \mathit{faD}) \\
\qquad\quad \mathit{refineArrayWithTransition}\ \mathit{array}\ (s, a, s') = \mathit{accum}\ (\mathit{curry}\ \mathit{snd})\ \mathit{array}\ [(s, \mathit{gamma}')] \\
\qquad\qquad \textbf{where}\ \mathit{gamma}' = \mathit{map}\ (\mathit{faIS} \circ \mathit{faMin})\ \$\ \mathit{filter}\ (\neg \circ \mathit{faNull}) \\
\qquad\qquad\qquad [\mathit{faPullBackFinalsBy}\ a\ \$\ \mathit{faIS} \$ \\
\qquad\qquad\qquad\ (\mathit{faE}\ \text{‘}\mathit{faConcat}\text{‘}\ \mathit{FA}\ [1]\ [(1, a, 2)]\ [2])\ \text{‘}\mathit{faIntersect}\text{‘}\ \mathit{faE}' \\
\qquad\qquad\qquad\quad \mid \mathit{faE} \leftarrow \mathit{array}\ !\ s, \mathit{faE}' \leftarrow \mathit{array}\ !\ s'\,]
\end{array}$$

$$\begin{array}{l}
\mathit{optimalGamma} :: (\mathit{Ix}\ s, \mathit{Eq}\ i) \Rightarrow [\mathit{FA}\ s\ i] \rightarrow [[(s, [\mathit{FA}\ \mathit{Int}\ i])]] \\
\mathit{optimalGamma}\ \mathit{faDs} = \\
\quad \mathit{fixedPointBy}\ \mathit{cardinalityEq}\ \$\ \mathit{iterate}\ (\mathit{zipWith}\ \mathit{refineGammaForSingleD}\ \mathit{faDs})\ \$\ \mathit{initialGamma}\ \mathit{faDs} \\
\quad \textbf{where}\ \mathit{cardinality}\ \mathit{aa} = \mathit{sum}\ [\mathit{length}\ \mathit{gammaOfs} \mid \mathit{associations} \leftarrow \mathit{aa}, (_, \mathit{gammaOfs}) \leftarrow \mathit{associations}] \\
\qquad\quad \mathit{cardinalityEq}\ \mathit{aa}\ \mathit{aa}' = \mathit{cardinality}\ \mathit{aa} \equiv \mathit{cardinality}\ \mathit{aa}'
\end{array}$$

Now that Γ is compatible with the transition structures of the $\mathcal{D}_i$, we can construct the automata $\{\mathcal{D}'_i\} = \mathrm{Optimize}(\{\mathcal{D}_i\})$:

$$\begin{array}{l}
\mathit{optimizeDomains} :: (\mathit{Ix}\ s, \mathit{Ord}\ s, \mathit{Eq}\ i) \Rightarrow [\mathit{FA}\ s\ i] \rightarrow [\mathit{FA}\ \mathit{Int}\ i] \\
\mathit{optimizeDomains}\ \mathit{faDs} = \\
\quad \mathit{zipWith}\ \mathit{optimalDomainFromGamma}\ \mathit{faDs}\ \$\ \mathit{optimalGamma}\ \mathit{faDs} \\
\quad \textbf{where}\ \mathit{optimalDomainFromGamma}\ \mathit{faD}\ \mathit{associations} = \mathit{faIS}\ \$\ \mathit{FA}\ \mathit{starts}\ \mathit{trans}\ \mathit{finals} \\
\qquad\quad \textbf{where}\ \mathit{starts} = [(s, \mathit{faE}) \mid s \leftarrow \mathit{faStarts}\ \mathit{faD}, \mathit{faE} \leftarrow \mathit{gamma}\ s] \\
\qquad\qquad \mathit{trans} = [((s, \mathit{faE}), a, (s', \mathit{faE}')) \mid (s, a, s') \leftarrow \mathit{faTrans}\ \mathit{faD}, \\
\qquad\qquad\qquad \mathit{faE} \leftarrow \mathit{gamma}\ s, \mathit{faE}' \leftarrow \mathit{gamma}\ s', \\
\qquad\qquad\qquad (\mathit{faIS}\ (\mathit{faE}\ \text{‘}\mathit{faConcat}\text{‘}\ (\mathit{FA}\ [1]\ [(1, a, 2)]\ [2])))\ \text{‘}\mathit{faSubset}\text{‘}\ \mathit{faE}'] \\
\qquad\qquad \mathit{finals} = [(s, \mathit{faE}) \mid s \leftarrow \mathit{faFinals}\ \mathit{faD}, \mathit{faE} \leftarrow \mathit{gamma}\ s] \\
\qquad\qquad \mathit{gamma}\ s = \mathit{fromJust}\ \$\ \mathit{lookup}\ s\ \mathit{associations}
\end{array}$$

$$\mathit{faSubset} :: (\mathit{Ord}\ s, \mathit{Eq}\ i) \Rightarrow \mathit{FA}\ s\ i \rightarrow \mathit{FA}\ s\ i \rightarrow \mathit{Bool}$$
$$\mathit{faSubset}\ \mathit{fa}\ \mathit{fa}' = \mathit{faNull}\ \$\ \mathit{fa}\ \text{‘}\mathit{faDifference}\text{‘}\ \mathit{fa}'$$

This concludes the implementation of Algorithm 4.

EXAMPLE

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

$$\begin{array}{l}
\mathit{fig7Domains} = [\mathit{FA}\ [1, 2]\ [(1, 0, 2), (1, 1, 2), (2, 0, 1)]\ [1, 2], \\
\qquad\qquad\qquad\quad \mathit{FA}\ [3\,..\,6]\ [(3, 1, 4), (4, 1, 5), (5, 0, 6), (6, 0, 3), (6, 1, 3)]\ [3\,..\,6]]
\end{array}$$

The above defined function $\mathit{optimizeDomains}$ when applied to $\mathit{fig7Domains}$ gives:

$$\begin{array}{l}
[\mathit{FA}\ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] \\
\quad [(1, 0, 9), (2, 0, 10), (3, 0, 11), (4, 0, 12), (5, 0, 13), (1, 1, 6), (2, 1, 7), (3, 1, 8), (4, 1, 7), \\
\quad\ (5, 1, 7), (6, 0, 1), (7, 0, 1), (8, 0, 1), (9, 0, 2), (10, 0, 2), (11, 0, 2), (12, 0, 2), (13, 0, 2)] \\
\quad [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13],
\end{array}$$

$$\begin{aligned}
&\mathit{FA}\ [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18],\\
&\quad [(1,1,6),(2,1,6),(3,1,6),(4,1,7),(5,1,7),(6,1,12),(7,1,12),(8,1,12),(9,1,12),\\
&\quad\ (10,1,12),(11,1,13),(12,0,16),(13,0,16),(14,0,16),(15,0,17),(16,0,1),(17,0,2),\\
&\quad\ (18,0,3),(16,1,4),(17,1,4),(18,1,5)],\\
&\quad [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18]]
\end{aligned}$$

(Compare FIG. 8 of [1]).

References

[1] C. S. McTague and J. P. Crutchfield. Automated pattern detection: An algorithm for constructing optimally synchronizing multi-regular language filters. *Theoretical Computer Science*. In press.

[2] S. L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, 1979.