On the Construction of Matchbound Certificates

Johannes Waldmann
Hochschule für Technik, Wirtschaft und Kultur
Leipzig, Germany
johannes.waldmann@htwk-leipzig.de

A *matchbound certificate* for a string rewriting system R is a labelled graph G that is closed with respect to a specific graph transformation depending on R.

We describe a new implementation of the certificate construction algorithm of Endrullis et. al. (2006), in a purely functional setting.

We use an extension of the (max,min) semiring, specific to this application; and a method for incremental computation of the set of occurrences of a labelled path in a graph, that may be of independent interest.

An intended side effect of this talk is to highlight the matchbound certificate construction problem as a potentially interesting test case for efficient methods and implementations of graph rewriting.

1 Matchbound Certificates

The matchbound method [5] proves termination of string rewriting (and recently, of cycle rewriting [8]). The method uses directed graphs where edges are labelled by a letter from a finite alphabet Σ, and a natural number.

An alternate view [7] is the following: The *fuzzy* semiring $\mathbb{F}$ has domain $\{-\infty\} \cup \mathbb{Z} \cup \{+\infty\}$, and semiring addition is "max", with neutral element (semiring zero) $-\infty$, also written $0_{\mathbb{F}}$, semiring multiplication is "min", with neutral element (semiring one) $+\infty$, sometimes written $1_{\mathbb{F}}$. A $\mathbb{F}$-weighted automaton A over Σ, with state set Q, is a mapping $A : \Sigma \to \mathbb{F}^{Q \times Q}$, where each letter gets assigned a Q-by-Q matrix of $\mathbb{F}$. Note that these matrices also form a semiring. That is why the mapping A can be extended from letters to words by matrix multiplication. So, the automaton computes a function $A : \Sigma^* \to \mathbb{F}^{Q \times Q}$.

The connection between both models is that the graph can be seen as a sparse representation of the matrices in the automaton. "Sparse" means that edges with weight zero ($0_{\mathbb{F}}$) are omitted. We have $A(w)(p,q) = f$ iff f is the maximum of the weights of all w-labelled paths from p to q, where the weight of one path is the minimum of the weight of its edges.

An $\mathbb{F}$-weighted automaton A is *compatible* with a string rewriting system R over Σ if for each $(l,r) \in R$, we have that $A(l) <_0 A(r)$ where $<_0$ denotes the point-wise extension (to matrices) of the relation $<_0$ on $\mathbb{F}$ defined by $x <_0 y$ iff $x < y \vee x = 0_{\mathbb{F}} = y$.

We remark that one could also consider the $(\min, \max)$ semiring. For matchbounds, we defined that a redex path should be covered by a *larger* reduct path, while for termination, we would expect that some weight is *decreasing*. This is just a matter of notation, as all operations and results are symmetric.

2 Construction of Matchbound Certificates

For constructing matchbound certificates, one starts with some trivial graph and adds paths whenever compatibility is violated. There is a method [6] that uses heuristics to re-use nodes when adding paths,

Submitted to:
TERMGRAPH

© J. Waldmann
This work is licensed under the
Creative Commons Attribution License.

but this may fail to terminate even if a certificate exists. There is a complete method [3] based on the idea of extending the alphabet by formal left and right inverses of letters. We focus on that method.

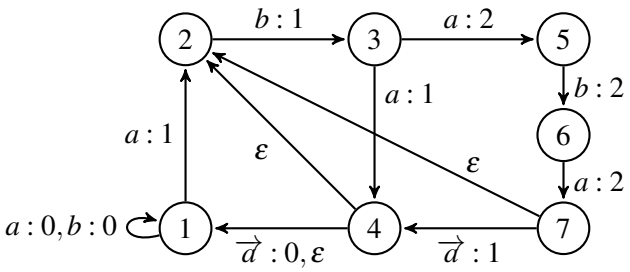
We introduce epsilon transitions. In the graph model, these are unlabelled edges. In the automaton model, we add a Q-by-Q matrix A_ε where each entry is $0_\mathbb{F}$ (no edge) or $1_\mathbb{F}$ (epsilon edge). We denote $A_\varepsilon(w) := A_\varepsilon A(w_1)A_\varepsilon \cdot \cdots \cdot A_\varepsilon A(w_n)A_\varepsilon$. Rules given below will ensure that (ultimately) A_ε is reflexively and transitively closed.

We extend the alphabet. For each $c \in \Sigma$, we introduce fresh letters $\overleftarrow{c}$, and $\overrightarrow{c}$. These will act as formal right and left inverses. We define $\overleftarrow{w} = \overleftarrow{w_n} \cdot \cdots \cdot \overleftarrow{w_1}$ and $\overrightarrow{w} = \overrightarrow{w_n} \cdot \cdots \cdot \overrightarrow{w_1}$.

Now, the algorithm to produce a matchbound certificate for R over Σ starts with the automaton A that has one state 1, a loop $1 \xrightarrow{c:0} 1$ for each letter $c \in \Sigma$, and A_ε as the identity relation, and repeatedly applies these rules.

1. [TRANS] if there are p,q with $(A_\varepsilon \cdot A_\varepsilon)(p,q) \neq 0_\mathbb{F}$, add epsilon transition from p to q.
2. [INV] if there are c,p,q with $A_\varepsilon(c\overleftarrow{c})(p,q) \neq 0_\mathbb{F}$, or $A_\varepsilon(\overrightarrow{c}c)(p,q) \neq 0_\mathbb{F}$, add epsilon transition from p to q. In fact, heights of inverses need to be taken into account, see Section 4.
3. [REWRITE] if there is $(l,r) \in R$ such that there is (p,q) such that $A_\varepsilon(l)(p,q) \not\leq_0 A_\varepsilon(r)(p,q)$, then:
   - let $p' \xrightarrow{c:h} q'$ with $c \in \Sigma, h \in \mathbb{F}$ be a transition of minimal height on a maximal l-labelled (p,q)-Path, such that $l = sct$ for $s,t \in \Sigma^*$,
   - then add a path from p' to q' over fresh states only that consists of a sequence of edges labelled by $\overrightarrow{s}$ with height h, r with weight $h+1$, $\overleftarrow{t}$ with height h.

Rule [REWRITE] should only be applied if none of [TRANS] and [INV] is applicable. The algorithm stops when no rule applies. The following example is for $R = \{aa \to aba\}$.



Reflexive ε edges (loops) are not shown. We apply rule [REWRITE] (for $p = p' = q' = q = 1$), producing the path $1 \xrightarrow{a:1} 2 \xrightarrow{b:1} 3 \xrightarrow{a:1} 4 \xrightarrow{\overrightarrow{d}:0} 1$, then [INV] (two times), producing $4 \xrightarrow{\varepsilon} 1$ and $4 \xrightarrow{\varepsilon} 2$, then [REWRITE] (for $p = p' = 3, q' = 4, q = 2$), then [INV].

It has been shown that this construction terminates iff R is matchbounded. In the example, matchbound (the highest label) is 2. Also, an efficient implementation had been given by Endrullis. It can build matchbound certificates with tens of thousands of states in a few seconds. The implementation was purpose-built, using an imperative programming style, in Java.

The goal of my current (and ongoing) work is to re-implement this method in a purely functional setting. My current implementation constructs the RFC matchbound certificate for `SRS/secret06/jambox1` (with 43495 states, for match height 12 — this is the "killer example" in [3]) in 8 seconds on a standard desktop computer. Novel features of the implementation are

- (specific to the matchbound application) how to find the location of the "pivot element" for applications of rule [REWRITE], see Section 4,
- (perhaps of more general interest) a method for incremental matching that updates occurrences of labelled paths in a graph while nodes and edges are being added, see Section 5.

3 Representing Weighted Relations

We make a few remarks here on representing relations, such that the operations of interest (multiplication, addition) can be realized efficiently. As we will explain in Section 5, we are particularly interested in multiplication where one of the arguments is small (contains few nonzero edges).

First, since we expect large matchbound certificates, we want to use a sparse representation. We assume some efficient implementation of finite maps, as given by `Data.Map` (balanced search tree) or `Data.IntMap` (Patricia tree) from the `containers` package for Haskell `http://hackage.haskell.org/package/containers`.

An S-weighted relation (a mapping $(P \times Q) \to S$) could literally be represented as `Map (p,q) s` but this is inefficient since we need to get all (nonzero) successors of a point $p \in P$ quickly. This suggests `Map p (Map q s)`, but now the implementation is biased: we cannot easily find predecessors - which we do need, for efficient multiplication. So, we also keep the representation in the opposite direction:

```
data Rel p q s = Rel { fore :: Map p (Map q s), back :: Map q (Map p s) }
```

To give an impression of the implementation:

```
plusWith :: (s -> s -> s) -> Rel p q s -> Rel p q s -> Rel p q s
plusWith f r s = Rel { fore = M.unionWith (M.unionWith f) (fore r)(fore s)
                     , back = M.unionWith (M.unionWith f) (back r)(back s)
                     }
timesWith :: (u -> u -> u) -> (s -> t -> u) -> Rel p q s -> Rel q r t -> Rel p r u
timesWith f g = M.foldl (plusWith f) empty
  $ M.intersectionWith (combine f g ) (back r) (fore s)

combine :: (c -> c -> c) -> (a -> b -> c) -> M.Map p a -> M.Map q b -> Rel p q c
combine f g qp qr = Rel { fore = M.map ( \ w1 -> M.map (\w2 -> f w1 w2) qr ) qp
                        , back = M.map ( \ w2 -> M.map (\w1 -> f w1 w2) qp ) qr
                        }
```

Function `M.intersectionWith` that combines finite maps m_1, m_2 is critical for performance. We expect it to work in $O(\min(|m_1|,|m_2|) \cdot \log\max(|m_1|,|m_2|))$ time.

This representation is similar to the one used in the `fgl` library [4].

```
type GraphRep a b = IntMap (Context' a b)
type Context' a b = (IntMap [b], a, IntMap [b])
```

The similarity is that two efficient maps are nested. The difference is that `fgl` graphs can have multiple edge labels (see `[b]`), while we just need one. This difference is not essential. The following is. The published `fgl` API (application programmer interface) has functions

```
match :: Node -> gr a b -> Decomp gr a b
type Decomp g a b = (MContext a b, g a b)
type MContext a b = Maybe (Context a b)
type Context a b = (Adj b, Node, a, Adj b)
type Adj b = [(b, Node)]
```

This represents sets (of neighbours of a node) as lists, which could be problematic. Imagine the computation of the set of common neighbours of two nodes with large degree. In particular, it is not obvious how to efficiently implement the product of two relations (matrices) in this representation.

4 An Extension of the (max,min) Semiring

While we were referring to the fuzzy (max,min) semiring $\mathbb{F}$ earlier, we do in fact need more information to realize matching efficiently. This section shows that we can still define a semiring structure on an enriched domain. This allows to use the semiring operations on the matrices.

The algorithm uses epsilon transitions. We said that epsilon edges have "no labels" but we can label them with weight $1_{\mathbb{F}}$. Then they behave reasonably w.r.t. multiplication. E.g., a matrix M (set of edges) is closed w.r.t. epsilon transitions from E if $E \cdot M \cdot E \leq M$. This refers to the natural order $a \leq c \iff \exists b : a + b = c$ in our idempotent semiring.

The algorithm requires closure w.r.t. rules like $\overrightarrow{c}c \to \varepsilon$, more precisely $\{\overrightarrow{c}_h c_{h'} \mid h \leq h'\}$. We extend the semiring by left and right inverted numbers $\overrightarrow{f}, \overleftarrow{f}$ for $f \in \mathbb{N}$, with multiplication rules

$$\overrightarrow{f} \cdot g = \text{if } f \leq g \text{ then } 1_{\mathbb{F}} \text{ else } 0_{\mathbb{F}}.$$

Then, the condition that the epsilon matrix E contains indeed all necessary transitions for inverses for letter $c \in \Sigma$, can be written $M_{\overrightarrow{c}} \cdot E \cdot M_c \leq E$.

The algorithm requires closure w.r.t. the operation of adding a path $\overleftarrow{s}\, r\, \overrightarrow{t}$ under certain conditions. We need to find an edge of minimal weight, among all maximal l-paths from p to q. We already have the weight of that edge — it is just the value in the matrix $A(l)$ (interpretation of l) at position (p,q). But we lost the information on where that minimal edge is located. This is repaired by carrying along the following extra information

```
data I = I { weight :: F -- ^ the original information
           , from  :: Q, to :: Q -- ^ start and end of edge with that weight
           , offset :: Int, total :: Int
           }
```

and these semiring operations

```
plus  i j = if weight i <= weight j then i else j -- ^ min operation
times i j = if weight i >= weight j -- ^ max operation, update offsets
  then i { total = total i + total j }
  else j { total = total i + total j , offset = total i + offset j }
```

where we use the Haskell notation of "record update" `base { name = val }` where components that are not mentioned in braces, have their value from `base`. So, when we have $A(l)(p,q) = i$, and need to find the decomposition into $l = sct$, we determine s as the prefix of length `offset i`, and the c edge is `(from i, to i)`.

We note that this satisfies the semiring axioms only up to some equivalence relation because the choice of result in case `weight i == weight j` is arbitrary.

5 Incremental Matching

To implement the algorithm given earlier, we have to compute and compare weights of paths that are labelled by left-hand sides and right-hand sides of rewrite rules. To do so efficiently, we use an incremental approach. After adding some nodes and edges to the graph, we do not want to re-scan the full graph to find the next match of an applicable rule.

In our model, where we view the graph (automaton) as a matrix interpretation over a semiring, the set of matches of a sequence w of labels is exactly the support (the set of indices for nonzero entries)

of $A(w)$. Consider A' which is A with some extra nodes and edges: $A' = A + \Delta$. Assume that w is decomposed in two words (e.g., of half length) $w = w_1 \cdot w_2$. Then $A(w)$ is the product of $\mathbb{F}$-matrices $A(w_1) \cdot A(w_2)$. We have $A'(w) = (A(w_1) + \Delta(w_1)) \cdot (A(w_2) + \Delta(w_2))$, so $A'(w) = A(w) + \Delta(w)$ with $\Delta(w) = \Delta(w_1)A'(w_2) + A'(w_1)\Delta(w_2)$. This gives a way to compute updates via matrix additions and multiplications.

As written, the algorithm is recursive (updates $\Delta(w_i)$ must be computed in the same manner) but we realize it bottom-up, using dynamic programming: Given a set L of sequences of labels (the left-hand sides of the graph rewriting rules), we determine a set of strings S such that

- S contains all one-letter words over the alphabet
- for $w \in S$ with $|w| > 1$, there are $w_1, w_2 \in S$, both shorter than w, with $w_1 w_2 = w$
- $L \subseteq S$.

We do this by repeatedly extracting a most frequent pair of letters, as in [1]. Then we store matrix products for all elements of S, and each time we add something to the graph, we update these matrices by processing S in order of increasing length.

We do not handle ε transitions specially. Rather, we treat ε as an extra letter. For computing the matchbound certificate for $\{aa \to aba\}$, we need to match these paths: $\varepsilon\varepsilon$ for rule [TRANS], $\{a\varepsilon\overleftarrow{a}, b\varepsilon\overleftarrow{b}, \overrightarrow{a}\varepsilon a, \overrightarrow{b}\varepsilon b\}$ for rule [INV], and $\{a\varepsilon a, \varepsilon a\varepsilon b\varepsilon a\varepsilon\}$ for rule [REWRITE]. The longer string (from the right-hand side of the rewrite rule) is needed to check the condition for rule [REWRITE] firing. It is decomposed as $\varepsilon \cdot ((a \cdot \varepsilon) \cdot ((b \cdot \varepsilon) \cdot (a \cdot \varepsilon)))$.

6 Streams of Relations

Using the ideas presented above, the current implementation has a main program that repeatedly performs graph rewrite rule applications. With an "imperative" viewpoint, the graph is updated (it could be in-place); in a "functional" setting, a list of graphs is produced.

I think the functional viewpoint is more interesting, and could lead to a program that "looks better" (shorter, and easier to prove correct).

The limit automaton A of the sequence $A_0, A_1, \ldots$ can be seen as a mapping $A : \Sigma \to M^\omega$ where M is the set of matrices over $\mathbb{F}$, and M^ω denotes monotone infinite sequences (streams) of such matrices. Then, A is specified as the solution of a system of equations between these streams.

The challenge is to implement the stream operations and write the equations in such a way that the (oriented) system is *productive* [2] so that the limit can be computed. For efficiency, it is better to represent a sequence $(M)_k$ as a sequence of partial sums of $(\Delta)_k$, Then most of Δ_k are expected to be small (graphs with few edges). Section 5 showed operations on individual Δ_k matrices, while now the idea is to extend this to streams.

Stream operations of interest include addition and multiplication. Representing a stream of S as the Haskell data type `[s]` (list of `s`), we write

```
instance Semiring s => Semiring [s] where
  plus xs ys = purge $ zipWith plus xs ys
```

To get the difference representation, we define

```
purge :: Semiring s => [s] -> [s]
purge xs = go zero xs where
  go done (x:xs) = minus x done : go (plus done x) xs
```

Here, `minus x y` should return an element `z` such that `x = plus y z`. This works since the natural order on $\mathbb{F}$ is total. For multiplication, the following code shows how we compute the result incrementally:

```
instance Semiring s => Semiring [s] where
  times xs ys = purge $ go (zero,xs) (zero,ys) where
    go (xdone,x:xs) (ydone,y:ys) =
        plus (times x y) ( plus (times xdone y) (times x ydone) )  -- (*)
      : go (plus xdone x,xs) (plus ydone y, ys)
```

Note that `times` in the line marked `(*)` are used at the base type `s`.

Open problems for this line of implementations are: precedence of rules ([TRANS], [INV] $>$ [REWRITE) must be respected, and some difference-like operation is needed to describe the condition in [REWRITE] — namely, that some other path is (still) missing. This is challenging because addition and multiplication are monotonic, but difference is not.

7 Summary and Discussion

The contributions of this note are

- an enhanced fuzzy semiring, to aid in implementing matchbound computations, see Section 4,
- a method for incremental matching of paths in labelled graphs, see Section 5.

These have already been used in my recent re-implementation of Endrullis' algorithm (`https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox`). Performance is in the same ballpark as the original implementation. Implementation of the ideas presented in Section 6 is ongoing work.

I recommend matchbound certificate construction as a test case for graph rewriting. It has simple rules, given in Section 2. An implementation should handle graphs of $\approx 10^4$ nodes in a few seconds.

References

[1] Alexander Bau, Markus Lohrey, Eric Nöth, and Johannes Waldmann. Compression of rewriting systems for termination analysis. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPIcs*, pages 97–112. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[2] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. *Theor. Comput. Sci.*, 411(4-5):765–782, 2010.

[3] Jörg Endrullis, Dieter Hofbauer, and Johannes Waldmann. Decomposing terminating rewrite relations. In *Workshop on Termination*, pages 39–43, 2006.

[4] Martin Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, 2001.

[5] Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Match-bounded string rewriting systems. *Appl. Algebra Eng. Commun. Comput.*, 15(3-4):149–171, 2004.

[6] Alfons Geser, Dieter Hofbauer, Johannes Waldmann, and Hans Zantema. Finding finite automata that certify termination of string rewriting systems. *Int. J. Found. Comput. Sci.*, 16(3):471–486, 2005.

[7] Johannes Waldmann. Weighted automata for proving termination of string rewriting. *Journal of Automata, Languages and Combinatorics*, 12(4):545–570, 2007.

[8] Hans Zantema, Barbara König, and H. J. Sander Bruggink. Termination of cycle rewriting. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 476–490. Springer, 2014.