Logical specification of transducers for NLP

Nathan Vaillette
Ohio State University and the Universität Tübingen

1 Overview This paper concerns applications of an algorithm that converts logical formulas which specify regular languages into finite-state machines accepting the corresponding languages. We provide an extension to the logic for a limited class of regular relations. We then demonstrate the usefulness of this logic for specifying the sort of finite-state transducers as are used in natural language processing by providing a simple logical definition of the conditional replacement operator.

2 Introduction Regular languages and relations can be encoded using regular expressions, which can be compiled to finite-state machines. This frees us from the need to work with the machines directly, thus providing a higher-level description language. However, this kind of description is not always the most convenient. It would also be useful to be able to describe regular sets in a logic. In many situations, this would be closer to the thought process and would provide a kind of executable specification for automata. Such a declarative approach is also advantageous for verification of correctness.

Monadic second-order logic (MSOL) provides an expressive framework with an important connection to regular sets. Theoretical work on the decidability of restrictive theories of arithmetic in [1] and [2] established an equivalence between certain MSOL languages and finite-state automata. Applications of the theory were not undertaken for some time, because of its extremely daunting complexity: in the worst case, the space requirements of the conversion from logic formulas to automata is bounded from below by a stack of exponentials proportional in size to the number of quantifiers in the formula [7]. However, recent implementations such as the the MONA [10] and MOSEL [8] projects have demonstrated that the worst case need not be a problem in practice.

The implementation described here is significantly less ambitious and general than these projects and unlike them does not use special techniques for automata representation. However, it has several features that are advantageous, especially for NLP. It is implemented within the FSA Utilities, an open-source finite-state toolkit [14] which has been used in several NLP applications. First of all, it is convenient to integrate the logic into such a toolkit. Regular expressions and defined macros of the Utilities can be mixed with and even embedded in logic formulas. Secondly, the Utilities extends the formalism of finite state machines slightly by providing a direct treatment of predicates and open (i.e. infinite) alphabets. This means that a transition in an automaton need not only read a single symbol or one of a set of symbols, but can also read a specification like `not_in([dog,cat])`. Such a transition accepts any symbol besides `dog` or `cat`, thus recognizing an infinite set. Finally, the Utilities also supports finite-state transducers, and in particular, transducers with open alphabets and predicates and identity transitions over these. For example, a identity transition can be specified to map any symbol besides `dog` or `cat` to itself. This is particularly useful for our extension of the logic to regular relations.

3 The logic The particular logical language we use is a variant of MSO(Str) (cf. [13]). On our interpretation, a formula denotes a language over an alphabet of symbols. The logic has monadic second-order variables, which are interpreted as sets of positions (locations of symbol tokens) in a string. (It lacks first-order variables, but these can be simulated using the `singleton` predicate.) Intuitively, a formula states a restriction and denotes the set of all strings that obey that restriction. The syntax we use will be based on terms of Prolog, in which the Utilities are implemented. A MSOL variable will be written as a Prolog atom wrapped in `v(...)`. The basic atomic formulas of the logic are as follows. `singleton(v(x))` says that the set of positions `v(x)` denotes is a singleton. `subset(v(x),v(y))` states a subset relation between two sets. `precedes(v(x),v(y))` states that `v(x)` and `v(y)` are both strings of positions, and that `v(x)` directly precedes `v(y)`. Finally, `matches(v(x),R)`, states that `v(x)` is a string of positions and expresses a matching relationship between `v(x)` and a regular expression `R`. Whereas MSO(Str) only has predicates relating a set to a single alphabet symbol, our version with `matches` allows arbitrary regular expressions to be embedded into logic formulas. `matches(v(x),R)` holds when `v(x)` is a member of the language denoted by `R`. The regular expression notation of the Utilities used in this paper is given in table 1.

The language also has connectives and quantifiers `and(F1,F2)`, `or(F1,F2)`, `not(F)`, `all(v(x),F)`, and

`[]`	empty string
`[E1,...En]`	concatenation of `E1 ...En`
`{E1,...En}`	union of `E1,...En`
`E*`	Kleene closure
`~E`	complement
`` `E ``	term complement
`E1 & E2`	intersection
`?`	any symbol
`E1 x E2`	cross-product
`E1 xx E2`	same-length cross-product
`A o B`	composition
`id(E)`	identity transduction

Table 1: Regular expression operators

`exists(v(x),F)` with obvious interpretations. In the implementation, derived formulas can be defined as macros. The languages describable by formulas of the logic are exactly the regular languages over an open alphabet.

An example formula is given in figure 1. It states that every occurrence of `[a,b]` must be directly followed by an occurrence of `[c,d]`. This denotes the same language as the regular expression `~[? *,a,b,~[c,d,? *]]`, but in a more transparent way.

```
all(v(x),
    if(
       matches(v(x),[a,b]),
       %% then
       exists(v(y),
              and(
                  matches(v(y),[c,d]),
                  precedes(v(x),v(y))
                 ))))
```

Figure 1: Example formula

4 Logic to automata The central insight which allows the conversion of the logic formulas to automata is the use of multi-tape automata, with extra tapes added to represent the interpretation of the free variables of a formula. Each free variable has its own tape with boolean values on it. The tape for variable `v(x)` has `T` in every position which is in the interpretation of `v(x)`, and `F` everywhere else. The interpretation of the formula as a language is obtained simply by stripping off the extra tapes. The reader is referred to the references cited for more details.

In our implementation, we simulate multi-tape transducers on one-tape machines by interleaving the tapes. We recursively compile logic formulas to (pseudo-)multi-tape regular expressions, which the Utilities compiles to automata.

5 A logic for regular relations The basic technique used to associate logic formulas with automata recognizing regular languages can be carried over to transducers for regular relations. A formula now characterizes a relation by restricting the pairs of strings that can occur in it. However, the problem arises that negation and conjunction of formulas involves complementing and intersecting the corresponding automata. Therefore, it is necessary for the class of regular languages used to be closed under these operations. The entire class of regular relations is unfortunately not. Our solution will be to use only same-length relations. These are ones which can be represented in a transducer in which each transition is over an pair (or more generally, n-tuple) of symbols. The variables are interpreted as sets of positions in the aligned pair of strings. Although this is a rather severe restriction, we will see in the next section that it does not necessarily preclude us from using the logic in the specification of a larger class of relations.

To describe relations, we simply allow relational regular expressions as the second argument of `matches`. The one complication that arises is that although the Utilities supports identity transitions over an open alphabet, it does not have a way to represent their negation, which would require a transition that reads any symbol in a set and writes any *distinct* symbol in that set. We circumvent this by using an modified representation internally where every transition is followed by either a `0`, `1`, or `2` to indicate whether it is a normal, identity, or distinctness transition, where the latter is used in the negation of identity. This means that the logic can represent transductions that cannot be converted back into the normal transducer representation of the Utilities, e.g. `not(matches(v(x),id(?)))` since they contain distinctness transductions. These however are not needed in practice.

The next section explores an application of this logic to the definition of the replace operator.

6 The replace operator Several varieties of finite-state replacement operators have been proposed, such as in [3], [4], [5], [6], [9], and [12]. These have in common that they implement as a transducer replacements like $\phi \rightarrow \psi/\lambda\_\_\rho$, which replaces every occurrence of ϕ in the input which occurs between λ and ρ with ψ. All of these implementations have a procedural nature, based on a cascade of transducers where special marker symbols are inserted to delimit regions and are later deleted. They can become rather complicated, and it would be hard to rigorously prove their correctness. However, we can give a purely declarative definition in the logic which has a clear connection to the intended interpretation.

[6] defines an unconditional replacement operator $\phi \rightarrow \psi$ as a regular relation made of regions where input ϕ is paired with output ψ, alternating with regions

```
macro(to_replace(v(S),Phi),
      matches(v(S),Phi xx ? *)).

macro(to_preserve(v(S),Phi),
      not(exists(v(S2),
             subset(v(S2),v(S)),
             to_replace(v(S2),Phi)))).

macro(in_replaced_region(v(X),Phi,Psi),
      exists(v(S),
         subset(v(X),v(S)),
         to_replace(v(S),Phi),
         matches(v(S),Phi x Psi),
         exists(v(S2),
            to_preserve(v(S2),Phi),
            matches(v(S2),id(?) *),
            precedes(v(S),v(S2))))).
```

Figure 2: Preliminary definitions for the unconditional replace operator

```
%% definition of Phi -> Psi

all(v(X),
    if(
       singleton(v(X)),
       %% then
       or(
          in_replaced_region(v(X),Phi,Psi)
          in_preserved_region(v(X),Phi,Psi)
         ))).
```

Figure 3: The unconditional replacement operator

that don't contain any occurrences of ϕ in the input and that output the input unchanged. We can define this similarly in the logic. Figure 2 gives some preliminary macro definitions. (The first argument of `macro` is a template which expands into the second argument, the uppercase symbols being parameters which are instantiated. For perspicuity, we allow formulas of the form `exists(v(V),F1,...Fn)` where the `F`s are interpreted conjunctively.)

A string is `to_replace` if it has the right input; it is `to_preserve` if it doesn't contain anything to replace. The region `v(S)` in the definition of `in_replaced_region` is one that is `to_replace` and actually gets replaced (matches `Input x Output`) and is followed by a region that is `to_preserve` and actually is preserved. A parallel definition can be given for `in_preserved_region`, true of a preserved region followed by a replaced one. With this machinery in place, the definition of the unconditional replacement operator in figure 3 states an alternation between such regions.

```
macro(cond_to_replace(v(S),Phi,Lambda,Rho),
      matches(v(S),Phi xx ? *),
      and(
          exists(v(l),
             matches(v(l),? * xx Lambda),
             precedes(v(l),v(S))),
          exists(v(r),
             matches(v(r),Rho xx ? *),
             precedes(v(S),v(r))))).
```

Figure 4: Conditional version of to_replace

This definition of the unconditional replacement operator is by itself no improvement on Karttunen's; in fact it is a bit more complicated. The advantage that it offers is that it can be extended very simply. First of all, to turn it into a conditional replacement operator, all we need to do is redefine `to_replace` to also look at the context, as in figure 4. This makes figure 3 into a definition of $\phi \rightarrow \psi / \lambda\_\rho$ without any further work. (Note that the context check in figure 4 is rightward oriented in the sense of [6] because it checks left context on the output side and right context on the input side. Obviously, other orientations can be defined just as simply.)

The definition of `to_replace` can be further restricted to yield different varieties of replacement. For instance, `v(S)` can also be constrained not to be a proper prefix of any other string with the right input in the right context. This can be used in a directed replacement operator that does a left-to-right longest-match replacement (though further restrictions are necessary to ensure the right directionality).

The operators defined here use a logic of same-length relations, so they cannot handle any replacements involving insertions or deletions, i.e. where `Phi x Psi` is not a same-length relation. However, this restriction can be easily circumvented by using a special alphabet symbol `0` to mimic nulls, as is done in two-level morphology [11]. Deletions are handled straightforwardly by mapping input to `0` and then composing the replace operator with a transducer that removes all `0`s from the output. For instance, `([a,b] -> [c,0] / x ___ y) o {id('0),0 x[]}*` replaces any occurrence of `[a,b]` between `x` and `y` with `c`. This is no problem, since `[a,b] x [c,0]` is a same-length relation.

For insertions, we can likewise compose the replace operator with a transducer that inserts `0`s into the input. However, this will produce spurious outputs, since the transducer can't generally know where to insert the `0`s. We can solve this problem by simply using a different definition of `matches`, according to which the input side is not required to match some L, but rather L with any number of `0`s inserted anywhere. With this mechanism, 0s in the right place will match as desired, while at the same time 0s in the wrong place will not block matching. This shows that the restriction of the logic to same-length relations need not necessarily be an obstacle to using it in the description of transducers for regular relations in general.

7 Future work The usefulness of the logic of regular relations in defining replacement operators should carry over to the declarative specification of other transducers used in NLP. In a related vein, logical specification of transducers is conceptually similar to the rules of two-level morphology. Both describe a mapping by constraining possible input-output correspondences. Therefore, it seems that the logic could be useful as more flexible formalism for stating two-level rules than that of [11]. Lastly, since most of the quantification we have used has been over strings of positions instead of arbitrary, possibly discontinuous sets, it would be interesting to explore whether a logic with more restricted quantification would have desirable properties.

References

[1] J.R. Büchi. Weak second order arithmetic and finite automata. *Zeitschr. f. math. Logik und Grundlagen d. Math.* 6:66–92, 1960.

[2] C. C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.* 98:21–52, 1961

[3] D. Gerdemann and G. van Noord. Transducers from Rewrite Rules with Backreferences. In *EACL 99*, Bergen, Norway, 1999.

[4] Ronald Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–379, 1994.

[5] Lauri Karttunen. Directed replacement. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, 1996.

[6] Lauri Karttunen. The replace operator. In Emannual Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 117–147. Bradford, MIT Press, 1997.

[7] A. R. Meyer Weak monadic second-order theory of successor is not elementary recursive. In R. Parikh (ed.), *Logic Colloquium (Proc. Symposium on Logic, Boston 1972), LNCS* 453, pages 132–154, 1975.

[8] P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. MOSEL: A Flexible Toolset for MOnadic SEcond-order Logic. In *Proc. TACAS'97, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, LNCS* 1217, pages 183–202. Springer Verlag, 1997.

[9] A. Kempe and L. Karttunen Parallel Replacement in Finite State Calculus In *COLING-96*, Copenhagen, 1996.

[10] N. Klarlund and A. Møller. *MONA Version 1.4 User's Manual.* Department of Computer Science, Aarhus, Denmark, 2001. Available from `http://www.brics.dk/mona/manual.html`.

[11] Kimmo Koskenniemi. *Two-level Morphology: a General Computational Model for Word-form Recognition and Production.* Technical report 11, Department of General Linguistics, University of Helsinki, Finland, 1983.

[12] Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, 1996.

[13] Wolfgang Thomas. *Languages, Automata, and Logic* Bericht 9607, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996.

[14] Gertjan van Noord. *FSA6 Reference Manual.* Alfa-informatica, University of Groningen. Available from `http://odur.let.rug.nl/~vannoord/Fsa/Manual/`