TTT: A tree transduction language for syntactic and semantic processing

Abstract

In this paper we present the tree to tree transduction language, TTT. We motivate the overall "template-to-template" approach to the design of the language, and outline its constructs, also providing some examples. We then show that TTT allows transparent formalization of rules for parse tree refinement, parse correction, predicate disambiguation, and refinement, inference, and verbalization of logical forms.

1 Introduction

Pattern matching and pattern-driven transformations are fundamental tools in AI. Many symbol manipulation tasks including operations on parse trees and logical forms, and even inference and aspects of dialogue and translation can be couched in the framework of pattern-directed transduction applied to list-structured symbolic expressions or trees.

The TTT system is directly applicable to concise and transparent specification of rules for such tasks, in particular (as we will show), parse tree refinement, parse tree correction, predicate disambiguation, logical form refinement, inference, and verbalization into English.

In parse tree refinement, our particular focus has been on repair of malformed parses of image captions, as obtained by the Charniak-Johnson parser (Charniak and Johnson, 2005). This has encompassed such tasks as distinguishing passive participles from past participles and temporal nominals from non-temporal ones, assimilation of verb particles into single constituents, deleting empty constituents, and particularizing prepositions. For example, standard treebank parses tag both past participles (as in "has written") and passive participles (as in "was written") as VBN. This is undesirable for subsequent compositional interpretation, as the meanings of past and passive participles are distinct. We can easily relabel the past participles as VBEN by looking for parse tree subexpressions where a VBN is preceded by a form of "have", either immediately or with an intervening adverb or adverbial, and replacing VBN by VBEN in such subexpressions. Of course this can be accomplished in a standard symbol manipulation language like Lisp, but the requisite multiple lines of code obscure the simple nature of the transduction.

We have also been able to repair systematic PP (prepositional phrase) misattachments, at least in the limited domain of image captions. For example, a common error is attachment of a PP to the last conjunct of a conjunction, where instead the entire conjunction should be modified by the PP. Thus when a statistically obtained parse of the sentence " Tanya and Grandma Lillian at her highschool graduation party" brackets as "Tanya and (Grandma Lillian (at her highschool graduation party.))", we want to lift the PP so that "at her highschool graduation party" modifies "Tanya and Grandma Lillian".

Another systematic error is faulty classification of relative pronouns/determiners as wh-question pronouns/determiners, e.g., "the student *whose* mother contacted you" vs. "I know *whose* mother contacted you" – an important distinction in compositional semantics. (Note that only the first occurrence, i.e., the relative determiner, can be paraphrased as *with the property that his*, and only the second occurrence, in which *whose* forms a wh-nominal, can be paraphrased as *the person with* *the property that his.*) An important point here is that detecting the relative-determiner status of a wh-word like *whose* may require taking account of an arbitrarily deep context. For example, in the phrase "the student in front of whose parents you are standing", *whose* lies two levels of phrasal structure below the nominal it is semantically bound to. Such phenomena motivate the devices in TTT for detecting "vertical patterns" of arbitrary depth. Furthermore, we need to be able to make local changes "on the fly" in matching vertical patterns, because the full set of tree fragments flanking a vertical match cannot in general be saved using match variables. In the case of a wh-word that is to be re-tagged as a relative word, we need to rewrite it *at the point where the vertical pattern matches it*, rather than in a separate tree-(re)construction phase following the tree-matching phase. .

An example of a discourse phenomenon that requires vertical matching is anaphoric referent determination. In particular, consider the well-known rule that a viable referent for an anaphoric pronoun is an NP that C-commands it, i.e., that is a (usually left) sibling of an ancestor of the pronoun. For example, in the sentence "John shows Lillian the snowman that he built", the NP for *John* C-commands the pronominal NP for *he*, and thus is a viable referent for it (modulo gender and number agreement). We will later show a simple TTT rule that tags such an anaphoric pronoun with the indices of its C-commanding NP nodes, thus setting the stage for semantic interpretation.

We have been also able to perform Skolemization, conjunct separation, simple inference, and logical form verbalization with TTT and suspect its utility to logic tasks will increase as development continues.

A beta version of our system will be made available; however the URL is not included in this paper for anonymity.

2 Related Work

There are several pattern matching facilities available; however, none proved sufficiently general and perspicuous to serve our various purposes.

The three related tools Tgrep, Tregex, and Tsurgeon provide powerful tree matching and restructuring capabilities (Levy and Andrew, 2006). However, Tgrep and Tregex provide no transduction mechanism, and Tsurgeon's modifications are limited to local transformations on trees. Also, it presupposes list structures that begin with an atom (as in Treebank trees, but not in parse trees with explicit phrasal features), and its patterns are fundamentally tree traversal patterns rather than tree templates, and can be quite hard to read.

Peter Norvig's pattern matching language, "pat-match", from (**?**) provides a nice pattern matching facility within the Lisp environment, allowing for explicit templates with variables (that can bind subexpressions or sequences of them), and including ways to apply arbitrary tests to expressions and to match boolean combinations of patterns. However, there is no provision for "vertical" pattern matching or subexpression replacement "on the fly". TTT supports both horizontal and vertical pattern matching, and both global (output template) and local (on the fly) tree transduction. Also the notation for alternatives, along with exclusions, is more concise than in Norvig's matcher, for instance not requiring explicit ORs. While pat-match supports matching multi-level structures, the pattern operators are not composable – a feature present in TTT that we have found to be quite useful.

Mathematica also allows for sophisticated pattern matching, including matching of sequences and trees. It also includes a sophisticated expression rewriting system, which is capable of rewriting sequences of expressions. It includes functions to apply patterns to arbitrary subtrees of a tree until all matches have been found or some threshold count is reached; as well, it can return all possible ways of applying a set of rules to an expression. However, as in the case of Norvig's matcher there is no provision for vertical patterns or on-the-fly transduction. (Wolfram Research, Inc, 2010)

Snobol, originally developed in the 1960's, is a language focused on string patterns and string transformations (Griswold, 1971). It has a notably different flavor to the other transformation systems. Its concepts of cursor and needle support pattern based transformations that rely on the current position in a string at pattern matching time, as well as the strings that the preceding patterns matched up to the current point. Snobol also supports named and thereby recursive patterns. While it includes recognition of balanced parenthesis, the expected data type for Snobol is the string – leaving it a less than direct tool for intricate manipulation of trees. An Python extension SnoPy adds Snobol's pattern matching capabilities to Python. (Rozenberg, 2002)

Haskell also includes a pattern matching system, but it is weaker than the other systems mentioned. The patterns are restricted to function arguments, and are not nearly as expressive as Mathematica's for trees nor Peter Norvig's system or Snobol for strings. (Hudak et. al, 2000)

3 TTT

Pattern Matching

Patterns in TTT are hierarchically composed of sub-patterns. The simplest kind of pattern is an arbitrary, explicit list structure (tree) containing no match operators, and this will match only an identical list structure. Slightly more flexible patterns are enabled by the "underscore operators" `_!`, `_+`, `_?`, `_*`. These match any single tree, any non-empty sequence of trees, the empty sequence or a sequence of one tree, and any (empty or non-empty) sequence of trees. These operators (as well as all others) can also be thought of as match variables, as they pick up the tree or sequence of trees they match as their binding.

The bindings are "non-sticky", i.e., an operator such as `_!` will match any tree, causing replacement of any prior binding (within the same pattern) by that tree. However, bindings can be preserved in two ways: by use of new variable names, or by use of sticky variables. New variable names are obtained by appending additional characters – conventionally, digits – to the basic ones, e.g., `_!1`, `_!2`, etc. Sticky variables are written with a dot, i.e., `_!.`, `_+.`, `_?.`, `_*.`, where again these symbols may be followed by additional digits or other characters. The important point concerning sticky variables is that multiple occurrences of such a variable in a pattern can only be bound by the same unique value. Transductions are specified by a special pattern operator `/` and will be described in the next section.

More flexible operators, allowing for alternatives, negation, and vertical patterns among other constructs, are written as a list headed by an operator without an underscore, followed by one or more arguments. For example, `(! A (B C))` will match either the symbol `A` or the list `(B C)`, i.e., the two arguments provide alternatives. As an example involving negation, `(+ A (B _!) ~ (B B))` will match any nonempty sequence whose elements are `A`s or two-element lists headed by `B`, but disallowing elements of type `(B B)`. Successful matches cause the matched expression or sequence of expressions to become the value of the operator. Again, sticky versions of match operators use a dot, and the operators may be extended by appending digits or other characters.

The ten basic argument-taking pattern operators are:

- `!` - Match exactly one sub-pattern argument.
- `+` - Match a sequence of one or more arguments.
- `?` - Match the empty sequence or one argument.
- `*` - Match the empty sequence or one or more arguments.
- `{}` - Match any permutation of the arguments.
- `<>` - Match the freestanding sequence of the arguments.
- `^` - Match a tree that has a child matching one of the arguments.
- `^*` - Match a tree that has a descendant matching one of the arguments.
- `^@` - Match a vertical path.
- `/` - Attempt a transduction. (Explained later.)

Various examples will be provided below. Any of the arguments to a pattern operator may be composed of arbitrary patterns.

Negation: The operators `!`, `+`, `?`, `*`, and `^` support negation (pattern exclusion); i.e., the arguments of these operators may include not only alternatives, but also a negation sign ~ (after the alternatives) that is immediately followed by one or more precluded patterns. If no alternatives are provided, only precluded patterns, this is interpreted as "anything goes", except for the precluded patterns. For example, `(+ ~ (A A) (B B))` will match any nonempty sequence of expressions that contains no elements of type `(A A)` or `(B B)`.

Conjunction: We have so far found no compelling need for an explicit conjunction operator. Of course, any pattern calling for a structured tree is by its nature conjunctive – all the tree components called for must be present. If necessary, a way to say that a tree must match each of two or more patterns is to use double negation. For example, suppose we want to say that an expression must begin with an `A` or `B` but must contain an `A` (at the top level); this could be expressed as
`(! ~ (! ~ ((! A B) _*) (_* A _*)))`.
However, this would be more perspicuously expressed in terms of alternatives, i.e.,
`(! (A _*) (B _* A _*))`. We also note that the allowance for computable predicates (discussed below) enables introduction of a simple construct like
`(! (and? patt1 patt2))`,
where `patt1` and `patt2` are arbitrary TTT patterns, and `and?` is an executable predicate that applies the TTT matcher to its arguments and returns a non-nil value if both succeed and nil otherwise. In the former case, the binding of the outer `!` will become the matched tree.

Bounded Iteration: The operators `!`, `+`, `?`, `*`, and `^` also support iterative bounding, using square brackets. This enables one to write patterns that match exactly n, at least n, at most n, or from n to m times, where n and m are integers. Eg. `(![3] A)` would match the sequence `A A A`. The vertical operator `^[n]` matches trees with a depth `n` descendant that matches one of the operator's arguments.

Vertical Paths: The operators `^*` and `^@` enable matching of vertical paths of arbitrary depth. The first, as indicated, requires the existence of a descendant of the specified type, while the second, with arguments such as `(^@` P_1 P_2 `...` P_n`)` matches a tree whose root matches P_1, and has a child matching P_2, which in turn has a child matching P_3, and so on. Note that this basic form is indifferent to the point of attachment of each successive offspring to its parent; but we can also specify a point of attachment in any of the P_1, P_2, etc., by writing `@` for one of its children. Note that the argument sequence P_1 P_2 `...` can itself be specified as a pattern (e.g., via `(+ ...)`), and in this case there is no advance commitment to the depth of the tree being matched.

Computable Predicates: Arbitrary predicates can be used during the pattern matching process (and consequently the transduction process). Symbols with names ending in the question mark, and with associated function definitions, are interpreted as predicates. When a predicate is encountered during pattern matching, it is called with the current subtree as input. The result is only inspected as far as nil/non-nil, and when nil is returned the current match fails. Addionally, supporting user-defined predicates enables the use of named patterns.

Some Example Patterns: Here are examples of particular patterns, with verbal explanations. Also see Table 1, at the top of the next page, for additional patterns with example bindings.

- `(! (+ A) (+ B))`
  Matches a non-empty sequence of `A`'s or a non-empty sequence of `B`'s, but not a sequence containing both.

- `(* (<> A A))`
  Matches an even number of `A`'s.

- `(B (* (<> B B)))`
  Matches an odd number of `B`'s.

- `(({} A B C))`
  Matches `(A B C)`, `(A C B)`, `(B A C)`, `(B C A)`, `(C A B)` and `(C B A)` and nothing else.

- `((<> A B C))`
  Matches `(A B C)` and nothing else.

- `(^* X)`
  Matches any tree that has descendant `X`.

- `(^@ (+ (@ _*)) X)`
  Matches any tree with leftmost leaf `X`.

Transductions

Transductions are specified with the transduction operator, `/`, which takes two arguments. The left argument may be any tree pattern and the right argument may be constructed of literals, variables from the lhs pattern, and function calls.

Transductions may be applied to the roots of trees or arbitrary subtrees, and they may be restricted to apply at most once, or until convergence. When applying transductions to arbitrary subtrees, trees are searched top-down, left to right. When a match to the transduction lhs pattern occurs, the resulting bindings and transduction rhs

Pattern	Tree	Bindings
`_!`	`(A B C)`	`(_! (A B C)`
`(A _! C)`	`(A B C)`	`(_! B)`
`(_* F)`	`(A B (C D E) F)`	`(_* A B (C D E))`
`(A B _? F)`	`(A B (C D E) F)`	`(_? (C D E))`
`(A B _? (C D E) F)`	`(A B (C D E) F)`	`(_? )`
`(^@ _! (C _*) E)`	`(A B (C D E) F)`	`(^@ (A B (C D E) F)) (_* D E)`
`(A B (<> (C D E)) F)`	`(A B (C D E) F)`	`(<> (C D E))`
`(A B (<> C D E) F)`	`(A B (C D E) F)`	`nil`

Table 1: Binding Examples

are used to create a new tree, that then replaces the tree (or subtree) that matched the lhs.

Here are a few examples of simple template to template transductions:

- `(/ X Y)`
  Replaces the symbol `X` with the symbol `Y`.
- `(/ (! X Y Z) (A))`
  Replaces any `X`, `Y`, or `Z` with `A`.
- `(/ (! X) (! !))`
  Duplicates an `X`.
- `(/ (X _* Y) (X Y))`
  Remove all subtrees between `X` and `Y`.
- `(/ (_! _* _!1) (_!1 _* _!))`
  Swaps the subtrees on the boundaries.

A transduction operator may appear nested within a composite pattern. The enclosing pattern effectively restricts the context in which the transduction will be applied, because only a match to the entire pattern will trigger a transduction. In this case, the transduction is applied at the location in the tree where it matches. The rhs of such a transduction is allowed to reference the bindings of variables that appear in the enclosing pattern. We call these local transductions, as distinct from replacement of entire trees. Local transductions are especially advantageous when performing vertical path operations, allowing for very concise specifications of local changes. For example, the transduction

```
(^@ (* ((! S SBAR) _+))
    (/ (WH _!)
       (REL-WH (WH _!)))))
```

wraps `(REL-WH ...)` around a `(WH ...)` constituent occurring as a descendant of a vertical succession of clausal (`S` or `SBAR`) constituents. Applied to the tree `(S (SBAR (WH X) B) A)`, this yields the new tree `(S (SBAR (REL-WH (WH X)) B) A)`. Additional examples appear later (especially in the parse tree refinement section).

TTT also supports constructive functions, with bound variables as arguments, in the rhs templates, such as `join-with-dash!`, which concatenates all the bound symbols with intervening dashes, and `subst-new!`, which will be discussed later. One can imagine additional functions, such as `reverse!`, `l-shift!`, `r-shift!`, or any other function of a list of nodes which may be useful to the application at hand. Symbols with names ending in the exclamation mark are assumed to be associated with function definitions, and when appearing as the first element of a list are executed during output template construction. To avoid writing many near-redundant functions, we use the simple function `apply!` to apply arbitrary Lisp functions during template construction.

Theoretical Properties

A good overview of the dimensions of variability among formal tree transducers is given in (Knight, 2007). The main properties are restrictions on the height of the tree fragments allowed in rules, linearity, and whether the rules can delete arbitrary subtrees. Among the more popular and recent ones, synchronous tree substitution grammars (STSG), synchronous tree sequence substitution grammars (STSSG), and multi bottom-up tree transducers (MBOT) constrain their rules to be linear and non-deleting, which is important for efficient rule learning and transduction execution (Chiang, 2004; Galley et. al, 2004; Yamada and Knight, 2001; Zhang et. al, 2008; Maletti, 2010).

The language TTT does not have any such

restrictions, as it is intended as a general programming aid, with a concise syntax for potentially radical transformations, rather than a model of particular classes of linguistic operations. Thus, for example, the 5-element pattern `(! ((* A) B) ((* A) C) ((* A) D) ((* A) E) ((* A)))` applied to the expression `(A A A A A)` rescans the latter 5 times, implying quadratic complexity. (Our current implementation does not attempt regular expression reduction for efficient recognition.) With the addition of the permutation operator {}, we can force all permutations of certain patterns be tried in an unsuccessful match (e.g., `(({} (! A B C) (! A B C) (! A B C)))` applied to `(C B E))`, leading to exponential complexity. (Again, our current implementation does not attempt to optimize.) Also, allowance for repeated application of a set of rules to a tree, until no further applications are possible, leads to Turing equivalence. This of course is true even if only the 4 underscore-operators are allowed: We can simulate the successive transformations of the configurations of a Turing machine with string rewriting rules, which are easily expressed in terms of those operators and `/`. Additionally, pattern predicates and function application in the right-hand sides of rules are features present in TTT that are not included in the above formal models. In themselves (even without iterative rule application), these unrestricted predicates and functions lead to Turing equivalence.

Nondeterminism and noncommutativity: In general, given a set of transductions (or even a single transduction) and an input tree there may be several ways to apply the transductions, resulting in different trees. This phenomenon comes from three sources:

- Rule application order - transductions are not in general commutative.
- Bindings - a pattern may have many sets of consistent bindings to a tree (e.g., pattern `(_* _*1)` can be bound to the tree `(X Y Z)` in four distinct ways).
- Subtree search order - a single transduction may be applicable to a tree in multiple locations (e.g., `(/ _! X)` could replace any node of a tree, including the root, with a single symbol).

Therefore some trees may have many reduced forms with respect to a set of transductions (where by reduced we mean a tree to which no transductions are applicable) and even more reachable forms.

Our current implementation does not attempt to enumerate possible transductions. Rather, for a given tree and a list of transductions, each transduction (in the order given) is applied in top-down fashion at each feasible location (matching the lhs), always using the first binding that results from this depth-first, left-to-right (i.e., pre-order) search. Our assumption is that the typical user has a clear sense of the order in which transformations are to be performed, and is working with rules that do not interact in unexpected ways. For example, consider the cases of PP misattachment mentioned earlier. In most cases, such misattachments are disjoint (e.g., consider a caption reading "John and Mary in front and David and Sue in the back", where both PPs may well have been attached to the proper noun immediately to the left, instead of to the appropriate conjunction). It is also possible for one rule application to change the context of another, but this is not necessarily problematic. For instance, suppose that in the sentence "John drove the speaker to the airport in a hurry" the PP "to the airport" has been misattached to the NP for "the speaker" and that the PP "in a hurry" has been misattached to the NP for "the airport". Suppose further that we have a repair rule that carries a PP attached to an NP upward in the parse tree until it reaches a VP node, reattaching the PP as a child of that VP. (The repair rule might incorporate a computable predicate that detects a poor fit between an NP and a PP that modifies it.) Then the result will be the same regardless of the order in which the two repairs are carried out. The difference is just that with a preorder discipline, the second PP ("in a hurry") will move upward by one step less than if the order is reversed, because the first rule application will have shortened the path to the dominating VP by one step.

In future it may be worthwhile to implement exhaustive exploration of all possible matches and expression rewrites, as has been done in Mathematica. In general this would call for lazy computation, since the set of rewrites may be an infinite set.

4 Some linguistic examples

Parse Tree Refinement: First, here is a simple transduction to delete nil constituents (i.e., empty brackets), which sometimes occur in the Brown corpus:

```
(/ (_* () _*1) (_* _*1))
```

To distinguish between past and passive participles, we want to search for the verb *have*, and change the participle token correspondingly, as discussed earlier. The following two transductions are equivalent – the first is global and the second is an example of a local or on-the-fly transduction. For simplicity we consider only the *has* form of *have*. Observe the more concise form, and simpler variable specifications of the second transduction.

```
 (/
   (VP _* (VBZ HAS) _*1 (VBN _!)  _*2)
   (VP _* (VBZ HAS) _*1 (VBEN _!)  _*2))


(VP _* (VBZ HAS) _* ((/ VBN VBEN) _!)  _*)
```

To distinguish temporal and non-temporal nominals, we use a computable predicate to detect temporal nouns, and then annotate the NP tag accordingly. (One more time, we show global and local variants.)

```
(/ (NP _* nn-temporal?)
    (NP-TIME _* nn-temporal?))


((/ NP NP-TIME) _* nn-temporal?)
```

Assimilation of verb particles into single constituents is useful to semantic interpretation, and is accomplished with the transduction:

```
(/ (VP (VB _!1) ( (PRT (RP _!2)) (NP _*1)))
   (VP (VB _!1 _!2) (NP _*1)))
```

We often particularize PPs to show the preposition involved, e.g., PP-OF, PP-FROM, etc. Note that this transduction uses the `join-with-dash!` function, which enables us to avoid writing a separate transduction for each preposition:

```
(/ (PP (IN _!)  _*1)
    ((join-with-dash!  PP _!)
      (IN _!)  _*1))
```

– such a rule transforms subtrees such as `(PP (IN FROM))` by annotating the PP tag as `(PP-FROM (IN FROM)`.

As a final syntactic processing example (transitioning to discourse phenomena and semantics), we illustrate the use of TTT in establishing potential coreferents licensed by C-command relations, for the sentence mentioned earlier. We assume that for reference purposes, NP nodes are decorated with a SEM-INDEX feature (with an integer value), and pronominal NPs are in addition decorated with a CANDIDATE-COREF feature, whose value is a list of indices (initially empty). Thus we have the following parse structure for the sentence at issue (where for understandabilty of the relatively complex parse tree we depart from Treebank conventions not only in the use of some explicit features but also in using linguistically more conventional phrasal and part-of-speech category names; R stands for relative clause):

```
(S ((NP SEM-INDEX 1) (NAME John))
   (VP (V shows)
       ((NP SEM-INDEX 2) (NAME Lillian))
       ((NP SEM-INDEX 3) (DET the)
        (N (N snowman)
           (R (RELPRON that)
              ((S GAP NP)
               ((NP SEM-INDEX 4
                 CANDIDATE-COREF ())
                (PRON he))
               ((VP GAP NP) (V built)
                ((NP SEM-INDEX 4)
                 (PRON *trace*)))))))))
```

Here is a TTT rule that adjoins the index of a C-commanding NP node to the CANDIDATE-COREF list of a C-commanded pronominal NP:

```
(_*
 ((NP _* SEM-INDEX _!.  _*) _+)
 _*
 (^* ((NP _* CANDIDATE-COREF
             (/ _!  (adjoin!  _!.  _!))
             _*)
       (PRON _!)))
 _*)
```

The NP on the second line is the C-commanding NP, and note that we are using a sticky variable '`_!.`' for its index, since we need to use it later. (None of the other match variables need to be sticky, and we reuse '`_*`' and '`_!`' multiple times.) The key to understanding the rule is the constituent headed by '`^*`', which triggers a search for a (right) sibling or descendant of a sibling of the NP node that reaches an NP consisting of a pronoun, and thus bearing the CANDIDATE-COREF feature. This feature is replaced "on the fly" by adjoining the index of the C-commanding node (the value of '`_!.`') to it. For the sample tree, the result is the following (note the value '`(1)`' of the CANDIDATE-COREF list):

```
(S ((NP SEM-INDEX 1) (NAME John))
   (VP (V shows)
       ((NP SEM-INDEX 2) (NAME Lillian))
       ((NP SEM-INDEX 3) (DET the)
        (N (N snowman)
           (R (RELPRON that)
              ((S GAP NP)
               ((NP SEM-INDEX 4
    CANDIDATE-COREF (1))
   (PRON he))
  ((VP GAP NP) (V built)
   ((NP SEM-INDEX 4)
    (PRON *trace*)))))))))
```

Of course, this does not yet incorporate number and gender checks, but while these could be included, it is preferable to gather candidates and heuristically pare them down later. Thus repeated application of the rule would also add the index 2 (for *Lillian*) to CANDIDATE-COREF.

Working with Logical Forms

Skolemization:

We wrote the function `subst-new!` to replace all occurrences of a free variable symbol in an expression with a new one, consistently labeled as such. (We assume that no variable occurs both bound and free in the same expression.) It uses a TTT transduction to accomplish this. Skolemization of an existential formula of type `(some x R S)`, where `x` is a variable, `R` is a restrictor formula and `S` is the nucleat scope, is then performed via the transduction

```
(/ (some _!  _!1 _!2)
   (subst-new!
            _!
            (_!1 and.cc _!2))).
```

For example, `(some x (x politician.n) (x honest.a))` becomes `((C1.skol politician.n) and.cc (C1.skol honest.a))`.

Inference: We can use the following rule to accomplish simple default inferences such as that if most things with property `P` have property `Q`, and most things with property `Q` have property `R`, then (in the absence of knowledge to the contrary) many things with property `P` also have property `R`. (Our logical forms use infix syntax for predication, i.e., the predicate follows the "subject" argument. Predicates can be lambda abstracts, and the computable boolean function `pred?` checks for arbitrary predicative constructs.)

```
(/ (_*
    (most _!.1 (_!.1 (!.p pred?))
               (_!.1 (!.q pred?)))
    _*
    (most _!.2 (_!.2 !.q)
               (_!.2 (!.r pred?)))
    _*)
   (many _!.1 (_!.1 !.p)
              (_!.1 !.r)))
```

For example, `((most x (x dog.n) (x pet.n)) (most y (y pet.n) (x friendly.a)))` yields the default inference `(many (x dog.n) (x friendly.a))`.

The assumption here is that the two *most*-formulas are embedded in a list of formulas (selected by the inference algorithm), and the three occurrences of `_*` allow for miscellaneous surrounding formulas. (To allow for arbitrary ordering of formulas in the working set, we also provide a variant with the two *most*-formulas in reverse order.)

Predicate Disambiguation: The following rules are applicable to patterns of predication such as `((det dog.n have.v (det tail.n))`, `((det bird.n have.v (det nest.n))`, and `((det man.n) have.v (det accident.n))`. (Think of `det` as an unspecified, unscoped quantifier.) The rules simultaneously introduce plausible patterns of quantification and plausible disambiguations of the various senses of `have.v` (e.g., have as part, possess, eat, experience):

```
(/ ((det (! animal?)) have.v
    (det (!1 animal-part?)))
   (all-or-most x (x !)
    (some e ((pair x e) enduring)
     (some y (y !1)
      ((x have-as-part.v y) ** e)))))

(/ ((det (! agent?)) have.v
    (det (!1 possession?)))
   (many x (x !)
    (some e
     (some y (y !1)
      (x possess.v y) ** e))))

(/ ((det (! animal?)) have.v
    (det (!1 food?)))
   (many x (x !)
    (occasional e
     (some y (y !1)
      (x eat.v y) ** e))))

(/ ((det (! person?)) have.v
    (det (!1 event?)))
   (many x (x !)
    (occasional e
     (some y (y !1)
      ((x experience.v y) ** e)))))
```

Computable predicates such as `animal?` and `event?` are evaluated with the help of WordNet and other resources. Details of the logical [form] need not concern us, but it should be noted that the '`**`' connects sentences to events they characterize much as in various other theories of events and situations.

Thus, for example, `((det dog.n have.v (det tail.n))` is mapped to:

```
(all-or-most x (x dog.n
 (some e ((pair x e) enduring)
  (some y (y tail.n)
   ((x have-as-part.v y) ** e)))))
```

This expresses that for all or most dogs, the dog has an enduring attribute (formalized as an agent-event pair) of having a tail as a part.

Logical Interpretation:

The following transductions directly map some simple parse trees to logical forms. The rules, applied as often as possible to a parse tree, replace all syntactic constructs, recognizable from (Treebank-style) phrase headers like `(NN ...)`, `(NNP ...)`, `(JJ ...)`, `(NP ...)`, `(VBD ...)`, `(VP ...)`, `(S ...)`, etc., by corresponding semantic constructs. For example, "The dog bit John Doe", parsed as

```
(S (NP (DT the) (NN dog))
   (VP (VBD bit)
       (NP (NNP John) (NNP Doe))))
```

yields `(the x (x dog.n) (x bit.v John_Doe.name))`.
Type-extensions such as '`.a`', '`.n`', and '`.v`' indicate adjectival, nominal, and verbal predicates, and the extension '`.name`' indicates an individual constant (name); these are added by the functions `make-adj!`, `make-noun!`, and so on. The fourth rule below combines two successive proper nouns (NNPs) into one. We omit event variables, tense and other refinements.

```
(/ (JJ _!)  (make-adj!  _!))
(/ (NN _!)  (make-noun!  _!))
(/ (VBD _!)  (make-verb!  _!))
(/ (_*.a (NNP _!.1) (NNP _!.2) _*.b)
     (_*.a (NNP _!.1 _!.2) _*.b))
(/ (NNP _+) (make-name!  (_+)))
(/ (NP _!)  _!)
(/ (S (NP (DT the) _!)  (VP _+))
(the x (x _!)  (x _+))
```

These rules are illustrative only, and are not fully compositional, as they interpret an NP with a determiner only in the context of a sentential subject, and a VP only in the context of a sentential predicate. Also, by scoping the variable of quantification, they do too much work at once. A more general approach would use compositional rules such as `(/ (S (!1 NP?) (!2 VP?)) ((sem! !1) (sem! !2)))`, where the `sem!` function again makes use of TTT, recursively unwinding the semantics, with rules like the first five above providing lexical-level `sem!`-values.

We have also experimented with rendering logical forms back into English, which is rather easier, mainly requiring dropping of variables and brackets and some reshuffling of constituents.

5 Conclusion

The TTT language is well-suited to the applications it was aimed at, and is already proving useful in current syntactic/semantic processing applications. It provides a very concise, transparent way of specifying transformations that previously required extensive symbolic processing. Some remaining issues are efficient access to, and deployment of, rules that are locally relevant to a transduction; and heuristics for executing matches and transductions more efficiently (e.g., recognizing various cases where a complex rule cannot possibly match a given tree, because the tree lacks some constituents called for by the rule; or use of efficient methods for matching regular-expression subpatterns).

The language also holds promise for rule-learning, thanks to its simple template-to-template basic syntax. The kinds of learning envisioned are learning parse-tree repair rules, and perhaps also LF repair rules and LF-to-English rules.

Acknowledgments

The work was supported by NSF and ONR grants and contracts. (Details are omitted for anonymity.)

References

Eugene Charniak and Mark Johnson. 2005. Coarse-to-Fine n-Best Parsing and MaxEnt Discriminative Reranking. *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics* (ACL'05), 173–180. Association for Computational Linguistics, Ann Arbor, MI, USA.

David Chiang. 2004. Evaluation of Grammar Formalisms for Applications to Natural Language Processing and Biological Sequence Analysis. Phd. Thesis. University of Pennsylvania.

Michel Galley and Mark Hopkins and Kevin Knight and Daniel Marcu 2004. What's in a Translation Rule?. *Proceedings of the 2004 Meeting of* *the North American chapter of the Association for Computational Linguistics* (NAACL '04), 273–280. Boston, MA, USA.

Ralph Griswold 1971. *The SNOBOL4 programming languge*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

Paul Hudak, John Peterson, and Joseph Fasel. 2000. A Gentle Introduction To Haskell: Version 98. Los Alamos National Laboratory. http://www.haskell.org/tutorial/patterns.html.

Kevin Knight 2007. Capturing practical natural language transformations. *Machine Translation*, Vol 21, Issue 2, 121–133. Kluwer Academic Publishers. Hingham, MA, USA.

Roger Levy and Galen Andrew. 2006. Tregex and Tsurgeon: tools for querying and manipulating tree data structures. *Language Resources Evaluation Conference* (LREC '06).

Andreas Maletti. 2010. Why synchronous tree substitution grammars?. *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics* (HLT '10). Association for Computational Linguistics, Stroudsburg, PA, USA.

Peter Norvig 1991. *Paradigms of Artificial Intelligence Programming* Morgan Kaufmann. Waltham, MA, USA.

Don Rozenberg 2002. SnoPy - Snobol Pattern Matching Extension for Python. http://snopy.sourceforge.net/user-guide.html.

Wolfram Research, Inc. 2010. *Wolfram Mathematica 8 Documentation*. Champagne, IL, USA. http://reference.wolfram.com/mathematica/guide/RulesAndPatterns.html.

Kenji Yamada and Kevin Knight 2001. A Syntax-Based Statistical Translation Model. *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics* (ACL '01), 523–530. Stroudsburg, PA, USA.

Min Zhang and Hongfei Jiang and Aiti Aw and Haizhou Li and Chew Lim Tan and Sheng Li 2008. A tree sequence alignment-based tree-to-tree translation model. *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics* (ACL '08).