

TTT - the Tree to Tree Transduction language

Adam Purtee Lenhart Schubert

Department of Computer Science
University of Rochester
Rochester, NY, USA
{apurtee,schubert}@cs.rochester.edu

April 24, 2011

Motivation

- Pattern matching and pattern-driven transformations of list-structured symbolic expressions or trees are fundamental tools in AI.
- Existing systems, such as Tiburon, Tsurgeon, and XSLT and models are efficient, but restricted in the types of transformations they allow. They are also not as immediately useful to our language processing tasks as we would like.
- 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.

Overview

TTT is composed of two main components, one for matching tree patterns and one for transformations of trees, and allows concise and transparent specification of transformations supporting such tasks as:

- parse tree refinement and correction
- predicate disambiguation
- logical form refinement
- inference
- verbalization of logical forms into English

This presentation will cover the matcher, then the transducer, then a set of language processing applications.

Pattern Matching Features

- Patterns are recursive.
- Patterns are composed of literal tree structure and operators.
- any tree structure may be specified (not only those with atomic left-most children)
- Pattern variables may bind to sequences of nodes.
- sticky bindings
- constraints on allowable bindings
- iterative constraints
- arbitrary predicates (e.g. `balanced?` `pp?` `nn-human?`)
- matching may be deep (search for sequence of siblings) or shallow (only test the root)
- Ten pattern operators: `!`, `?`, `+`, `*`, `<>`, `{}`, `^n`, `^@`, `^*`, and `/`.

Pattern Operators

T^3 includes the ten operators:

- ! - exactly one
- ? - zero or one
- + - one or more
- * - zero or more
- <> - ordered sequence
- {} - permuted sequence
- ^n - depth-n children
- ^* - descendant
- ^@ - vertical path
- / - local transduction

Example Patterns

- `(! (+ 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.

Example Patterns

Pattern	Tree	Bindings
<code>_!</code>	<code>(A B C)</code>	<code>(_! (A B C))</code>
<code>(A _! C)</code>	<code>(A B C)</code>	<code>(_! B)</code>
<code>(_* F)</code>	<code>(A B (C D E) F)</code>	<code>(_* A B (C D E))</code>
<code>(A B _? F)</code>	<code>(A B (C D E) F)</code>	<code>(_? (C D E))</code>
<code>(A B _? (C D E) F)</code>	<code>(A B (C D E) F)</code>	<code>(_?)</code>
<code>(^@ _! (C _*) E)</code>	<code>(A B (C D E) F)</code>	<code>(^@ (A B (C D E) F)) (_* D E)</code>
<code>(A B (<> (C D E)) F)</code>	<code>(A B (C D E) F)</code>	<code>(<> (C D E))</code>
<code>(A B (<> C D E) F)</code>	<code>(A B (C D E) F)</code>	<code>fail</code>

Features and Operator

Transduction features:

- The bindings created as a result of a successful pattern match may be used to construct new trees from a template.
- The resulting tree can be radically different (such as involving changes to the root) or simply be the result of modifications to a subtree of the original tree.
- One shot or until convergence.
- Local transductions allow large contexts.
- constructive functions with bound variables as arguments

Transductions are performed via the / operator:

- The / operator may appear anywhere within a pattern.
- Currently, at most one / operator per rule is supported.
- Can be at top level, e.g: (*/ source target*)
- Or nested, e.g: (*^@ .* (/ source target)*)

Brief Examples

- `(/ 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.
- `(/ (_* () _*1) (_* _*1))`
Deletes empty brackets (which sometimes occur in the Brown corpus)

Brief Examples

Source tree: (A B (X Y (Z X K)) T)

Transduction	Result
(/ X Y)	(A B (Y Y (Z Y K)) T)
(/ (A B _) (A _*))	(A (X Y (Z X K)) T)
(/ (!_ _) (_* _!))	(T A B (X Y (Z X K))) ... (A B (X Y (Z X K)) T) ...
(/ (-+ (Z _! K)) (_! _+))	(A B (X X Y) T)

Brief Examples

Repair of faulty classification of relative pronouns as wh-question pronouns:

Transduction:

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

Source tree:

```
(S (SBAR (WH X) B) A)
```

Result:

```
(S (SBAR (REL-WH (WH X)) B) A).
```

Time Complexity

Matching complexity depends on the operators involved, for example:

- $(! ((* A) B) ((* A) C) ((* A) D) ((* A) E) ((* A)))$ applied to the expression $(A A A A A)$ rescans the latter 5 times, implying quadratic complexity.
- With the addition of the permutation operator $\{\}$, we can force all permutations of certain patterns to 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.

Turing Equivalence

T^3 subsumes regular tree expressions:

- alternation can be expressed with !
- (vertical) iteration with \sim and $*$.
- The example expression from *TATA* can be specified as $(\sim (* (\text{cons } 0 \ @)) \text{nil})$, which matches Lisp expressions corresponding to lists of zero or more zeros.

T^3 is Turing equivalent:

- via repeated application of a set of rules to a tree (even with only the 4 underscore operators)
- pattern predicates and function application in the right-hand sides of rules are unrestricted

Nondeterminism

Nondeterminism arises in two ways:

- 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, and even more reachable forms. Our current approach is to apply each rule to subtrees in top-down, left-to-right (pre-order), until convergence.

Exhaustive exploration (via infinite streams) will be added in the future.

Parse Tree Refinement

Distinguishing between past and passive participles:

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

Using a local transduction:

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

Distinguishing temporal and non-temporal nominals:

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

Same rule, but as a local transduction:

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

Assimilation of verb particles into single constituents:

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

Particularization of prepositions:

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

Coreferent Determination

This transduction populates lists of candidate coreferents:

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

Source tree:

```
(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*)))))
                    ))
                ))
            ))
        ))
```

Output tree:

```
(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*)))))
                    ))
                ))
            ))
        ))
```


Skolemization

Skolemization of an existential formula of type $(\text{some } x \text{ R } S)$, is 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))
```

Predicate Disambiguation

The following rules disambiguate among various senses of have (e.g. have as part, as possession, as experience):

<pre>(/ ((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)))))</pre>	<pre>(/ ((det (! animal?)) have.v (det (!1 food?))) (many x (x !)) (occasional e (some y (y !1) (x eat.v y) ** e)))))</pre>
<pre>(/ ((det (! agent?)) have.v (det (!1 possession?))) (many x (x !)) (some e (some y (y !1) (x possess.v y) ** e)))))</pre>	<pre>(/ ((det (! person?)) have.v (det (!1 event?))) (many x (x !)) (occasional e (some y (y !1) ((x experience.v y) ** e)))))</pre>

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)))))
```

Logical Interpretation

The following transductions directly map some simple parse trees to logical forms.

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

For example, the input parse tree:

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

yields the logical form,

```
(the x (x dog.n) (x bit.v John\_Doe.name))
```

Conclusions and Future Work

The TTT language

- is well suited to the applications for which it is intended to be used,
- is already proving useful in current syntactic/semantic applications,
- and provides a very concise, transparent way of specifying transformations

Some remaining issues and future work are:

- efficient access to, and deployment of, rules that are locally relevant to a transduction;
- and heuristics for executing matches and transductions more efficiently
- investigate learning of parse repair, LF repair, and LF-to-English rules.

Acknowledgments

The work was supported by ONR-STTR award N00014-11-10417, and NSF grants IIS-1016735, NSF IIS-0916599, and NSF IIS-0910611.

Thanks to Dan Gildea for his advice throughout the project. Also thanks to the anonymous reviewers for their helpful comments on the first submission of the paper.