trx: Regular-tree expressions, now in Scheme

Ilya Bagrak
University of California, Berkeley
ibagrak@eecs.berkeley.edu

Olin Shivers
College of Computing
Georgia Institute of Technology
shivers@cc.gatech.edu

Abstract

Regular-tree expressions are to semi-structured data, such as XML and Lisp s-expressions, what standard regular expressions are to strings: a powerful "chainsaw" for describing, searching and transforming structure in large data sets. We have designed and implemented a little language, *trx*, for defining regular-tree patterns. We discuss the design of *trx*, its underlying mathematical formalisation with various kinds of tree automata, and its implementation technology. One of the attractions of *trx* is that, rather than being a complete, *ad hoc* language for computing with trees, it is instead embedded within Scheme by means of the Scheme macro system. The features of the design are demonstrated with multiple motivating examples. The resulting system is of general use to programmers who wish to operate on tree-structured data in Scheme.

1 LCD data representations

Semi-structured and tree-structured data has become an important topic in the world of software engineering in the past few years, due to the widespread adoption of XML as a generic representation format for data. While this may be news to rest of the world, it is a very familiar picture to programmers in the Lisp family of languages. The Scheme and Lisp community has long been aware of the benefits of fixing on a general-purpose data structure for representing trees, and specifying a standard concrete character representation for these trees. Lisp s-expressions are essentially XML trees; the Lisp community has worked within the s-expression framework for representing data since the inception of the language in the 1960's.

Part of the power of the Lisp family of languages comes from this focus on s-expressions as the central data structure of the language. A Perlis aphorism [16] captures the benefit well: "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures." S-expressions are the "least common denominator" (LCD) representation for data in the Lisp family of programming languages, in the same sense that strings are the LCD representation in the world of Unix tools: the multitude of functions that operate upon and produce results in this form can therefore easily be connected together to construct larger computations, providing for a large degree of code reuse. In the world of Scheme programming, s-expressions are the universal interchange format.

The charm of s-expressions as an LCD representation is that, unlike strings, they come with some degree of existing structure. This eliminates much of the parsing/unparsing overhead that is typically required when computational agents interact using string intermediate representations (parsing that, in the Unix-tools setting, is frequently done by means of heuristic, incomplete, error-prone handwritten parsers). Another Perlis aphorism makes clear the downsides of using strings as an LCD form: "The string is a stark data structure and everywhere it is passed there is much duplication of process. It is a perfect vehicle for hiding information." The problem with strings as a least-common denominator is that they are too "least," that is, too low level. We operate upon strings a character at a time—a level where it is all too easy to break the invariants of the associated grammar that typically imposes structure and meaning on the strings.

Even when s-expressions may not be the appropriate representation for the core data structures of an application, they still frequently find use around the application's "fringe," being converted to and from the internal, more highly-engineered core structures as they move across the application's boundary—with the associated benefit that it is *much* simpler and more robust to parse a tree than a string.

2 Regular trees, little languages and Scheme

Given that Lisp and Scheme programmers have been working with "semi-structured" tree data roughly three and half decades longer than XML has even existed, it is surprising that this community has never bothered to adopt one of the great, expressive tools for manipulating such data: regular trees and their associated patterns. Just as traditional regular expressions are an expressive tool for describing structure occurring within strings, regular trees can serve a similar role when dealing with recursively defined patterns occurring within trees—trees such as ones we frequently represent using Scheme s-expressions.

We have long grumbled about the lack of such tools. Each time we write a low-level Scheme macro, for example, and we find ourselves writing an incomplete and awkward syntax-checker/parser for our new form directly in Scheme (Is the form exactly four elements long? Is the second element a list of identifier/expression pairs? *Etc.*), we pause to wish for a better way. When the XML world began wisely to exploit the extensive theoretical machinery

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

developed to describe regular trees and their recognisers, we were finally pushed to carry out the design and implementation exercise we had put off so long.

The result is *trx*, a language for describing regular-tree patterns. Embedding our "little language" within Scheme provides for several benefits, which we've described elsewhere in detail [17]. On one hand, it made our task as designers and implementors easier. We only needed to design and implement tree patterns; we didn't need to implement the machinery already provided by Scheme: floating-point numbers, first-class functions, variables, loops, *etc.* On the other hand, for our programmer clients, the result was a tool that allowed regular-tree pattern matching to be tightly integrated with Scheme programs, instead of forcing this kind of operation out into a separate, distinct program written in some distinct, *ad hoc*, self-contained domain-specific language.

The rest of this paper traces out the following arc. First, we will survey the basic elements of tree automata, the underlying mathematical formalism that connects the static, declarative world of regular-tree patterns to the computational or algorithmic paradigm of their recognisers. Then we will consider the particular needs of tree pattern matching that arise when working in the setting of Scheme s-expressions. This exploration of design requirements and design rationale, plus the useful constraints imposed by the computational power of tree automata, allow us to proceed to a design for regular-tree patterns that integrates with Scheme s-expressions. The syntax and semantics of *trx* are provided in the next section, followed by examples of *trx* patterns in use. Then we will examine some details the current implementation, before concluding with a description of related and future work.

3 Overview of tree automata

Every interesting programming language is just a cover for an interesting model of computation: regular expressions and finite automata; context-free grammars and push-down automata; SQL and the relational calculus; Smalltalk and message-passing; APL and SIMD array processing; and, of course, Scheme and the λ calculus. The interesting formal model of computation underlying the design of *trx* is *finite tree automata*. The short overview that follows will spell out some of the fundamental concepts of these formal machines.

In the following sections we differentiate between *traditional* tree automata and *simplified* tree automata. This paper uses the "simplified" and "traditional" qualifiers for differentiation only; they are not part of the established nomenclature. Elsewhere in the literature, both classes of automata are referred to as tree automata interchangeably.

3.1 Traditional tree automata

Tree automata operate on labelled, finite trees: trees where every node is assigned a label f drawn from some alphabet $\mathcal{F}$. Traditional automata also require the label alphabet to be *ranked*, that is, each label has an associated natural number. Each tree node must have exactly as many children as the rank assigned its label; thus, leaf nodes are marked with rank-zero labels. We write $\mathcal{F}_n$ for the set of rank-n labels in alphabet $\mathcal{F}$.

A **traditional finite tree automaton** (FTA) over a ranked alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, where Q is a set of states, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transition rules of the form:

$$f(q_1, \ldots, q_n) \to q,$$

where $n \geq 0$, $f \in \mathcal{F}_n$, and $q, q_1, \ldots, q_n \in Q$. The symbols $q_1, \ldots, q_n$ and q are called the *initial* and *final* states of the transition, respectively.

The operation of a tree automaton involves propagating state information up (or down) through the tree. Transition rules determine how this propagation takes place. Whenever a label f is seen at a tree node and that label has the states $q_1, \ldots, q_n$ "bubbled-up" to its children, and a rule $f(q_1, \ldots, q_n) \to q$ exists in Δ, state q is propagated to the f-labelled node. In turn, the bubbled-up state then feeds into its parent node. The propagation continues until some state is bubbled up to the root node of the tree. If this state is in Q_f, then the tree term is accepted. If no final state bubbles up to the top, the tree is rejected.

In addition to the type of transition rule described above, traditional tree automata allow for ε-move transitions $q \to q'$, that occur spontaneously, changing the state assigned to a node from q to q'. Equivalence of tree automata with and without ε rules is a well-established result [6]; establishing the equivalence involves working with the ε-closure of states, *i.e.*, the set of states reachable from a state via ε-rules.

Fundamentally, every tree automaton $\mathcal{A}$ is a machine corresponding to some *tree language*. The tree language $\mathcal{L}(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of all trees accepted by $\mathcal{A}$.

We've described the operation of an FTA in a bottom-up manner, but it can also be operated in a top-down manner, starting with an accept state for the root, and running the transition rules "backwards" to find the labels assigned to children, *etc.*

3.2 Simplified tree automata

A **simplified finite tree automaton** (SFTA) over an unranked alphabet $\mathcal{F}$ is a tuple $\mathcal{A}_s = (Q, \mathcal{F}, Q_i, \Delta)$, where Q is a set of states, $Q_i \subseteq Q$ is a set of initial states, and Δ is a set of transition rules. Transition rules can be either be *labelled* or *empty*:

$$f(q_{\text{in}}), q_{\text{out}} \to q \text{ or } () \to q.$$

In order to understand the way a simplified tree automaton computes, we must change our mental model of how individual nodes are "wired" together in the tree. Each node now includes a reference to its closest sibling to the right, and its leftmost child (if any, in both cases). This setup implies that at any given point in an automaton's operation, state-directed control can flow along two pathways—down to children and right to siblings. This is in contrast to traditional tree automata where state information is propagated to/from *all* children simultaneously.

A simplified automaton begins at the root of the tree, nondeterministically selecting a start state from Q_i. If the automaton is visiting an f-labelled node n while in state q, the machine selects an $f(q_{\text{in}}), q_{\text{out}} \to q$ transition. (If there is no such transition, the machine halts, reporting failure.) If n has children, the machine attempts to recursively accept them, starting in state q_{in} with n's leftmost child; if this succeeds, it then proceeds to n's siblings. If n is a leaf node, the machine checks for an empty transition $() \to q_{\text{out}}$, then proceeds to n's siblings. If the children-match attempt fails, or there is no empty rule handling the leaf node, the machine halts, reporting failure.

Figure 1
Nondeterministic simplified finite tree automaton

```
matchnode(n,q) {
  f := n.label

  /* Fail if no rule selectable. */
  Select f(q_in),q_out → q from Δ

  if n is leaf
  then matchempty(q_in)
  else matchnode(n.leftchild, q_in)

  if n has closest right sibling s
  then matchnode(s,q_out)
  else matchempty(q_out)
  }

matchempty(q) {
 if () → q ∈ Δ then return
 else fail
 }
```

To proceed to n's siblings, the machine jumps to n's closest right sibling and changes state to q_{out}. If n has no right sibling, the machine accepts iff there is an empty transition $q_{\text{out}} \to ()$. Thus empty transition rules are needed to terminate an automaton's recursive descent over a tree. Pseudocode for an SFTA is shown in figure 1.

As a trivial example, consider a regular-tree language consisting of a single term: a root node labelled with a and three child nodes labelled with b, c, d. A simplified tree automaton for recognizing such a language would have transitions

$$\begin{array}{ll} a(q_1), q_2 \to q_0 & b(q_2), q_4 \to q_1 \\ c(q_2), q_6 \to q_4 & d(q_2), q_2 \to q_6 \\ () \to q_2 & \end{array}$$

with $Q_i = \{q_0\}$.

Note that the label alphabet $\mathcal{F}$ is unranked in the sense that when a node labelled f is processed, the automaton is only concerned with the presence of the node's leftmost child and closest right sibling. Nothing in the rule format enforces how many children or siblings a given node is allowed to have. A simplified automaton can fix the number of children permitted a tree node by encoding this in the states traversed as it scans across the siblings, but it may also permit a child to have an arbitrary number of children, a degree of power not available with traditional automata. Thus simplified automata are strictly more powerful than traditional automata. This power is useful for the kinds of s-expression and XML trees we process in the real world.

3.3 Converting between models

A traditional tree automaton can be converted to an equivalent simplified tree automaton in the following way. Starting from each state q of the initial states, the algorithm selects all the rules that have q as their final state. For each rule $f(q_1,\ldots,q_n) \to q$, a labelled rule $f(q_1), q' \to q$ and an empty rule $() \to q'$ are added to the simplified automaton, for fresh state q'. Then a new state q'' and empty rule $() \to q''$ are added to the automaton, for fresh state q''. The children states are processed similarly, except that instead of generating a fresh "out" state as we did for the rules starting in q, the rules starting in q_n are processed recursively with their "out" state as their right sibling's final state q_{n+1}. The "out" state for the rightmost sibling is the newly-generated q''.

A simplified tree automaton resulting from the conversion recognizes the same language as its traditional equivalent. However, the arity information is now encoded directly in the rules; symbols no longer have an intrinsic arity attached to them.

A conversion of a simplified tree automaton to an equivalent traditional tree automaton is not possible in the general case: as we've seen, traditional tree automata cannot handle symbols with unbounded arity, and thus are strictly less powerful.

3.4 Nondeterminism in tree automata

Both of the above definitions describe non-deterministic automata, *i.e.,* automata that can "fork" in a number of directions if multiple transitions can be applied in a given machine configuration. As with regular-string languages, a deterministic variant of tree automata can be defined as a subset of the general nondeterministic one.

The equivalence of deterministic and non-deterministic automata has been established for traditional tree automata [6]. A traditional tree automaton is said to be **deterministic** (DFTA) if there are no rules with the same left-hand side, and no ε-rules.

The construction of an equivalent automaton proceeds as follows. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a non-deterministic tree automaton. Then there is an equivalent DFTA, $\mathcal{A}_d = (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$ such that (1) every state in Q_d is a non-empty set of original states in Q, and (2) every rule in Δ_d is computed with

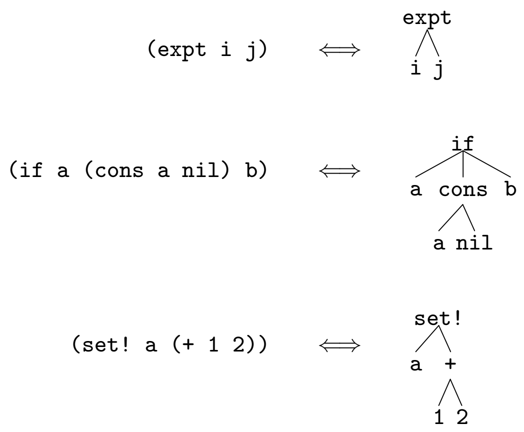
$$\begin{array}{l} f(s_1,\ldots,s_n) \to s' \in \Delta_d \text{ iff} \\ s = \{q \mid q_1 \in s_1,\ldots,q_n \in s_n, f(q_1,\ldots,q_n) \to q \in \Delta\} \\ s' = \varepsilon\text{-closure}(s) \end{array}$$

Just as with regular-string expressions and their associated finite-state automata, this power-set construction of an equivalent deterministic tree automaton may, in the worst case, result in exponential state explosion [6]. And just as with regular-string expressions, this negative effect is offset by the fact that deterministic tree automata are considerably faster at parsing certain regular-tree languages [6], in the standard space *vs.* search trade-off.

We are not aware of any results pertaining to equivalence of deterministic and non-deterministic simplified tree automata. Our SFTA technology works strictly with non-deterministic machines—that is, it manages non-determinacy at run time by performing backtracking search.

4 S-expressions, XML, and regular trees

Lisp s-expressions are frequently used to represent labelled trees, using the encoding that an internal tree node is represented by a list whose head is its label and whose tail is the list of its children; leaf nodes are simply represented by non-list data. The following schematics illustrate the mapping:



In the domain of XML, the delimiting characters (and) are replaced with <*tag attr-list*> and </*tag*>, with *tag* serving as the node's label. This treatment slightly oversimplifies the way XML documents are put together, but it exposes the features common to labelled-tree languages at large, which is what the *trx* language is intended to process. The XML community has developed a plethora of language tools for describing regular-tree patterns as well as transducers operating on the trees matched by these patterns [12, 4, 6, 21, 3]. Although these tools are outside the scope of this paper, their existence reaffirms the utility of convenient, robust tools for regular-tree processing.

4.1 Variable-arity constructors

One issue that arises when considering tree structure in XML and symbol-labelled s-expression trees is the possibility of variable-arity constructors (that is, variable-rank tree labels). Both `(+ 3 7)` and `(+ 2 6 3 1)` are legal Scheme expressions, yet the `+` label must be assigned a fixed arity in order for a traditional tree-automata to be able to match it. These cases arise in both XML and s-expression trees, and when they do, we must resort to the more powerful model of simplified tree automata.

4.2 Unlabelled tree nodes

Another issue we find in the context of interpreting s-expressions as labelled trees is that in some s-expressions, not all tree nodes are labelled. For example, consider the structure of a Scheme `let` form, which has the following syntax:

`(let ((`*var exp*`) ...)` *body* `...)`

The first child of a `let` node, the bindings list `((`*var exp*`) ...)`, has no label. (As we discussed above, it is also variable-arity.) We can find the same problem in the syntax of the individual clauses of a `cond` expression, or even in the "default" syntax of Scheme function calls, which are not introduced by any kind of `call` keyword.

It is not technically difficult to handle unlabelled nodes within the finite-automaton model; our implementation does so by introducing special anonymous symbols that have unique names with respect to the rest of the automaton's label set. The critical point is that, to handle this tree idiom, which occurs in common practice with s-expressions, we must account for them in the design of our pattern notation.

4.3 Factoring pattern, automaton and data

One theme in the design of *trx* is factoring the layers of the design. Whether the terms under consideration are s-expressions, XML documents, or some other form of tree data, the basic pattern notation and the underlying abstract automata models, which specify the basic processing engine for tree terms, should remain unaltered. We'll return to this factoring in the discussion of the implementation.

Another design concern was abstracting over the nuts and bolts of finite-tree automata or other possible semantic engines for *trx*. (*E.g.*, is a pattern implemented as a non-deterministic traditional FTA using search, as a deterministic traditional FTA without backtracking, or with an SFTA?) Where possible, we kept the notation and its semantics independent of these implementation pragmatics (adding variable-arity patterns in the pattern notation does restrict the space of possible implementations, however, so this is not 100% possible).

4.4 Escapes to Scheme

We've found it useful in previous little-language designs to provide mechanisms not only for embedding the little language within Scheme, but for embedding general Scheme within the little language. This, of course, dramatically changes the power of the pattern model, allowing us to define pattern matchers whose top-level control skeleton is an FTA, but who may invoke arbitrary computation at the "leaves" of the computation. Adding such a facility has an impact on the implementation of the system, restricting our ability to statically analyse patterns (a price we pay for the increased computational power), and requiring the implementation to be written so it can simply pass the embedded Scheme code through the pattern compiler, to be dropped into place in the final output.

As we'll see, the ability to invoke arbitrary Scheme at the leaves of the pattern matcher in particular allow us to have trees whose leaf nodes are not just symbols, but any kind of data. This is important in the world of Scheme s-expressions, which are frequently composed of more than symbols and parentheses—they may, for example, contain records or booleans or strings. We might, in some contexts, wish to permit only leaves that are positive integers between 0 and 100—something which does not fit the basic tree-automata model, which discriminates only on the symbols that label nodes, including leaf nodes.

Similarly, when a user writes down a regular-tree expression to describe the syntax of the Scheme `let` form, he will want to capture in the pattern the fact that the left-hand sides of the binding forms can be any symbol at all... but only a symbol, not a general subtree.[1] Allowing escapes to general Scheme code permits us to write

[1] If he wants to capture the constraint that these bound identifiers must be distinct from one another, he's completely outside the power of the regular-tree model. The price we pay for specialised notations and restricted computational models is that we can't solve all possible problems. Note that our hypothetical programmer could always use a more complex escape to Scheme to check this distinct-identifier constraint, or defer it to a later check. This is analogous to the way compilers detect some illegal programs while parsing (*i.e.*, syntax errors), leaving others for later static analysis to find (*e.g.*, type errors). This distinction happens for the same reason—the expressiveness of context-free grammars and the power of the associated push-down automata that recognise their languages are restricted, making them unable to encode all the static constraints

such patterns, yet remain in the tree-automaton/declarative-pattern model for the most part. This functionality is conceptually aligned with the Scheme's own type system where types are typically defined by means of general Scheme predicates which discriminate between members and non-members of the type, *e.g.*, functions such as `list?`, `string?`, `symbol?`, and so forth.

4.5 Dynamic patterns

Besides inlining Scheme predicates to match tree leaves, we might also want to escape to Scheme code within a pattern in order to *compute* a sub-pattern. This allows users to dynamically stitch tree automata together, or construct patterns which may have a run-time dependency on particular computations or input data.

4.6 Collecting submatch data

We frequently want our patterns to do more than simply recognise trees, reporting only a "yes" or a "no." In many cases, we want to use our patterns to *select* indicated parts of a tree. One mechanism for doing so is to add elements to the pattern language for marking components of a tree that match particular pieces of a pattern. On a successful match, the selected sub-trees are then returned to the programmer as a result. String regular expressions frequently have similar kinds of support for picking out elements of a matched string. Providing submatches in the pattern notation complicates the implementation of the pattern matcher; in particular, the pattern optimiser has to be careful not to optimize away subpatterns that contain a submatch.

5 The *trx* language

The syntax of *trx* patterns takes the form of the familiar s-expression and borrows extensively from the *SRE* regular-expression notation introduced in *scsh* [19, 18]. The grammar is given in figure 2.

A regular-tree expression (or pattern) denotes a set of trees. A pattern which is simply a literal, such as the number 5, is a pattern matching only the leaf tree 5. Similarly, the pattern `'fred` (or, equivalently, `(quote fred)`) matches the leaf which is the symbol `fred`.

The pattern `(@` *symbol rte* `...)` matches a tree whose root is labelled with *symbol*, and whose children match the *rte* sub-patterns. When *symbol* doesn't conflict with one of the pattern keywords, the `@` can be elided. A tree with an unlabelled root can be matched with a `(^` *rte* `...)` pattern.

We introduce choice with the pattern `(|` *rte* `...)` which matches any tree matched by any of the *rte* subforms.

The pattern `(any)` matches any tree. We can write a match which matches no tree with the empty-choice pattern `(|)`. This is not particularly useful for user-written patterns, but could be useful for patterns produced mechanically, either from higher-level macros or dynamically in response to program input.

The sequence operators `*`, `+` and `?` match zero-or-more, one-or-more and zero-or-one trees matching their subpattern, respectively.

---

of a well-formed legal program. The role of a little language is to make the common cases easy; the role of a general purpose language (such as our escapes to Scheme) is to make the rest of the cases possible.

What makes regular-tree patterns interesting is recursion in the patterns. This is introduced with the `rec` and `letrec` forms. The pattern `(rec` *ident rte*`)` matches a tree that matches *rte*, with the proviso that free references to *ident* in *rte* must recursively match the pattern, as well. Thus we can describe a pattern that matches binary trees whose internal nodes are labelled `+` and whose leaves are 42 with the pattern

```
(rec t (| 42 (@ + t t)))
```

This would match any of the trees `42`, `(+ 42 42)`, `(+ 42 (+ 42 42))`, `(+ (+ 42 42) 42)` and so forth.

The `letrec` form allows mutual recursion by binding the pattern identifiers in a recursive scope. We can also bind pattern identifiers with simple lexical scope with the `let` form.

The `(submatch` *rte*`)` form lets us mark a part of a larger pattern to indicate to the pattern matcher that, in the event of a complete match, the sub-trees matching *rte* should be retained for later retrieval. Note that a single submatch can match more than a single tree term. For instance, the patterns

```
(rec t (| 42 (@ + (submatch t) t)))
(@ + (* (submatch 42)))
```

would produce, upon a successful match, a variable number of submatches depending on the height and width of the tree term. The matcher produces a list of terms for any single `submatch` form, ordered according to the pre-order position of submatched terms in the original tree. Thus for pattern

```
(rec t (| ,number? (submatch (@ + t t))))
```

and tree term `(+ (+ 1 2) (+ 3 4))`, the list of saved items for the submatch will consist of every internal node in the source tree:

```
((+ (+ 1 2) (+ 3 4))
 (+ 1 2)
 (+ 3 4))
```

Finally, we can escape to general Scheme code in two different ways. The pattern `,`*exp* allows us to write a Scheme expression providing a general predicate which accepts or rejects trees. Thus we can change our sum-of-42s example above to be a sum tree for general numbers with the pattern

```
(rec t (| ,number? (@ + t t)))
```

or a sum tree of even numbers with the pattern

```
(rec t (| ,(λ (x) (and (number? x)
                       (even? x)))
          (@ + t t)))
```

The pattern `,@`*scheme-exp*, allows us to write a Scheme expression that itself evaluates to a *trx* pattern value, which is then plugged into the enclosing pattern. This allows us to dynamically construct *trx* patterns, instead of restricting them to patterns that are completely fixed at compile time. (Consequently, this feature has major implications on the compile-time handling of patterns—when it is used, we must do a kind of "partial evaluation" of the pattern, deferring the processing of dynamic components to run time. Fortunately, we *can* statically determine if a particular pattern uses this feature of the language, and so only need defer such processing with patterns that do so. So the extra overhead of dynamic pattern construction is only invoked as needed, making it a pay-as-you-go feature.)

Figure 2 Syntax of trx regular-tree expressions.

rte `::=` *literal* | `'`*symbol*	; Literal atom
| `(@` *symbol* *rte* `...)`	; Tree with root labelled *symbol*
| `(`*symbol* *rte* `...)`	; As for `@`, when no ambiguity.
| `(^` *rte* `...)`	; Tree with unlabelled root
| `(any)`	; Matches any tree
| `(|` *rte* `...)`	; Choice
| `(*` *rte*`)`	; Matches a sequence of $[0,\infty)$ *rte*'s
| `(+` *rte*`)`	; Matches a sequence of $[1,\infty)$ *rte*'s
| `(?` *rte*`)`	; Matches a sequence of $[0,1]$ *rte*'s
| `(rec` *ident* *rte*`)`	; Recursively defined pattern
| `(let ((`*ident* *rte*`) ...)` *rte*`)`	; Lexical pattern binding
| `(let* ((`*ident* *rte*`) ...)` *rte*`)`	; Lexical pattern binding
| `(letrec ((`*ident* *rte*`) ...)` *rte*`)`	; Pattern with mutual recursion
| *ident*	; Reference to pattern bound by `rec`, `let` or `letrec`
| `(submatch` *rte*`)`	; Matched subtree saved for subsequent retrieval
| `,`*scheme-exp*	; General predicate
| `,@`*scheme-exp*	; Dynamically computed tree automaton
ident `::=` *symbol*	
literal `::=` *number* | *string* | *boolean* | *char*	

As an example putting multiple components of the language together, a pattern which specifies the syntax of the Scheme `let` expression is

```
(@ let (^ (* (^ ,symbol? (any))))
   (+ (any)))
```

or, with components of the pattern let-bound for clarity,

```
(let* ((binding  (^ ,symbol? (any)))
       (bindings (^ (* binding)))
       (body     (+ (any))))
  (@ let bindings body))
```

Notice how the unlabelled-tree patterns are used to match each (*var exp*) binding form as well as the list of these bindings. As an exercise, you may wish to extend the pattern to handle named-let forms used for iteration.

6 Static semantics

Our informal description of the *trx* language has glossed over a distinction between the language's various constructs. While an operator such as `@` produces a pattern that matches a tree, the `*`, `+` and `?` operators produce patterns that match a *sequence* of trees. These sequence or "forest" patterns can appear anywhere in the language a tree pattern can appear, with the restriction that a complete, top-level pattern cannot be a sequence pattern. We cannot encode this directly in the grammar due to the presence of the pattern-binding forms (`let`, `letrec` and `rec`)—there's no way to design the grammar to guarantee that a particular identifier reference is made to a tree-pattern binding and not a forest-pattern binding. This is the sort of restriction that one typically manages in the post-parse static-semantics phase of a compiler, in a type-system-like manner. This is exactly what we do. The macros that process tree patterns check them to ensure that the top-level pattern has a "tree-pattern" type. Similarly, because identifier references are resolved lexically, references to unbound identifiers are checked for and rejected at macro-expansion time.

7 Examples

At last, we present a set of examples which work to illustrate the capabilities of the *trx* language. We embed patterns into Scheme code by means of the Scheme form `(trx` *rte*`)`. This is a Scheme expression whose body *rte* is not Scheme code, but rather a *trx* regular-tree pattern. The `trx` form produces a tree-automaton value which can be passed to the `trx-match` pattern matcher. It is implemented as a macro that compiles its pattern body to a tree automaton, represented with an abstract data type. More details of the implementation are given in the following section. The matcher function is invoked as `(trx-match` *pat* *s-exp*`)`, taking a tree automaton *pat* (produced by a `(trx` *rte*`)` form), and a tree *s-exp* to which it should be applied. It returns a non-false value for a successful match, and `#f` otherwise.

Example 1 We begin with a set of Scheme expressions that construct nested lists of numbers. The number leaves are matched using the Scheme `number?` procedure. The example demonstrates escaped Scheme code and the use of the `rec` operator.

```
(let ((p (trx (rec q (| (cons q q)
                        (cons ,number? q)
                        nil)))))
  (trx-match p '(cons 1 nil))       ; match
  (trx-match p '(cons nil nil))     ; match
  (trx-match p '(cons (cons a nil) ; fail
                      (cons 1 nil))))
```

Example 2 A somewhat more interesting use of escapes to Scheme code involves user input. The language is similar to the first example, except that numbers must be divisible by the user-specified divisor.

```
(let* ((i (read))
       (idiv? (λ (n) (= (modulo n i) 0)))
       (p (trx (rec q (| (cons q q)
                         (cons ,idiv? q)
                         nil)))))
  (trx-match p '(cons nil nil))      ; match
  (trx-match p '(cons (cons a nil)   ; fail
                      (cons 1 nil))))
```

Example 3 The purpose of this example is to illustrate the use of the `letrec` construct. The pattern below matches any Scheme expression that consists solely of applications of `+` and `*` such that `+` is never applied to a `+` expression, and *vice versa*. In other words, the constructors must alternate within the tree. This example also illustrates the use of variable-arity constructors and labels that collide with reserved keywords.

```
(let ((p (trx (letrec ((m (| n (@ * (* a))))
                       (a (| n (@ + (* m))))
                       (n ,number?))
                (| m a)))))
   (trx-match p '(* 2 (+ 3 4))) ; match
   (trx-match p '(* 2 (* 3 4)))); fail
```

Example 4 We now consider a pattern for recognizing XML-like data sets capturing data entered into an online purchase-order form.

```
(trx (order (date ,string?)
            (shipto (name ,string?)
                    (address ,string?))
            (? (comment ,string?))
            (+ (item (part ,number?)
                     (quantity ,number?)
                     (price ,number?)))))
```

The pattern will match

```
(order (date "2004-06-11")
       (shipto (name "Bill")
               (address "1 Main Street"))
       (comment "Please, hurry!")
       (item (part 111)
             (quantity 1)
             (price 1.00))
       (item (part 222)
             (quantity 2)
             (price 2.00)))
```

but not

```
(order (date "2004-06-11")
       (comment "Please, hurry!")
       (shipto (name "Bill")
               (address "1 Main Street"))
       (item (part 111)
             (quantity 1)
             (price 1.00))
       (item (part 222)
             (quantity 2)
             (price 2.00)))
```

or

```
(order (date "2004-06-11")
       (shipto (name "Bill")
               (address "1 Main Street"))
       (comment "Please, hurry!"))
```

Example 5 Building on the previous example, we illustrate how the `submatch` operator can be used to transduce tree terms. Specifically, we want to match a sequence of orders, but we also want to add a free gift from the company to every order in the sequence which includes two or more distinct items. We augment the previous definition of an order-matching pattern to match "sequence-of-orders" datasets. Figure 3 shows how we can match a suitable sequence of orders against this new pattern, alter submatched terms, and reconstitute them into a new list of orders.

Example 6 As a final example of *trx*, figure 4 shows the grammar of *trx*, expressed as a *trx* pattern. It's not an accident that this is so straightforward to encode: the long-standing convention s-expression language designers use for their syntax design[2] is *exactly* the labelled-tree model processed by tree automata. So this example gives away one of the elements of our development agenda. We are interested in the use of *trx* to provide more precise syntax specification and error-checking for the kinds of languages we like: the s-expression-based ones.

8 Implementation

We have implemented the *trx* system as a module in the scsh Scheme environment [19]. The code is fairly portable; its most significant element of non-portability is its use of a non-R5RS low-level macro system. Our implementation can be subdivided into following components:

- A macro embedding applications of the *trx* notation into Scheme forms. Basically, the macro simply interfaces the *trx* compiler to the underlying Scheme compiler.
- A set of Scheme data-type definitions encoding the abstract syntax of the *trx* language as well as the resulting automata values. This establishes a set of ADTs around which processing of regular-tree patterns takes place.
- A pair of Scheme procedures for parsing tree patterns from their s-expression concrete syntax into their representation using the internal AST structures; and for unparsing from an AST value back to its external, printable s-expression representation.
- A procedure that translates a *trx* AST into an automata value. This is the heart of the *trx* compiler.
- Pattern-matching procedures. One of these procedures is the pattern matcher; the other is a routine that extracts submatched terms from a result of a successful match.

8.1 The *trx* macro

As mentioned above, the *trx* compiler is invoked on every occurrence of the `trx` macro in Scheme source code. In our implementation, we utilize Scheme 48's low-level macro facility, the Clinger/Rees "explicit renaming" macros [5], which allows both full control of hygiene and permits macros to be written in general Scheme code. The latter feature is particularly important, as the *trx* machinery is fairly complex—at the complexity level of a small compiler, as opposed to the kind of simple pattern-directed extensions usually implemented via the R5RS high-level macro facility. The macro simply invokes the rest of the machinery, which parses the pattern into an AST, performs static-semantics checks, simplifies the pattern, compiles it to a value in the automata ADT, and finally renders the result automaton as a block of Scheme code.

[2] Barring certain regrettable exercises in syntactic excess, such as the Common Lisp `loop` form.

Figure 3 A simple tree transducer.

```
(let ((pat (trx (order (date ,string?)
                      (shipto (name ,string?)
                              (address ,string?))
                      (? (comment ,string?))
                      (submatch (+ (item (part ,number?)
                                         (quantity ,number?)
                                         (price ,number?))))))))
  (map (λ (order) (cond ((trx-match pat order) =>
                       (λ (match) (let ((items (trx-submatch match 1)))
                                    (if (>= (length items) 2)
                                        (cons '(item (part 001)
                                                     (quantity 1)
                                                     (price 0))
                                              items)
                                        items))))
                      (else (error "Illegal order"))))
       orders))
```

Figure 4 The *trx* grammar as a *trx* pattern.

```
(rec rte (| ,string? ,number? ,character?          ; Literals
            ,(λ (x) (or (not x) (eq? x #t)))       ; boolean?
            (@ quote ,symbol?)                     ; 'symbol
            (@ @ (* rte))                          ; (@ rte ...)
            (^ ,(λ (x)                             ; (symbol rte ...)
                  (and (symbol? x)
                       (not (member? x '(quote @ ^ any or * + ?
                                         rec let let* letrec submatch
                                         unquote splicing-unquote)))))
               (* rte))
            (@ ^ (* rte))                          ; (^ rte ...)
            (@ any)                                ; (any)
            (@ | (* rte))                          ; (| rte ...)
            (@ * rte)                              ; (* rte)
            (@ + rte)                              ; (+ rte)
            (@ ? rte)                              ; (? rte)
            (@ rec ,symbol? rte)                   ; (rec id rte)
            (@ let    (^ (* (^ ,symbol? rte))) rte) ; (let    ((id rte) ...) rte)
            (@ letrec (^ (* (^ ,symbol? rte))) rte) ; (letrec ((id rte) ...) rte)
            (@ submatch rte)                       ; (submatch rte)
            (@ unquote (any))                      ; ,scheme-exp
            (@ unquote-splicing (any))))           ; ,@scheme-exp
```

If the pattern contains dynamically-computed components, then those parts of the pattern are not known at compile time. In these cases, the macro expands into Scheme code that is essentially a template for the AST—code which will compute the dynamic components and then assemble the rest of the AST around them. This AST-assembly code is then inserted into code which will invoke the pattern compiler on the AST. Essentially, the macro arranges for the compiler invocation it represents to be delayed to run time, thus deferring production of the final automaton to run time. With this exception of handling dynamic components, compiling a *trx* pattern happens entirely at macro-expansion time (that is, compile time). By run time, a source *trx* pattern has already been converted to an equivalent automaton value.

8.2 Abstract syntax

Abstract syntax consists of a handful of record definitions that encode nodes of a *trx* abstract syntax tree (AST). Employing an AST allows us to make the *trx* tool chain independent of the details of our concrete notation; one could try out alternate syntaxes without much work. Furthermore, processing that can be done on the AST can be shared by distinct back-ends that might target different automata models or implementations of those automata.

The AST is defined using a set of record types. Some examples are

```
(define-record ast-sym-node
  symbol    ; Label of root
  children  ; Child patterns
  private)

(define-record ast-seq-node
  quantifier ; One of * + ?
  child      ; Child node
  private)

(define-record ast-choice-node ; (| ...) node
  children   ; List of nodes
  private)

(define-record ast-code-node  ; ,<scheme> node
  code       ; The <scheme> exp (as an s-exp)
  private)
```

Note that each of the records contains a special `private` field. This field is used by the compiler to manage accounting information. For instance, the `private` field tells us, among other things, whether a node has already been visited by the compiler (which is not uncommon due to prevalence of cycles).

Another interesting datatype that is there purely for the convenience of the compiler developers is the AST "handle" node.

```
(define-record ast-handle
  ref)  ; a reference to actual AST
```

Handle nodes are useful when translating recursive patterns, *i.e.*, patterns that begin with `rec` and `letrec`. When these patterns are compiled it is common for one part of the pattern to refer to another part of the pattern that has not been compiled yet. We address this problem by referencing all recursive patterns through a handle node which is first created without a reference field set and is later filled with the reference to the actual abstract syntax tree.

8.3 Automata values

Once an AST is constructed, it must be converted to an automata value. An automata value is represented with yet another record datatype.

```
(define-record sfta
  states              ; Symbol list
  alphabet            ; Symbol list
  labeled-rules
  empty-rules
  final-states)
```

The labelled rules are encoded with

```
(define-record label-rule
  sym-name
  in-state
  out-state
  final-state)
```

and empty rules with

```
(define-record empty-rule
  final-state)
```

Note that the `sfta` record doesn't have fields to support the full requirements of the *trx* notation, such as dynamically-created automata, escapes to Scheme code, and submatches. We delegate tracking of this information to another record datatype:

```
(define-record complex-fta
  sfta             ; Finite tree automaton
  special-states   ; State->inlined-code alist
  submatch-states); Submatched states
```

The `special-states` field is an association list of states and suspended lambda values that correspond to individuals chunks of Scheme code inlined in the *trx* notation. The `submatch-states` field is a list of states at which submatches are to be saved for later retrieval.

Note that the simplified automata ADT permits multiple backends for different models of execution. The one we have implemented uses the automaton itself as a run-time value which is passed, along with a subject tree, to a backtracking SFTA interpreter for execution, which proceeds in a top-down manner. One could consider, alternatively, compiling an SFTA directly to Scheme code; this is something we would like to do.

An earlier version of the system had support for both traditional and simplified automata. The choice in the type of automata value was guided by whether the source pattern included variable-arity constructors. If it did, a simplified automaton would be produced; otherwise, a traditional automaton would be produced by default. Keeping traditional automata in the system allowed for the possibility of "compiling away search" by expanding a non-deterministic TFTA to a deterministic one, buying execution speed for the price of extra compile time and potential state-space explosion. The implementation left the choice of whether to search with a small non-deterministic machine or do fast, non-backtracking execution with an expanded deterministic one in the hands of the application programmer.

We subsequently dropped traditional automata as one of the alternative backend models of computation. (The factoring of the automata ADT into the `sfta` and `complex-fta` records is, in fact, a relic of this earlier implementation—both traditional and simplified FTAs shared inlined-code and submatch annotations by means of the common `complex-fta` record.) As mentioned in Section 3.2, traditional automata are strictly less powerful than simplified tree automata. Keeping two backends to the *trx* compiler did nothing to expand the semantic power of the notation, while it did considerably increase the complexity of our code. Simplifying the implementation made it easier for us to focus on the design of the language; we may revisit automata determinisation as an implementation technology at a later date.

8.4 Compilation

Thus far we have described the way the *trx* compiler is invoked and the types of intermediate and final values it generates. We now describe how these AST and automata ADT values are generated.

The source-level concrete-syntax pattern s-expression is parsed into an AST with a simple recursive translation; the static semantics of the AST are likewise checked with a simple recursive tree-walk that "type checks" the identifier bindings and references.

Translation between ASTs and automata values is a bit more complex. We provide a high-level description of the algorithm, which generates a set of labelled and empty rules for an SFTA from a given abstract-syntax tree.

In the case of a `ast-sym-node`, this entails generating a fresh state that is the "out" state for the label, and a fresh state that is the "out" state for the rightmost child of the AST node. We add empty transitions leading to both of these states. We then obtain the label's "in" state by folding states, beginning with the rightmost child's "out" state, across the children nodes processing each one of the children recursively. Given a label's "out" and "in" states, we generate a fresh final state and add the corresponding rule for that label.

When processing `ast-seq-node` nodes, we restrict the compiler to generate only rules with the same "out" and "final" state. This restriction is necessary to capture the fact that for patterns of the form `(*` *pattern*`)`, the final state after parsing one term matching *pattern* is the same as the "out" state used in translating the term's left sibling. Translation of other AST datatypes is straightforward and follows the same general template.

Macros must produce concrete s-expressions—that is, Scheme source to be handed to the Scheme compiler. Automata values, which are defined as records, do not qualify as such. As a final step, then, the compiler translates the automaton, represented with the ADT, into a Scheme expression describing the direct construction of that value, as a tree of calls to the record constructors. So the `trx` macro finally expands into Scheme code that, when executed at run time, will construct the automaton (as an ADT) used to match the pattern.

8.5 AST-to-AST optimizations

In addition to pattern-to-automaton translation, our compiler also performs some simple optimizations on pattern ASTs. These optimizations are beneficial because the reduced ASTs result in smaller equivalent automata. Some example optimizations are:

- **Propagating `(any)` matches**
  If an `(any)` match is encountered in one of the arms of a choice clause, then the whole clause may be replaced with an `(any)` node. This simplification can bubble up the tree.
- **Propagating dead matches**
  A dead match `(|)` usually allows its containing form to be reduced to a dead match, as well. This simplification also bubbles up through the tree.
- **Merging choice nodes**
  If `(| ...)` forms are nested, they can be flattened into a single such form.

The presence of submatch forms in deleted code can allow an observer to detect some of the transformations, so care must be taken in these cases not to simplify away `submatch` forms that might bind data in the original pattern.

8.6 Executing automata

Our implementation provides a single pattern matcher, the procedure `trx-match`, that takes an tree automaton and an s-expression. The result of a successful match is a *match record* that contains for every submatch (in order of occurrence of the `submatch` clause in the original pattern) a list of submatched terms. We provide a special procedure `(trx-submatch` *m i*`)` for retrieval of submatched information, where *m* is the match record, and *i* is the index of a particular submatch form in the original pattern. Submatch forms are assigned a match-record index in the top/down, left-to-right preorder of the pattern.

The implementation of `trx-match` is a simple SFTA interpreter, which is a pretty direct transcription of the pseudocode of figure 1 into Scheme.

9 Related work

Trx closely follows the choices made in the design of *rx* low-level macro and its associated *sre* regular-expression forms, originally conceived for the *scsh* environment [18]. Similar high-level semantic features, such as choice, repetition, submatches, and the inlining of Scheme code, have similar syntactic encodings in both languages. We were thus able to leverage the design work that went into the *sre* system, and wel also hope that this will make it easier for programmers familiar with the *sre* notation to read and write *trx* patterns, mapping intuition gained from dealing with string-matching patterns to problems in the completely different domain of trees and their patterns.

Tree automata, which provide the foundation of *trx*'s semantics, have enjoyed a consistent flow of research contributions over the last three decades [20, 10, 8, 9]. A resurgence of interest in the late 90s coincided with emergence of semi-structured and tree-structured data, especially in reference to XML [1, 12, 2]. Incidentally, the programming languages built to describe and manipulate tree-structured data have been consistently targeted at handling XML [11, 7]. *trx* differentiates itself from these efforts by combining the benefits of a domain-specific language with the benefits of being able to leverage the functionality of the host language. The authors are not aware of another regular-tree pattern language that is not a standalone domain-specific language.

While we have chosen to embed our little language within Scheme due to the ease of inventing new constructs and translating them into the host language, the functional-programming paradigm is equally valuable to the parsing algorithm, its utility already recognized in the context of XML validation [13]. As evident from the description of the way tree automata process their input, the subsequent application of automata rules to the top-level term and its subterms is naturally recursive.

10 Future work

We are currently developing an SFTA-to-Scheme compiler that will allow static *trx* patterns to be expanded directly into executable Scheme code, rather than requiring an SFTA interpreter. Although there is an existing SFTA-to-Scheme compiler [15], it operates on a more restrictive language than *trx*. For instance, that language does not handle patterns of the form `(| (@` l_1 `...) (@` l_2 `...))` when the labels l_1 and l_2 are identical. We are currently investigating ways to overcome this limitation.

One clear limitation of *trx* is that it builds upon but a tiny fraction of the research work available in the domain of tree automata, semi-structured and tree-structured data, and XML processing. In the future we would like to mine the designs of the existing Scheme-based XML-processing tools SXML and SXSLT [14] to enhance the feature set of *trx*. These systems already employ the functional-programming paradigm to manipulate XML data and have a facility for encoding XML data as s-expressions. Both of these features are consistent with the design goals for *trx*.

We also wish to extend *trx* to incorporate functionality such as ML-style pattern matching and richer functionality for tree matching and transformation. For example, it would be very useful for certain classes of tree structures (such as the mail-order structures given in example 4), to have a pattern that matches, not a sequence of child patterns occuring in a fixed order, but rather the set of child patterns allowed to occur in any order. For example, we could specify that an *order* node must have a *date*, *shipto*, optional *comment*, and *item* child node—but that these children may occur in any order. This would provide logarithmic compression of unordered-sequence patterns, greatly simplifying these kinds of patterns. Some XML pattern-matchers provide this kind of functionality; we'd like to add it to the *trx* notation.

Finally, application of the notation to help write real programs will provide the most valuable feedback on the design of the language, exposing shortcomings and potential areas of extension. (In fact, we're already dissatisfied with the submatch facility and intend to redesign it.) We look forward to gaining more experience with uses of *trx*.

11 Conclusions

Now that the rest of the world has caught on to the benefits of working with semi-structured data with a fixed concrete representation, the importance of tools that help operate on this kind of data is only going to increase—it's reasonable to assume that in the very near future, a significant percentage of the world's data is going to be stored in XML format. The *trx* pattern language, or some future revision of it, can help Scheme programmers describe and operate on this data. Note that while our implementation of *trx* provides for matching patterns against trees that come in the form of s-expressions, neither the design nor the implementation is s-expression specific. Using the notation and adapting our implementation to allow matching and transforming other kinds of labelled trees—such as XML—would not be difficult. Almost the entire code base could be reused in a modular way. The implementor would only need to write a new SFTA interpreter (or compiler) that allows SFTAs to operate upon the new tree structures.

Note also that, just as the Lisp community stole a 40-year march on the rest of the world by adopting semi-structured data early, we have other technologies that continue to bring advantage to the Scheme-programming experience. Chief among these is the ability of Scheme programmers to tightly integrate little languages within Scheme by means of the powerful Scheme macro system. This means that (1) domain-specific extensions can focus on their domain-specific components without needing to re-invent the entire wheel of a general-purpose programming language, and (2) different components of a system written with different domain-specific extensions can be closely coupled within the same program, rather than needing to appear in two completely distinct programs written in two completely distinct domain-specific languages.

This is exactly the story of *trx*—exploiting the domain-specific expressiveness of regular-tree patterns within the powerful, general-purpose framework of the Scheme language.

12 References

[1] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML). *The World Wide Web Journal*, 2(4):29–66, 1997.

[2] F. Bry and S. Schaffert. Pattern queries for xml and semistructured data. Technical report, Institute for Computer Sciences, University of Munich, 2002.

[3] R. D. Cameron. Rex: Xml shallow parsing with regular expressions. Technical report, Simon Fraser University, 1998.

[4] B. Chidlovskii. Using regular tree automata as xml schemas. In *Proc. IEEE Advances in Digital Libraries*, pages 89–104, 2000.

[5] W. Clinger and J. Rees. Macros that work. In *Proceedings of Conference on Principles of Programming Languages*, pages 155–162, 1991.

[6] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. released October, 1 2002.

[7] V. Gapeyev and B. Pierce. Regular object types, 2003.

[8] F. G. Gécseg and M. Steinby. *Tree automata*. Akademiai Kiado, 1984.

[9] F. G. Gécseg and M. Steinby. *Handbook of Formal languages*, volume 3, chapter Tree languages. Springer-Verlag, 1997.

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

[11] H. Hosoya and B. C. Pierce. "XDuce: A typed XML processing language". In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

[12] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.

[13] O. Kiselyov. A better XML parser through functional programming. *Lecture Notes in Computer Science*, 2257, 2001.

[14] O. Kiselyov and K. Lisovsky. XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT. In *International Lisp Conference*, New York, NY, 2003.

[15] M. Y. Levin. Compiling regular patterns. In *Proceedings of*

the eighth ACM SIGPLAN international conference on Functional programming, pages 65–77. ACM Press, 2003.

[16] Alan J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9), September 1982.

[17] O. Shivers. A universal scripting framework or lambda: the ultimate "little language". In *Proceedings of Concurrency and Parallelism, Programming, Networking, and Security: Second Asian Computing Science Conference, ASIAN*, volume 1179 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 1996.

[18] O. Shivers, B. D. Carlstrom, M. Gasbichler, and M. Sperber. *Scsh Reference Manual*, 0.6.4 edition, April 2003.

[19] Olin Shivers. A scheme shell. *Higher-order and Symbolic Computation*. To appear.

[20] J. Thatcher. *Currents in the Theory of Computing*, chapter Tree automata: An informal survey. Prentice-Hall, 1973.

[21] TREX. TREX—Tree regular expressions for XML `http://www.thaiopensource.com/trex/`, 2004.