Evolving TXL

Adrian D. Thurston James R. Cordy

School of Computing
Queen's University
Kingston, Canada
{thurston,cordy}@cs.queensu.ca

Abstract

TXL is a functional programming language specifically designed for expressing source transformation tasks. Originally designed for the rapid prototyping of modest syntactic enhancements, in recent years it has been extensively used in large scale source code analysis and reengineering applications that are much more challenging. As a result, many common programming techniques needed in these larger scale applications are difficult or impossible to express in TXL. Examples include multi-way decisions, generic rules and functions, polymorphism and information hiding. In this paper we introduce ETXL, an experimental extension of TXL which includes convenient features designed to address these issues. Designed to be a compatible variant that remains faithful to the original TXL syntax and semantics, ETXL has itself been prototyped as a source transformation to original TXL.

```
redefine statement
      ...
   |  [coalesced_addition]
end redefine

define coalesced_addition
      [reference] += [expression]
end define

rule transformCoalescedAssignments
   replace [statement]
      V [reference] += E [expression]
   by
      V := V + ( E )
end rule
```

Figure 1. An example of language extension using TXL. This rule adds coalesced assignment to Pascal.

1. Introduction

TXL [2] is a programming language explicitly designed for expressing source transformations. At the heart of the language is a rule-based semantics not unlike term rewriting systems. Overlaid on top is first order functional programming semantics to control the rules. The result is a unique language aptly suited to traversing and rewriting parse trees.

While originally designed to express simple syntactic extensions such as the addition of a coalesced assignment to Pascal (Figure 1) [3], TXL has been adopted for use in a variety of source analysis and transformation tasks of a much larger scale and complexity, including applications involving highly structured rule sets with hundreds of rules, such as LS/2000 [5], multilinguistic website migration [13], aspect analysis and refactoring [7].

Implementation of these applications in TXL introduces a size and complexity that was not envisaged when TXL was designed. In particular, these applications are not just a set of simple independent rewriting rules, but rather involve complex functional programming and decision-making logic; they introduce a range of rule application strategies that are reused repeatedly in different contexts; they involve many detailed variants of the same transformation on different types or patterns; and their sheer size introduces the need for information hiding and modularity. While coding conventions and paradigms such as agile parsing [4] have evolved to help address some of these problems, for the most part TXL is clumsy in meeting these needs.

In this paper we introduce ETXL, an experimental extension to TXL designed to address these issues directly by introducing explicit new language features. In part it helps to alleviate the limitations of TXL itself, and in part it serves as a testbed for understanding features from which we hope to eventually derive the next generation of general-purpose transformational programming languages. In keeping with its heritage, ETXL is itself implemented as a TXL transformation from ETXL to standard TXL [14].

In order to maintain back-compatibility and to concentrate on the features themselves rather than on new language design, we have constrained ourselves in a number of ways. We limit the features of ETXL to have a pure TXL semantics - that is, we cannot modify or extend the semantic model of TXL itself. In this way we can guarantee that our new features will be efficient and familiar to the existing body of TXL users. We limit ourselves to a syntax consistent with existing TXL. In this way we can concentrate on semantics rather than syntactic sugar. We provide only pure language extensions. That is, our new features are orthogonal to the existing features of TXL. In this way our extensions are compatible with existing TXL programs and allow for easy migration of existing programs to use new features.

The paper is organized as follows. Each section begins with a motivating example of existing TXL practice and its limitations, followed by the introduction of the ETXL feature designed to address the problem and a number of examples of its use. Section 7 describes our approach in implementing ETXL and as an example explains the implementation of *if clauses*. Section 8 compares our work to the ways these same problems are addressed in other transformational systems, and Section 9 summarizes our observations, conclusions and future plans.

2. Selection in TXL

TXL programmers currently borrow from rule application and execution semantics to mimic the behavior of a selection construct. Consider the problem of implementing a binary search, shown in Figure 2. Branching is accomplished by applying a different rule for each case of the binary search, with each case's condition embedded into the rule. This technique can be awkward and results in code that is error prone and difficult to read. This is especially true as the complexity of the conditions increases.

The most obvious difference between this form of selection and selection in general-purpose programming languages is that each case is encoded in a separate rule. A more subtle difference is that cases must be explicitly written such that they are mutually exclusive. Even if a particular case succeeds, all successive cases are still tested and may also succeed if not explicitly programmed to exclude the previous cases. What this means for the example in Figure 2 is that even when both less-than and greater-than tests have failed, we must still explicitly test for equality.

In some programming problems, explicitly programming mutual exclusion is not possible because a case whose test has succeeded may alter the data queried, thus invalidating subsequent tests put in place to ensure that the case did not succeed. When different cases see different program state they cannot correctly exclude each other. In TXL

```
function findLeft Key [number]
   replace [node]
      NodeKey [number] NodeVal [value]
      Left [node] Right [node]
   where
      Key [< NodeKey]
   by
      NodeKey NodeVal Left [findAndApply Key] Right
end function

function findRight Key [number]
   replace [node]
      NodeKey [number] NodeVal [value]
      Left [node] Right [node]
   where
      Key [> NodeKey]
   by
      NodeKey NodeVal Left Right [findAndApply Key]
end function

function findHere Key [number]
   replace [node]
      NodeKey [number] NodeVal [value]
      Left [node] Right [node]
   where
      Key [= NodeKey]
   by
      NodeKey NodeVal [transform] Left Right
end function

function findAndApply Key [number]
   replace [node]
      Node [any]
   by
      Node
         [findLeft Key]
         [findRight Key]
         [findHere Key]
end function
```

Figure 2. Selection in Standard TXL. Branching is accomplished by coding each case as a separate rule. The cases must be explicitly programmed to be mutually exclusive.

this kind of problem requires the use of global variables to record when a test has succeeded.

Though using global variables can produce a working solution, in the event that recursion is required the global variable must in fact be a stack of flags in order to be correct, making the implementation far more tedious and error prone. Since one could easily argue that recursion is inevitably required in every non-trivial TXL program, the global variable solution to mutual exclusion leaves much to be desired.

In order to allow conditional selection without requiring the explicit programming of mutually exclusive branches and without requiring the separation of cases into different rules, we have added *if clauses*. Our selection construct behaves similarly to if statements in general-purpose programming languages and yet it does not deviate from existing TXL conventions. The challenge has been to incorporate generalized selection into the functional superstructure that

```
function findAndApply Key [number]
   replace [node]
      NodeKey [number] NodeVal [value]
      Left [node] Right [node]
   if where
      Key [< NodeKey]
   then by
      NodeKey NodeVal Left [findAndApply Key] Right
   else if where
      Key [> NodeKey]
   then by
      NodeKey NodeVal Left Right [findAndApply Key]
   else by
      NodeKey NodeVal [transform] Left Right
   end if
end function
```

Figure 3. Binary search programmed using an ETXL if clause.

exists on top of the rule-based system, without violating the rule-based semantics. Figure 3 demonstrates the use of an if clause to implement the binary search of Figure 2.

We permit any sequence of TXL clauses in both the test and body blocks of an if branch. Should all clauses in the if test succeed, the body block is entered and no more branches are considered. Once a body block has been entered, the success of the clause list containing the if clause is determined by the clauses in the body block. If all of these clauses succeed, control is passed to the code following the if clause.

If any clause in an if test fails, the rest of the test block and the corresponding body block is abandoned. The test block of the next else-if clause is then tried. If no test block succeeds, flow continues to the else clause, if there is one.

While the if-clause expression of the binary search in Figure 3 is much more compact and intuitive than the original, it still suffers from a lack of generality. Such a binary search should also be reusable, allowing for arbitrary key types and transformation rules to be applied, not just `[number]` and `[transform]`. In the next section we address the problem of writing more generic and reusable code in TXL.

3. Generics

A common problem faced by TXL programmers is that given the need to reuse a transformation, there is often no choice but to duplicate code. This can happen in a number of scenarios. For example, TXL begins to show signs of strain when one must employ a custom search strategy a number of times. Suppose there are many times in a program when a tree must be traversed in a common way looking for an instance of a pattern, but in each use of the traversal the target should be rewritten using a different rule.

Since TXL is a functional programming language using copy-on-write, value-based semantics, it is not possible to obtain a pointer to the subtree of interest using a generic search strategy and then modify it using the returned pointer. In some cases it may be possible to extract a subtree of interest, apply a rule to modify it and then reinsert it after modification, but this is awkward and not always possible. In most cases in TXL we have no choice but to embed the rule that modifies the tree directly into the search strategy.

In other cases, a search strategy or transformation is written multiple times with the only difference between the versions being the type of tree on which they operate. For example, this can happen when writing list processing algorithms such as sort routines or list reversals.

A need to duplicate code and differentiate it only by type can also arise along with grammar type specializations. A type specialization in grammar programming involves duplicating and slightly modifying a section of a grammar with possibly a smaller, larger or just plain different set of allowable strings. In such cases, a lack of language features for specifying type independence prevents us from applying rewrite rules that were designed for the original types on the differently-named type specializations, where applicable.

To enable the writing of generic search strategies and rewrite rules we have added *rule parameters* and *type parameters*. Both rule and type parameters are given in parameter and argument lists alongside standard tree-based parameters. In a parameter list, a rule parameter is specified by using the keyword `rule` instead of a tree type. A type parameter is specified using the keyword `type`.

Inside a rule parameter specification, the list of parameters that the rule parameter itself will accept is given following the `rule` keyword. Full specification of the signature of the rule parameter is necessitated by the fact that TXL rules are statically typed. Requiring the full signature with the rule parameter also ensures that a rule with rule parameters can be semantically checked and implemented independently of the instances of its use. Unfortunately, the same is not true of type parameters, and these must be checked dynamically.

At rule invocation point, a rule can be passed as an argument either by naming a defined rule or by referencing a local rule parameter. No change to the syntax is required to support the passing of rules as arguments. Inside a rule, a rule parameter is available for use in the rule body as if it were a defined rule.

A type can be passed as an argument either by specifying a literal type or by referencing a local type parameter. A literal type may either be a pure type such as `[declarator]` or it may contain a modifier as in `[repeat statement]`. Inside a rule, a type parameter may also be used as if it were a defined type.

Rule and type parameters allow the expression of a

```
rule sort T [type] LessThan [rule [T]]
   replace [repeat T]
      N1 [T] N2 [T] Rest [repeat T]
   where
      N2 [LessThan N1]
   by
      N2 N1 Rest
end rule

...
   construct Sorted [repeat pair]
      Pairs [sort [pair] pairLess]
```

Figure 4. A generic sorting routine that can be applied to a list of any type. The sort routine requires a less-than operator for comparing items.

generic sort as shown in Figure 4. This sort can operate on any type for which a less-than rule can be written.

In the next section we present a solution to another form of abstraction that TXL is lacking, pattern abstraction.

4. Pattern abstraction using out parameters

The principal method of analyzing a parse tree in TXL is the pattern match. Pattern matches serve two purposes in TXL, both specifying the shape and elements of interesting parse trees and extracting matched subtrees for later use. TXL has no language feature for naming or abstracting patterns, making it difficult to separate code that needs to extract subtrees for later use from the patterns that are used to extract them.

The ability to abstract pattern matching would allow TXL programmers to better organize code in a manner similar to the role of subroutines in imperative languages, allowing a pattern to be specified once and used multiple times.

Like most functional languages, TXL allows only a single return value from a rule. The result is returned from a rule as the transformed scope of application, with the tree to be searched passed in as a parameter. An elided binary search retrieval function implemented in this way is shown in Figure 5. Unfortunately, many patterns bind more than one variable for later use. Should multiple bindings need to be returned, the TXL programmer must encapsulate the multiple return values using a new grammar definition for the scope of the searching rule, or alternatively use global variables to hold the multiple results.

Instead of returning values via a rule's search scope, or using global variables, TXL users should be able to encapsulate the extraction of matched subtrees using native language features. Figure 6 gives an example of a scenario in which encapsulation would be beneficial. In this example, two rules use the same pattern for different purposes. TXL

```
function findValue Tree [tree] Key [number]
   replace [opt value]
      % Empty
   ...
   by
      Value
end function

...
   construct OptValue [opt value]
      _ [findValue Tree Key]
   deconstruct OptValue
      Value [value]
```

Figure 5. In standard TXL the only way to return a value from a rule is via the tree type on which it operates. Only one value may be returned.

```
rule rewriteAssignment1
   replace [statement]
      assign( Id [id], Expr [expr] );
   where
      Id [needToRewrite]
   where
      Expr [isConst]
   by
      Id [_ 'set] ( AssignExpr );
end rule

rule rewriteAssignment2
   replace [statement]
      assign( Id [id], Expr [expr] );
   where
      Id [needToRewrite]
   where
      Expr [isConst]
   by
      Id = Expr;
end rule
```

Figure 6. In standard TXL we have no choice but to duplicate a pattern each time it is required.

programmers presently have no choice but to duplicate the pattern in such cases.

In order to allow pattern abstractions, in ETXL we have added *out parameters*. An out parameter binds a value from a called rule to a variable name in a calling rule. Out parameters are separated from normal (in) parameters in parameter and argument lists using a colon. Inside the called rule, an out parameter must be bound either as a binding variable in a pattern, using an explicit variable constructor, or as an out parameter of another rule invocation. The behaviour of out parameters is designed to mimic the behaviour of pattern matches in TXL. If a pattern fails to find a match then the variables of the pattern are not bound and control cannot proceed past the pattern. Similarly, if a rule fails to bind an out parameter then the caller is unable to proceed.

```
rule assignPat : Id [id] Expr [expr]
   match [statement]
      assign( Id [id], Expr [expr] );
   where
      Id [needToRewrite]
   where
      Expr [isConst]
end rule

rule rewriteAssignment1
   replace [statement]
      Statement [statement]
   where
      Statement [assignPat : Id [id] Expr [expr]]
   by
      Id [_ 'set] ( AssignExpr );
end rule

rule rewriteAssignment2
   replace [statement]
      Statement [statement]
   where
      Statement [assignPat : Id [id] Expr [expr]]
   by
      Id = Expr;
end rule
```

Figure 7. Use of out parameters to abstract a pattern.

The successful transfer of an out parameter to the caller is independent of the success or failure of the rule in which it was bound. A rule that succeeds in binding an out parameter but fails in matching its primary pattern will not necessarily fail the caller clause list. Maintaining these two mechanisms as independent processes helps to keep the design simple and the semantics consistent with the rest of TXL.

On the face of it it might seem that out parameters breach TXL's functional semantics. However, since the semantics of a rule invocation with out parameters is the same as a TXL rule invocation on a scope that gathers together the original scope of the rule with the out parameters, we can say that we have not violated TXL's functional semantics.

Because TXL rules have implicit local backtracking, any clause that follows the replace clause of a rule may potentially be executed more than once during a single rule invocation. This means that an out parameter may be bound more than once before it is used. As is the case for local variables, the last value bound is the binding used.

In TXL, the introduction of a new variable is always accompanied by its type in order to allow for local static type checking in patterns and replacements and to reduce ambiguity for the reader. For these same reasons, the introduction of new variables in out argument lists follows this same design principle, and out arguments in ETXL are explicitly typed in the calling argument list even though their type can be inferred from the signature of the called rule.

The use of out parameters to abstract a pattern match

```
function find Key [number] : Value [value]
   match [node]
      NodeKey [number] NodeValue [value]
      Left [node] Right [node]
   if where
      Key [< NodeKey]
   then where
      Left [find Key : Value [value]]
   else if where
      Key [> NodeKey]
   then where
      Right [find Key : Value [value]]
   else construct Value [value]
      NodeValue
   end if
end function

...
   where
      Tree [find Key : Value]
```

Figure 8. Returning a result value using an out parameter.

is demonstrated in Figure 7. Out parameters allow us to eliminate pattern duplication.

Returning to the example of Figure 5, in which the scope of a function is used to return a value, we can now use out parameters to move the return value to the parameter list and the tree to be searched can go back to being the object of the search function where it should more naturally be. This is shown in Figure 8.

Out parameters also find many other uses in ETXL, including the propagation of synthetic attributes up the tree when explicitly walking a parse tree. In the section that follows we show how we can combine out parameters with rule parameters to attain another pattern-based programming construct which is missing from TXL - parameterized patterns.

5. Parameterized patterns

Out parameters provide a language-based solution to the reuse of the same pattern in various different contexts. The inverse problem, the need to use different patterns in an otherwise identical surrounding rule, is also common in TXL programs. An example is shown in Figure 9. If we combine out parameters with rule parameters we have an elegant solution to this problem. Since we can use out parameters to encapsulate a pattern in a rule, we can parameterize such rules to gain *pattern parameters*.

If we use pattern parameters in cases where we require many instances of a rule, each time with different patterns that bind the same variables, we are freed from having to duplicate the surrounding code. We can write a single generic rule, passing the method of pattern matching in as a parameter. In Figure 10, a rule parameter binds the left hand

```
rule replaceAssign1
   replace [statement]
      Id [id] = Expr [expr];
   by
      Id [_ 'set] ( Expr );
end rule

rule replaceAssign2
   replace [statement]
      assign( Id [id], Expr [expr] );
   by
      Id [_ 'set] ( Expr );
end rule

...
   by
      Program
        [replaceAssign1]
        [replaceAssign2]
```

Figure 9. Use of multiple patterns in otherwise identical context requires duplication of the surrounding code.

```
function getAssign1 : Id [id] Expr [expr]
   match [statement]
      Id [id] = Expr [expr];
end function

function getAssign2 : Id [id] Expr [expr]
   match [statement]
      assign( Id [id], Expr [expr] );
end function

rule genericReplace
      AssignPat [rule : [id] [expr]]
   replace [statement]
      Stmt [statement]
   where
      Stmt [AssignPat : Id [id] Expr [expr]]
   by
      Id [_ 'set] ( Expr );
end rule

...
   by
      Program
        [genericReplace getAssign1]
        [genericReplace getAssign2]
```

Figure 10. An example of the parameterization of patterns by using out parameters in combination with rule parameters.

side (LHS) and right hand side (RHS) of a value assignment using out parameters and thereby acts as a parameterized pattern. This allows us to express a generic replacement routine that can easily be used with various different patterns that locate value assignments, provided that the patterns bind the necessary LHS and RHS elements.

In the next section we introduce our final enhancement to the TXL programming language - the need to provide modularity and information hiding.

6. Modularity

TXL was originally designed for small rapid prototyping tasks. Now it finds use in large production transformations with many thousands of lines of code, on which multiple developers work. As the size of TXL programs grows, a mechanism for information hiding becomes increasingly necessary. The ability to maintain a section of a program independent of the remainder of the program is very important when developing large systems. Collaborating developers must be concerned about such issues as avoiding name collisions and specializing rule names to reflect the areas of the TXL program in which they are meaningful.

In existing TXL programs, avoiding name collisions is accomplished by employing naming conventions whereby all entities are prefixed with a qualifying name. Private entities can be further distinguished by prepending an underscore to the name. This convention works, but it is somewhat tedious as every name definition and associated reference must be prefixed. Moreover, there is no language support for these conventions, and errors or "cheats" that violate the intended modularity are too easily possible.

Naming conventions can succeed at avoiding name collisions but do not permit the true hiding of names. That is, there can be no enforced distinction between entities that are private and entities that are public. Name hiding is very important in large-scale systems because its use can dramatically simplify the comprehension of a program. It is a key requirement when maintaining a library that implements a carefully designed interface.

Another problem with simple naming conventions is that there is no way to elide the use of a frequently used qualifying name as a matter of convenience.

Given these limitations we have added to TXL a facility for defining abstraction layers that allow developers to either hide or expose grammar definitions, rules and global variables and to group code into logical program modules. Our *module system* aids in collaboration on large projects, eases the writing of generic libraries and improves the quality of TXL software from a software engineering perspective.

We chose to implement the module system at the language level rather than at the file level because TXL already has a convenient and heavily used source file inclusion system. In ETXL, transformation rules and functions, grammar defines and global variables can be modularized by wrapping them in a module statement. By default, an entity within a module is private and as such is not accessible outside of the module. An entity may be made public by listing it in the *public* statement. A public entity may then be referenced outside of its module by qualifying it with its module name and a dot. Figure 11 shows an HTML markup module.

```
module HTML
   public
      boldize
   end public

   define item
      [begin_tag] [any] [opt end_tag]
   end define

   define begin_tag
      < [id] [repeat option] >
   end define

   define end_tag
      < / [id] >
   end define

   function boldize
      replace [any]
         A [any]
      construct BoldTag [begin_tag]
         <B>
      by
         A [tagwith BoldTag]
   end function

   function tagwith Tag [begin_tag]
      deconstruct Tag
         < TagId [id] TagOptions [repeat option] >
      replace [any]
         Anything [any]
      construct TaggedThing [item]
         <TagId TagOptions> Anything </TagId>
      deconstruct TaggedThing
         TaggedAnything [any]
      by
         TaggedAnything
   end function
end module

rule tagIds
   replace $ [id]
      Id [id]
   by
      Id [HTML.boldize]
end rule
```

Figure 11. HTML markup module.

The need to qualify a public entity can be alleviated by importing its module with the *using* statement. The using statement is permissible in the top level scope as well as in a module. Once a module has been imported, all its public entities are accessible without qualification as if they were directly defined in the importing scope. Consequently, no collisions between the public names of the imported module and the existing names in the scope can exist in order for the import to succeed. Previously imported names are included in this collision check. That is, it is not possible to import the same name into a module more than once.

ETXL modules, like all of our new features, have been implemented as a TXL transformation. In the following section we describe our overall approach, using the implementation of if clauses as an example.

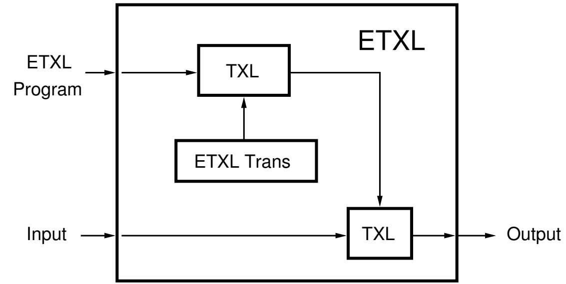

Figure 12. Implementation architecture.

7. Implementation

In a testament to the usefulness of TXL as a language design tool we have chosen to rapidly prototype ETXL using a TXL transformation. It is, by design, a natural choice for this problem. Since we are extending the TXL language, we also use it as the target of the transformation, transforming ETXL to standard TXL in a manner that preserves all existing language features as well as implementing the new features we have designed.

Figure 12 illustrates our overall approach. Our prototype aims to be complete and directly usable by the TXL community, with the hope that once our new language features have been tested and tuned in a production environment they can be re-implemented directly in the TXL engine. In the remainder of this section we describe the implementation of ETXL if clauses as an example of the transformation.

7.1. If clause implementation

If clauses are implemented by splitting each if test into two function calls and invoking them both in the way that a programmer might implement a multi-way decision by hand in TXL. One function explores the test-success branch and the other goes down the test-failure branch. Since propagating subsequently bound variables back to the caller is not possible in standard TXL, we do not return control to the caller at the point of the split. Instead, each branch is responsible for executing its case and the rest of the clauses that follow the if clause. This requires us to propagate local variables forward using parameters.

In order to implement mutual exclusion between test branches, our implementation uses a global stack of flags for indicating when a control path has been taken. The rule implementing the test-success branch first pushes the value *false* (0) to the stack, it then proceeds to evaluate the test. Upon success of the test, it overwrites the top of the stack with the value *true* (1) and proceeds into the body of the branch.

```
function findAndApply Key [number]
  replace [node]
    _This [node]
  deconstruct _This
    NodeKey [number] NodeVal [value]
    Left [node] Right [node]
  by
    _This [_branch_1 Key NodeKey NodeVal Left Right]
      [_branch_2 Key NodeKey NodeVal Left Right]
end function

function _branch_1 Key [number] NodeKey [number]
    NodeVal [value] Left [node] Right [node]
  construct _ [any]
    _ [_if_stack_push_false]
  replace [node]
    _This [node]
  where
    Key [< NodeKey]
  construct _ [any]
    _ [_if_stack_set_top_true]
  by
    NodeKey NodeVal Left [findAndApply Key] Right
end function

function _branch_2 Key [number] NodeKey [number]
    NodeVal [value] Left [node] Right [node]
  construct _IfStackTop [number]
    _ [_if_stack_pop]
  deconstruct _IfStackTop
    0
  replace [node]
    _This [node]
  by
    _This [_branch_3 Key NodeKey NodeVal Left Right]
      [_branch_4 Key NodeKey NodeVal Left Right]
end function

function _branch_3 Key [number] NodeKey [number]
    NodeVal [value] Left [node] Right [node]
  construct _ [any]
    _ [_if_stack_push_false]
  replace [node]
    _This [node]
  where
    Key [> NodeKey]
  construct _ [any]
    _ [_if_stack_set_top_true]
  by
    NodeKey NodeVal Left Right [findAndApply Key]
end function

function _branch_4 Key [number] NodeKey [number]
    NodeVal [value] Left [node] Right [node]
  construct _IfStackTop [number]
    _ [_if_stack_pop]
  deconstruct _IfStackTop
    0
  replace [node]
    _This [node]
  by
    NodeKey NodeVal [transform] Left Right
end function
```

Figure 13. Implementation of Figure 3 as transformed to standard TXL by the ETXL prototype. Branch one implements the first if test and its body block. Branch two ensures that the if test failed, then proceeds with the remaining cases. Branch three implements the else-if test and its body block. Branch four ensures that the else-if test failed then executes the else part.

The rule implementing the test-failure branch, which by TXL semantics is invoked regardless of the outcome of the test, first examines the top of the stack. If the value is false it may then proceed to either invoke another branch point for an else-if test or it may fall into an else block. Since the rule implementing the test-failure branch is always executed, it is given the responsibility of popping the test value from the stack. Figure 13 shows the TXL code implementing the ETXL binary search shown in Figure 3.

The rule-based semantics at the core of TXL presents an additional challenge in the implementation of if clauses. The default behaviour of a TXL rule is to repeatedly search the tree to which it is applied after each replacement until no further instances can be found. Specifically, when rule is applied it traverses the parse tree in a preorder manner, looking for instances of the pattern. When a match is found, the pattern refinements (other clauses of the rule) are tested. If they succeed, the replacement is made and the rule is reapplied at the root of the resulting tree. Once no pattern match with successful refinements can be found, the rule terminates.

Because our prototype implementation uses subrules to implement if branches, pattern refinements may be moved into a generated subrule when an if clause splits a pattern from one or more of its refinements. Since the constructors containing the generated subrule invocations will always be successful, the rule will no longer terminate properly. To solve this problem, we must propagate the notion of rule failure from the generated subrules back to the top-level rule. Figure 14 shows an example of a pattern and refinement which gets split across multiple rules by the implementation of if clauses. In this example, *deconstruct* statements are used to determine what the replacement should be. The need for explicit termination of search reapplication can also arise when an if clause separates a pattern from its refinements, because the transformation pushes clauses following an if clause into the rules implementing the branches.

In order to preserve rule termination behaviour, our transformation of if clauses adds code to explicitly transfer failure from the generated subrules back to the top level rule. Once again, a global stack of boolean flags is used to address this control flow problem. Immediately before the first branch point separating a pattern and a replacement, the value false is pushed to this stack. Each of the subrules implementing a branch of the if clause replaces this top entry with the value true if all of its embedded clauses succeed. The top-level rule containing the original if clause then pops the top entry and uses a deconstructor to test that it is true (that is, that some branch succeeded) before continuing. If the popped entry is not true (that is, none of the branches succeeded) then the deconstructor fails and the rule terminates normally.

```
rule eliminateIf
   replace [repeat stmt]
      'if ( Expr [expr] )
         Stmt [stmt]
      Rest [repeat stmt]
   construct Label [id]
      _ [uniqueLabel]
   if
      deconstruct Expr
         Var [id] = _ [expr]
      by
         Expr;
         test_goto ( ! Var ) Label;
         Stmt
         Label: noop;
         Rest
   else
      deconstruct Expr
         Var [id] == SubExpr [expr]
      by
         test_goto ( Var != SubExpr ) Label;
         Stmt
         Label: noop;
         Rest
   end if
end rule
```

Figure 14. An ETXL example which requires the explicit termination of the original rule. Although the main pattern will remain in the top-level rule, pattern refinements and replacements appearing inside and following the if clause will be moved into subrules for the if branches as part of our implementation, causing problems with termination.

The corresponding implementation of Figure 14 is shown in Figure 15.

8. Related work

The features we have added in ETXL are by no means unique. General functional programming languages such as Lisp [9] and Haskell [8] have always offered explicit if-then-else constructs. Our if-then-else is distinguished from these and others however in its application in the pattern-matching context. Based on the TXL success-fail semantics borrowed originally from Snobol [6], ETXL's if-then-else allows for any subchains of the pattern-match-deconstruct-construct-guard-replace-by sequence of a guarded rewrite rule to be conditionalized, allowing for a range of kinds of applications quite unlike traditional if-then-else.

Other source transformation systems have also offered solutions to the challenges addressed by our extensions to TXL. For example, ASF+SDF [15] has an inherently modular structure, in which subgrammars and their associated transformation rules are packaged and maintained together as modules, which are then combined to form whole systems. ETXL modules are in many ways similar.

```
rule eliminateIf
   replace [repeat stmt]
      _This [repeat stmt]
   deconstruct _This
      'if ( Expr [expr] ) Stmt [stmt]
      Rest [repeat stmt]
   construct Label [id]
      _ [uniqueLabel]
   construct _ [any]
      _ [_repl_stack_push_false]
   construct _Result [repeat stmt]
      _This [_branch_1 Expr Stmt Rest Label]
         [_branch_2 Expr Stmt Rest Label]
   construct _ReplStackTop [number]
      _ [_repl_stack_pop]
   deconstruct _ReplStackTop
      1
   by
      _Result
end rule

function _branch_1 Expr [expr] Stmt [stmt]
      Rest [repeat stmt] Label [id]
   construct _ [any]
      _ [_if_stack_push_false]
   replace [repeat stmt]
      _This [repeat stmt]
   deconstruct Expr
      Var [id] = _ [expr]
   construct _ [any]
      _ [_if_stack_set_top_true]
   construct _ [any]
      _ [_repl_stack_set_top_true]
   by
      Expr; test_goto ( ! Var ) Label;
      Stmt Label: noop; Rest
end function

function _branch_2 Expr [expr] Stmt [stmt]
      Rest [repeat stmt] Label [id]
   construct _IfStackTop [number]
      _ [_if_stack_pop]
   deconstruct _IfStackTop
      0
   replace [repeat stmt]
      _This [repeat stmt]
   deconstruct Expr
      Var [id] == SubExpr [expr]
   construct _ [any]
      _ [_repl_stack_set_top_true]
   by
      test_goto ( Var != SubExpr ) Label;
      Stmt Label: noop; Rest
end function
```

Figure 15. Corresponding implementation of Figure 14 which shows the explicit termination of the top level rule. Before invoking the subrules implementing the if branches, the top-level rule pushes the value false (0) to the global stack. In each of the generated subrules, this value is changed to true (1) if all of the pattern refinements and replacements in the subrule succeed. Upon return to the top-level rule, the value is popped and tested using a deconstructor. If the popped value is true then rule continues, otherwise it terminates normally.

Stratego [16] also provides a similar modular structure, but at a higher level of abstraction. In Stratego, more general aspects and levels of understanding can be modularized. Stratego is focussed on reuse and is particularly strong in its generic facilities, most notably generic traversal strategies, which allow separation of rules from their application and application orders independent of specific syntax structures, and dynamic rules, which allow for instance-specific contextualization of rewriting rules. ETXL abstracts strategies using higher order rules (i.e., rule and type parameters) and uses out parameters to augment TXL's usual parameter-based instance contextualization.

Both ASF+SDF and Stratego handle multi-way conditions using guarded conditional rewriting much like TXL's rules, and have no explicit if-then-else. Stratego also offers a deterministic choice operator which eliminates the need to explicitly program mutual exclusion. This binary operator applies a second rule on the condition that the first has failed. In many cases, this operator can be used to address uses of ETXL's if-then-else.

Systems such as ANTLR [12], APTS [11, 10] and DMS [1], which use semi-imperative transformation specifications, can take advantage of the language constructs of their host imperative languages to handle if-then-else and modularity, but can lose some of the declarative nature of the rewriting process as a result.

9. Conclusion

In this work we have addressed a number of real problems encountered by TXL users. If clauses ease the implementation of multi-way decisions. Rule and type parameters make it possible to specify rules that are independent of the types they operate on and the subrules they apply. Out parameters allow users to abstract patters into independent and reusable units. The combination of rule parameters and out parameters allows one to parameterize such patterns. Our modularity system allows large, colloborative works to be better engineered. All of our new features enable the expression of programs that were previously either tedious or prohibitively difficult to express in standard TXL.

Once we have more experience with these features using our prototype implementation, they may be re-implemented as part of the TXL engine itself. In the long run we plan to incorporate what we have learned from enhancing TXL into a new transformation system targeted at a much broader audience. In the meantime, our ETXL prototype is available for evaluation from the project home page [14].

References

[1] I. D. Baxter. Parallel support for source code analysis and modification. In *2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pages 3–14, 2002.

[2] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[3] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, January 1991.

[4] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. Agile parsing in TXL. *Journal of Automated Software Engineering*, 10(4):311–336, October 2003.

[5] T. R. Dean, J. R. Cordy, K. A. Schneider, and A. J. Malton. Experience using design recovery techniques to transform legacy systems. *17th IEEE International Conference on Software Maintenance (ICSM'01)*, pages 622–631, November 2001.

[6] D. J. Farber, R. E. Griswold, and I. P. Polonsky. Snobol, a string manipulation language. *Journal of the ACM*, 11(1):21–30, 1964.

[7] S. Grant and J. R. Cordy. An interactive interface for refactoring using source transformation. In *1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE'03)*, pages 30–33, Victoria, November 2003.

[8] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell version 1.1. Technical report, Departments of Computer Science, University of Glasgow and Yale Universtiy, Aug 1991.

[9] J. McCarthy et al. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[10] R. Paige. Apts external specification manual, 1993. http://www.cs.nyu.edu/∼jessie/.

[11] R. Paige. Viewing a program transformation system at work. In *Programming Language Implementation and Logic Programming: 6th International Symposium (PLILP'94)*, volume 844 of *Lecture Notes in Computer Science*, May 1994.

[12] T. J. Parr. An overview of sorcerer: A simple tree-parser generator, 1994.

[13] N. Synytskyy, J. R. Cordy, and T. R. Dean. Robust multilingual parsing using island grammars. In *2003 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'03)*, pages 149–161, Toronto, October 2003.

[14] A. Thurston. ETXL homepage, 2006. http://www.cs.queensu.ca/home/thurston/etxl/.

[15] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[16] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Spinger-Verlag, May 2001.