Evolving TXL

Adrian Thurston and James R. Cordy

Software Technology Laboratory, School of Computing
Queen's University, Kingston, Ontario, Canada

Introduction

- TXL originally designed for small program transformation tasks
- Aid in the development of the Turing Programming Language

- Developers are writing larger TXL programs
- Multiple developers involved in projects
- Application domain has grown
- TXL is now used for problems unforeseen during original design
- Want to improve TXL in response to these changes
- Cater to existing users and hope to gain new users

Design Goals

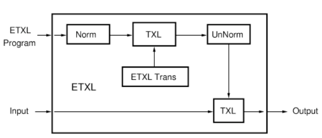
- Enable modularity and abstraction
- Introduce general purpose language features
- Increase expressiveness
- Minimize the use of globals variables
- Support TXL's unique paradigms

Nine New Features

1. Must Matching Rules
2. Objectless Rules
3. Strong Typing
4. Nested Rules
5. Rule Parameters
6. Type Parameters
7. If Clauses
8. Out Parameters
9. Modularity

+ TXL = ETXL

Approach



Must Matching Rules

- Rules that fail to make a match silently return the original tree
- In most cases this is the desired behaviour

- Sometimes a rule is written such that it is expected to match
- Examples:
  - Patterns that involve necessary conditions for semantic legality
  - Multi-stage transformations
- Onus is on the programmer to verify that these rules are actually matching
- Whether or not these rules actually match often goes unchecked

- Whether or not a rule is to always match is a static property
- Should be expressible in the language
- Prepend the must keyword to a replace clause
- Enables the TXL engine to verify that the rule is indeed matching
- If a must matching rule fails to match a run-time error is raised

Objectless Rules

- TXL has two kinds of rules
  - Replacing rules
  - Matching-only rules

```
rule reverse
    replace $ [pair]
        N1 [number] N2 [number]
    by
        N2 N1
end rule
```

```
rule consecutive
    match [pair]
        N1 [number] N2 [number]
    construct Diff [number]
        N1 [- N2]
    deconstruct N1
        1
end rule
```

- Sometimes it is not necessary to use the main pattern of a rule
- One simply wants to program a sequence of operations
- These rules are neither replacing nor matching rules
- One uses a match any clause in place of the main pattern

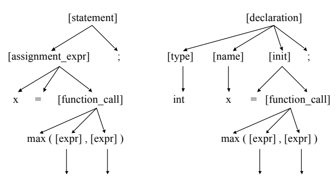
```
function checkErrorsCount
    match [any]
        _ [any]
    import SyntaxErrs [number]
        0
end rule
```

```
function checkErrorsCount
    import SyntaxErrs [number]
        0
end rule
```

- We have added objectless rules which do not require a match or replace clause

Strong Typing

- A rule may succeed from different tree roots, but ...
  ... be a valid transformation from only one root
- Want to communicate this restriction on application in the language
- Environments where code is maintained by several programmers
- Compiler enforcement



Solution:

- Strong typing allows the programmer to specify an exclusive type to which a rule may be applied
- Application of a rule to the wrong type generates a compile-time error

Out Parameters

- Cannot return data from a rule without using globals or the primary pattern
- Not possible to define an abstraction layer between code that needs to deconstruct a tree and code that does the deconstruction.

Solution:

- Allow rules to return values
- Can create rules that function as patterns
- Analogous to abstracting away code by pushing it into a function call

If Clauses

```
function findLeft Key [id]
    match [tree]
        NodeKey [id] NodeVal [id]
        Left [tree] Right [tree]
    where
        Key [< NodeKey]
    where
        Left [find Key]
end function

function findRight Key [id]
    match [tree]
        NodeKey [id] NodeVal [id]
        Left [tree] Right [tree]
    where
        Key [> NodeKey]
    where
        Right [find Key]
end function
```

```
function findHere Key [id]
    match [tree]
        NodeKey [id] NodeVal [id]
        Left [tree] Right [tree]
    where
        NodeKey [= Key]
end function

function find Key [id]
    match [tree]
        Tree [tree]
    where
        Tree
            [findLeft Key]
            [findRight Key]
            [findHere Key]
end function
```

- As complexity increases, programming mutual exclusion gets harder
- Want a native branching clause that is
  - Familiar to programmers
  - Easy to use

```
function find Key [id]
    match [tree]
        NodeKey [id] NodeVal [id]
        Left [tree] Right [tree]
    if where
        Key [< NodeKey]
    then where
        Left [find Key]
    else if where
        Key [> NodeKey]
    then where
        Right [find Key]
    else construct _ [id]
        NodeVal [print]
    end if
end function
```

Rule Parameters

- Traversals and pattern matching are independent processes
- Should be expressing them independently
- Custom tree traversal is specified by manually programming rule applications from within replacements
- Traversals and pattern matching are closely tied together

Solution:

- Separate traversals and pattern matching with rule parameters
- Write the traversal
- Parameterize it by the rules to apply

Type Parameters

- Parse tree structure and parse tree types are indepenedent constructs
- Several types can share the same basic structure
- An ability to specify operations on a structure independent of the specific types involved is desirable
- Examples: list reversal, sorting, walking of homogeneous trees

Solution:

- Type parameters enable this abstraction
- Write the operation on the structure
- Parameterize it by specific types

Pattern Parameters

- Combining out parameters with rule parameters gives pattern parameters
- Allows one to define a rule parameter that is expected to return a value

```
rule genericReplace PatternParam [rule : [id] [expression]]
    replace $ [statement]
        Stmt [statement]
    where
        Stmt [PatternParam : Id [id] Expr [expression]]
    by
        Id [_ 'set] '( Expr ') ';
end rule
```

Nested Rules

- Tree traversals often require the propagation of data down the tree
- Normally implemented by pausing a traversal, collecting data, then passing the data down to deeper parts of the traversal using parameters
- Can result in exessive parameter passing

```
function meetsPrefixCriteria ClassKey [class_key]
        ClassId [id] OptBase [opt base_clause]
        FuncDeclSpec [repeat decl_specifier]
        FuncId [id]
    ...
end function

rule prefixInFunc ClassKey [class_key] ClassId [id]
        OptBase [opt base_clause]
        FuncDeclSpec [repeat decl_specifier] FuncId [id]
    replace $ [init_declarator]
        Id [id] OptInit [opt initializer]
    where
        Id [meetsPrefixCriteria ClassKey ClassId
            OptBase FuncDeclSpec FuncId]
    by
        ClassId [_ FuncId] [_ Id] OptInit
end rule

rule prefixInClass ClassKey [class_key] ClassId [id]
        OptBase [opt base_clause]
    replace $ [function_definition]
        FuncDeclSpec [repeat decl_specifier]
        FuncDeclarator [declarator]
        FuncBody [function_def_body]
    deconstruct * [id] FuncDeclarator
        FuncId [id]
    by
        FuncDeclSpec FuncDeclarator
        FuncBody [prefixInFunc ClassKey ClassId
            OptBase FuncDeclSpec FuncId]
end rule

rule prefixLocals
    replace $ [class_specifier]
        ClassKey [class_key] ClassId [id]
            OptBase [opt base_clause]
            { MemberSpec [opt member_specification] }
    by
        ClassKey ClassId OptBase
            { MemberSpec [prefixInClass
            ClassKey ClassId OptBase] }
end rule
```

Solution:

- Permit a rule to be nested in another
- Implicit access to variables declared in an ancestor
- No need to plan ahead what to propagate
- Reduced need for editing of parameter lists

Additional Benefit:

- Nested rules allow the programmer to group code at the task level

Modularity

- TXL programs have grown
- Multiple developers have become involved in single projects
- Some mechanism for information hiding becomes necessary

Currently:

- Modularity features can be emulated by naming conventions
- Emulation leaves much to be desired
- Instead some modularity features should be incorporated into the language

Allow Programmers To:

- Independently maintain sections of code without being concerned about name collisions.
- Hide internals
- Define reduced abstraction interfaces

Solution:

- New modularity statement
- Grammar definitions, rules and global variables may have their names encapsulated in a module statement
- Entities are private by default and may be made public