Excerpts from the TXL Cookbook

James R. Cordy

School of Computing, Queen's University
Kingston ON, K7L 3N6, Canada
cordy@cs.queensu.ca
http://www.cs.queensu.ca/~cordy

Abstract. While source transformation systems and languages like DMS, Stratego, ASF + SDF, Rascal and TXL provide a general, powerful base from which to attack a wide range of analysis, transformation and migration problems in the hands of an expert, new users often find it difficult to see how these tools can be applied to their particular kind of problem. The difficulty is not that these very general systems are ill-suited to the solution of the problems, it is that the paradigms for solving them using combinations of the system's language features are not obvious.

In this paper we attempt to approach this difficulty for the TXL language in a non-traditional way - by introducing the paradigms of use for each kind of problem directly. Rather than simply introducing TXL's language features, we present them in context as they are required in the paradigms for solving particular classes of problems such as parsing, restructuring, optimization, static analysis and interpretation. In essence this paper presents the beginnings of a "TXL Cookbook" of recipes for effectively using TXL, and to some extent other similar tools, in a range of common source processing and analysis problems. We begin with a short introduction to TXL concepts, then dive right in to some specific problems in parsing, restructuring and static analysis.

Keywords: source transformation, source analysis, TXL, coding paradigms.

1 Introduction

Source transformation systems and languages like DMS [2], Stratego [6], ASF + SDF [3,5], Rascal [20] and TXL [8] provide a very general, powerful set of capabilities for addressing a wide range of software analysis and migration problems. However, almost all successful practical applications of these systems have involved the original authors or long-time experts with the tools. New potential users usually find it difficult and frustrating to discover how they can leverage these systems to attack the particular problems they are facing.

This is not an accident. These systems are intentionally designed to be be very general, and their features and facilities are therefore at a level of abstraction far from the level at which new users understand their particular problems. What

R. Lämmel et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 27–91, 2011.

they are interested in is not what the general language features are, but rather how they should be used to solve problems of the kind they are facing. The real core of the solution for any particular problem is not in the language or system itself, but rather in the paradigm for using it to solve that kind of problem.

In this paper we aim to address this issue head-on, by explicitly outlining the paradigms for solving a representative set of parsing, transformation and analysis problems using the TXL source transformation language. In the long run we are aiming at a "TXL Cookbook", a set of recipes for applying TXL to the many different kinds of problems for which it is well suited. While the paradigms are couched here in terms of TXL, in many cases the same paradigms can be used with other source transformation systems as well.

In what follows we begin with a short introduction to the basics of TXL, just to set the context, and then dive directly into some representative problems from four different problem domains: parsing, restructuring, optimization, and static and dynamic analysis. With each specific problem we outline the basic paradigms used in concrete solutions written in TXL. Along the way we discuss TXL's philosophy and implementation as they influence the solutions. Although it covers many issues, this set of problems is by no means complete, and it is expected that the cookbook will grow in future to be more comprehensive.

Our example problems are set in the context of a small, simple imperative language designed for the purpose of demonstrating transformation and analysis toolsets. The language, TIL ("Tiny Imperative Language") [11], was designed by Jim Cordy and Eelco Visser as the intended basis of a set of benchmarks for source transformation and analysis systems.

It is not our intention to cover the features of the TXL language itself here - there are already other published papers on the language [8] and programming in it [9], and features of the language are covered in detail in the TXL reference manual [10]. Rather, here we concentrate on the paradigms for solving problems using it, assuming only a basic knowledge.

2 TXL Basics

TXL [8] is a programming language explicitly designed for authoring source transformation tasks of all kinds. It has been used in a wide range of applications involving millions of lines of code [7]. Unlike most other source transformation tools, TXL is completely self-contained - all aspects of the source transformation, including the scanner, parser, transformer and output pretty-printer are all written in TXL. Because they have no dependencies on external parsers, frameworks or libraries, TXL programs are easily ported across platforms.

2.1 The TXL Paradigm

The TXL paradigm is the usual for source transformation systems (Figure 1). Input text is scanned and parsed into an internal parse tree, pattern-replacement rewriting rules are applied to the parse tree to transform it to another, and then the transformed tree is unparsed to the new output text.

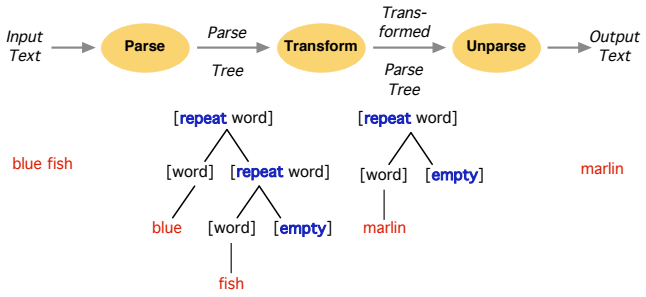

Fig. 1. The TXL Paradigm

Grammars and transformation rules are specified in the TXL language, which is efficiently implemented by the TXL Processor (Figure 2). The TXL processor is directly interpretive, with no parser generator or compile phase, which means there is little overhead and no maintenance barrier to running multiple transformations end-to-end. Thus TXL is well suited to test-driven development and rapid turnaround. But more importantly, transformations can be decomposed into independent steps with only a very small performance penalty, and thus most complex TXL transformations are architected as a sequence of successive approximations to the final result.

2.2 Anatomy of a TXL Program

A TXL program typically has three parts (Figure 3) : The *base grammar* defines the lexical forms (tokens) and the rooted set of syntactic forms (nonterminals or types) of the input language. Often the base grammar is kept in a separate file and included using an include statement. The *program* nonterminal is the root of the grammar, defining the form of the entire input. The optional *grammar overrides* extend or modify the syntactic forms of the grammar to allow output and intermediate forms of the transformation that are not part of the input language. Finally, the *rule set* defines the rooted set of transformation rules and functions to be applied to the input. The *main* rule or function is the root of the rule set, and is automatically applied to the entire input.

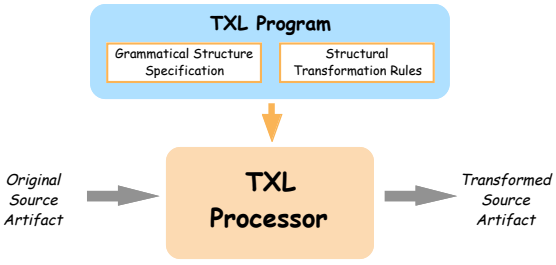

Fig. 2. The TXL Processor

Fig. 3. Parts of a TXL Program

While TXL programs are typically laid out as base grammar followed by overrides then rules, there is no ordering requirement and grammatical forms and rules can be mixed in the TXL program. To aid in readability, both grammars and rule sets are typically defined in topological order, starting from the *program* nonterminal and the *main* rule.

2.3 The Grammar: Specifying Lexical Forms

Lexical forms specify how the input text is partitioned into indivisible basic symbols (tokens or terminal symbols) of the input language. These form the basic types of the TXL program. The *tokens* statement gives regular expressions for each kind of token in the input language, for example, C hexadecimal integers:

```
tokens
    hexintegernumber    "0[xX][abcdefABCDEF\d]+"
end tokens
```

Tokens are referred to in the grammar using their name enclosed in square brackets (e.g., [hexintegernumber]). A set of default token forms are predefined in TXL, including C-style identifiers [id], integer and floating point numbers [number], string literals [stringlit], and character literals [charlit].

The *comments* statement specifies the commenting conventions of the input language, that is, sections of input source that are to be considered as commentary. For example, C commenting conventions can be defined as follows:

```
comments
    /*   */
    //
end comments
```

By default, comments are ignored (treated as white space) by TXL, but they can be treated as significant input tokens if desired. Most analysis tasks can ignore comments, but transformation tasks may want to preserve them.

The *keys* statement specifies that certain identifiers are to be treated as unique special symbols (and not as identifiers), that is, keywords of the input language. For example, the following could be used to specify the keywords of a subset of standard Pascal. The "end" keyword must be quoted (preceded by a single quote) to distinguish it from TXL's own end keyword. In general, TXL's own keywords and special symbols are the only things that need to be quoted in TXL, and other words and symbols simply represent themselves.

```
keys
    program procedure function
    repeat until for while do begin 'end
end keys
```

The *compounds* statement specifies character sequences to be treated as a single character, that is, compound tokens. Since "%" is TXL's end-of-line comment character, symbols containing percent signs must be quoted in TXL programs. Compounds are really just a shorthand for (unnamed) token definitions.

```
compounds
    :=  <=  >=  ->  <->  '%=          % note quoted "%"
end compounds
```

2.4 The Grammar: Specifying Syntactic Forms

Syntactic forms (nonterminal symbols or *types*) specify how sequences of input tokens are grouped into the structures of the input language. These form the structured types of the TXL program - In essence, each TXL program defines its own symbols and type system. Syntactic forms are specified using an (almost) unrestricted ambiguous context free grammar in extended BNF notation, where:

```
 X      literal terminal symbols (tokens) represent themselves
[X]     terminal (token) and nonterminal types appear in brackets
 |      or bar separates alternative syntactic forms
```

Each syntactic form is specified using a *define* statement. The special type [program] describes the structure of the entire input. For example, here is a simple precedence grammar for numeric expressions:

```
File "Expr.grm"

    define program    % goal symbol of input
        [expression]
    end define

    define expression
          [term]
        | [expression] + [term]
        | [expression] - [term]
    end define

    define term
          [primary]
        | [term] * [primary]
        | [term] / [primary]
    end define

    define primary
          [number]
        | ( [expression] )
    end define
```

Grammars are most efficient and natural in TXL when most user-oriented, using sequences in preference to recursion, and simpler forms rather than semantically distinguished cases. In general, *yacc*-style compiler "implementation" grammars should be avoided.

Sequences and optional items can be specified using an extended BNF-like sequence notation:

```
[repeat X]  or  [X*]        sequence of zero or more (X*)
[repeat X+] or  [X+]        sequence of one or more (X+)
[list X]    or  [X,]        comma-separated list of zero or more
[list X+]   or  [X,+]       comma-separated list one or more
[opt X]     or  [X?]        optional (zero or one)
```

For more natural patterns in transformation rules, these forms should always be used in preference to hand-coded recursion for specifying sequences in grammars, since TXL is optimized for handling them.

Formatting cues in defines specify how output text is to be formatted:

```
[NL]    newline  in unparsed output
[IN]    indent subsequent unparsed output by four spaces
[EX]    exdent subsequent unparsed output by four spaces
```

Formatting cues have no effect on input parsing and transformation patterns.

2.5 Input Parsing

Input is automatically tokenized and parsed according to the grammar. The entire input must be recognizable as the type [program], and the result is represented internally as a parse tree representing the structural understanding of the

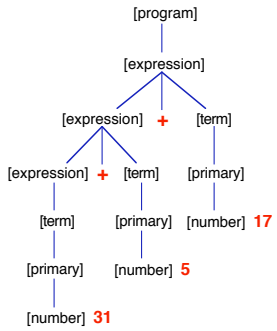

Fig. 4. Parse tree for the expression 31+5+17 according to the example grammar

input according to the grammar. Figure 4 shows the parse tree resulting from the input of the numeric expression "31+5+17" to a TXL program using the TXL grammar shown above.

All pattern matching and transformation operations in TXL rules and functions work on the parse tree. Since each TXL program defines its own grammar, it is important to remember that syntax errors in the input may indicate an incorrect grammar rather than a malformed input.

2.6 Base Grammars and Overrides

The base grammar for the syntax of the input language is normally kept in a separate grammar file which is rarely if ever changed, and is included in the TXL program using a TXL *include* statement. While the base grammar itself can serve for many purposes, since TXL is strongly typed in the type system of the grammar (that is, all trees manipulated or produced by a TXL transformation rule must be valid instances of their grammatical type, so that malformed results cannot be created), it is often necessary to add output or intermediate forms. Analysis and transformation tasks that require output or intermediate forms not explained by the grammar can add their own forms using *grammar overrides*.

Grammar overrides modify or extend the base grammar's lexical and syntactic forms by modifying existing forms using *redefine* statements. Redefine statements can completely replace an original form, for example, this redefine of [primary] in the example grammar above will modify it to allow identifiers and lists of expressions:

```
include "Expr.grm"          % the original example grammar

redefine primary
    [id]
  | [number]
  | ( [expression,+] )
end redefine
```

The semantics of such a redefine is that the original form is replaced by the new form in the grammar.

Grammar overrides can also extend the existing forms of a grammatical type, using the notation "..." to refer to the original definition of the nonterminal type. For example, this redefine will allow XML markup on any [statement] by extending the definition of [statement] to allow a marked-up statement.

```
include "C.grm"                    % the C language grammar

redefine statement
    ...                            % includes the original forms
  | <[id]> [statement] </[id]>     % adds the new XML markup form
end redefine
```

The redefine says that a statement can be any form it was before ("..."), or the new form. "..." is not an elision here, it is part of the TXL language syntax, meaning "whatever [statement] was before".

2.7 Transformation Rules

Once the input is parsed, the actual input to output source transformation is specified in TXL using a rooted set of transformation *rules*. Each transformation rule specifies a *target type* to be transformed, a *pattern* (an example of the particular instance of the type that we are interested in replacing) and a *replacement* (an example of the result we want when we find such an instance).

```
% replace every 1+1 expression with 2
rule addOnePlusOne
    replace [expression]      % target type to search for
        1 + 1                 % pattern to match
    by
        2                     % replacement to make
end rule
```

TXL rules are strongly typed - that is, the replacement must be of the same grammatical type as the pattern (that is, the target type). While this seems to preclude cross-language and cross-form transformations, as we shall see, because of grammar overrides this is not the case!

The pattern can be thought of as an actual source text example of the instances we want to replace, and when programming TXL one should think by example, not by parse tree. Patterns consist of a combination of tokens (input symbols, which represent themselves) and named variables (tagged nonterminal types, which match any instance of the type). For example, the TXL variable N1 in the pattern of the following rule will match any item of type [number] :

```
rule optimizeAddZero
    replace [expression]
        N1 [number] + 0
    by
        N1
end rule
```

When the pattern is matched, variable names are bound to the corresponding item of their type in the matched instance of the target type. Variables can be used in the replacement to copy their bound instance into the result, for example the item bound to N1 will be copied to the replacement of each [expression] matching the pattern of the rule above.

Similarly, the replacement can be thought of as a source text example of the desired result. Replacements consist of tokens (input symbols, which represent themselves) and references to bound variables (using the tag name of the variable from the pattern). References to bound variables in the replacement denote copying of the variable's bound instance into the result.

References to variables can be optionally further transformed by subrules (other transformation rules), which further transform (only) the copy of the variable's bound instance before it is copied into the result. Subrules are applied to a variable reference using postfix square bracket notation X[f], which in functional notation would be f(X). X[f][g] denotes functional composition of subrules - that is, g(f(X)). For example, this rule looks for instances of expressions (including subexpressions) consisting of a number plus a number, and resolves the addition by transforming copy of the first number using the [+] subrule to add the second number to it. ([+] is one of a large set of TXL built-in functions.)

```
rule resolveAdditions
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end rule
```

When a rule is applied to a variable, we say that the variable's copied value is the rule's *scope*. A rule application only transforms inside the scope it is applied to. The distinguished rule called *main* is automatically applied to the entire input as its scope - any other rules must be explicitly applied as subrules to have any effect. Often the main rule is a simple *function* to apply other rules:

```
function main
    replace [program]
        EntireInput [program]
    by
        EntireInput [resolveAdditions] [resolveSubtractions]
                    [resolveMultiplys] [resolveDivisions]
end function
```

2.8 Rules and Functions

TXL has two kinds of transformation rules, *rules* and *functions*, which are distinguished by whether they should transform only one (for functions) or many (for rules) occurrences of their pattern. By default, rules repeatedly search their scope for the first instance of their target type matching their pattern, transform it to yield a new scope, and then reapply to the entire new scope until no more matches are found. By default, functions do not search, but attempt to match only their entire scope to their pattern, transforming it if it matches. For example, this function will match only if the entire expression it is applied to is a number plus a number, and will not search for matching subexpressions:

```
function resolveEntireAdditionExpression
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end function
```

Searching functions, denoted by "replace *", search to find and transform the first occurrence of their pattern in their scope, but do not repeat. Searching functions are used when only one match is expected, or only the first match should be transformed.

```
function resolveFirstAdditionExpression
    replace * [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end function
```

2.9 Rule Parameters

Rules and functions may be passed parameters, which bind the values of variables in the applying rule to the formal parameters of the subrule. Parameters can be used to build transformed results out of many parts, or to pass global context into a transformation rule or function. In this example, the [resolveConstants] outer rule finds a Pascal named constant declaration, and passes both the name and the value to the subrule [replaceByValue] which replaces all references to the constant name in the following statements by its value. The constant declaration is then removed by [resolveConstants] in its replacement.

```
rule resolveConstants
    replace [statement*]
        const C [id] = V [primary];
        RestOfScope [statement*]
    by
        RestOfScope [replaceByValue C V]
end rule

rule replaceByValue ConstName [id] Value [primary]
    replace [primary]
        ConstName
    by
        Value
end rule
```

2.10 Patterns and Replacements

The example-like patterns and replacements in rules and functions are parsed using the grammar in the same way as the input, to make pattern-tree / replacement-tree pairs. Figure 5 shows an example of the pattern and replacement trees for the [resolveAdditions] example rule. While sometimes it is helpful to be aware of the tree representation of patterns, in general it is best to think at the source level in a by-example style when programming TXL.

Rules are implemented by searching the scope parse tree for tree pattern matches of the pattern tree, and replacing each matched instance with a corresponding instantiation of the replacement tree. In Figure 6 we can see the sequence of matches that the rule [resolveAdditions] will find in the parse tree for the input expression "36+5+17". It's important to note that the second match does not even exist in the original scope - it only comes to be after the first replacement. This underlines the semantics of TXL rules, which search for

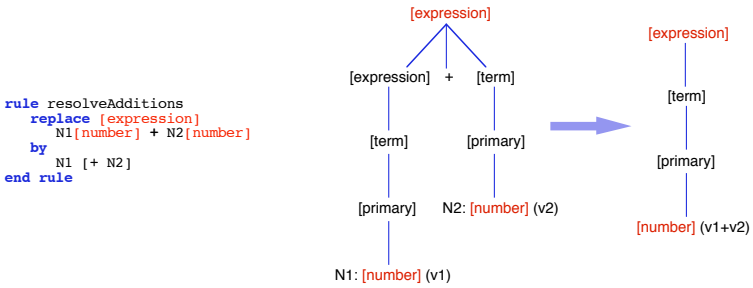

Fig. 5. Pattern and replacement trees for the [resolveAdditions] rule

one match at a time in their scope, and following a replacement, search the entire new scope for the next match.

Patterns may refer to a previously bound variable later in the same pattern (technically called *strong pattern matching*). This parameterizes the pattern with a copy of the bound variable, to specify that two parts of the matching instance must be the same in order to have a match. For example, the following rule's pattern matches only expressions consisting of the addition of two identical subexpressions (e.g., 1+1, 2*4+2*4, and (3-2*7)+(3-2*7)).

```
rule optimizeDoubles
    replace [expression]
        E [term] + E
    by
        2 * E
end rule
```

Patterns can also be parameterized by formal parameters of the rule, or other bound variables, to specify that matching instances must contain an identical copy of the variable's bound value at that point in the pattern. (We saw an example in the [replaceByValue] rule on the previous page.) A simple way to think about TXL variables is that references to a variable always mean a copy of its bound value, no matter what the context is.

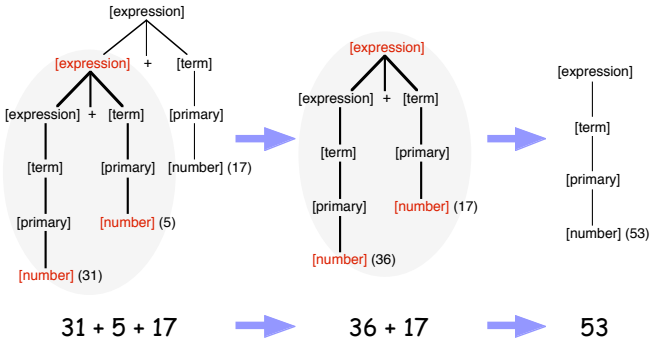

Fig. 6. Example application of the [resolveAdditions] rule

2.11 Deconstructors and Constructors

Patterns can be piecewise refined to more specific patterns using *deconstruct* clauses. Deconstructors specify that the deconstructed variable's bound value must match the given pattern - if not, the entire pattern match fails. Deconstructors act like functions - by default, the entire bound value must match the deconstructor's pattern, but "deconstruct *" (a *deep* deconstruct) searches within the bound value for a match. The following example demonstrates both kinds - a deep deconstruct matches the [if_condition] in the matched IfStatement, and the second deconstruct matches the entire IfCond only if it is exactly the word *false*.

```
rule optimizeFalseIfs
    replace [statement*]
        IfStatement [if_statement] ;
        RestOfStatements [statement*]
    deconstruct * [if_condition] IfStatement
        IfCond [if_condition]
    deconstruct IfCond
        'false
    by
        RestOfStatements
end rule
```

Pattern matches can also be constrained using *where* clauses, which allow for arbitrary matching conditions to be tested by subrules. The where clause succeeds only if its subrules find a match of their patterns. Like deconstructors, if a where clause fails, the entire pattern match fails. Here's an example use of where clauses to test that two sequential assignment statements do not interfere with each other (and thus the pair can be parallelized):

```
rule vectorizeScalarAssignments
    replace [statement*]
        V1 [variable] := E1 [expression];
        V2 [variable] := E2 [expression];
        RestOfScope [statement*]
    where not
        E2 [references V1]
    where not
        E1 [references V2]
    by
        < V1,V2 > := < E1,E2 > ;
        RestOfScope
end rule
```

While where clauses are more general, for efficiency reasons it is always better to use a deconstruct than a where clause when possible. Where clauses use a special kind of rule called a *condition rule*, for example [references] in the example above.

```
function references V [variable]
    deconstruct * [id] V
        Vid [id]
    match * [id]
        Vid
end function
```

Condition rules are different in that they have only a (possibly very complex) pattern, but no replacement - they simply succeed or fail to match their

pattern, but do not make any replacement. In this case, [references] looks inside its expression scope to see if there are any uses of the main identifier naming the variable it is passed.

Replacements can also be piecewise refined, to construct results from several independent pieces using *construct* clauses. Constructors allow partial results to be bound to new variables, allowing subrules to further transform them in the replacement or other constructors. In the example below, NewUnsortedSequence is constructed so that it can be further transformed by the subrule [sortFirstIntoPlace] in the replacement.

```
rule addToSortedSequence NewNum [number]
    replace [number*]
        OldSortedSequence [number*]
    construct NewUnsortedSequence [number*]
        NewNum OldSortedSequence
    by
        NewUnsortedSequence [sortFirstIntoPlace]
end rule
```

Even when constructors are not really required, constructing a complex replacement in well-named pieces can aid in readability of the rule.

This ends our basic introduction to TXL. We now move on to the real focus of this paper - the paradigms for solving real parsing, analysis and transformation problems using it. We begin by introducing TIL, the toy example language used as a platform for our example problems.

3 The TIL Chairmarks

TIL (Tiny Imperative Language) is a very small imperative language with assignments, conditionals, and loops, designed by Eelco Visser and James Cordy as a basis for small illustrative example transformations. All of the example applications in the TXL Cookbook work on TIL or extended dialects of it. Figure 7 shows two examples of basic TIL programs.

The TIL Chairmarks [12] are a small set of benchmark transformation and analysis tasks based on TIL. They are called "chairmarks" because they are too

File "factors.til"

```
// Find factors of a given number
var n;
write "Input n please";
read n;
write "The factors of n are";
var f;
f := 2;
while n != 1 do
    while (n / f) * f = n do
        write f;
        n := n / f;
    end;
    f := f + 1;
end;
```

File "multiples.til"

```
// First 10 multiples of numbers 1 through 9
for i := 1 to 9 do
for j := 1 to 10 do
    write i*j;
end; end;
```

Fig. 7. Example TIL programs

small to be called "benchmarks". These tasks form the basis of our cookbook, and the examples in this tutorial are TXL solutions to some of the problems posed in the Chairmarks. The TIL Chairmark problems are split into six categories: parsing, restructuring, optimization, static and dynamic analysis, and language implementation. In this tutorial we only have room for one or two specific problems from each category. In each case, a specific concrete problem is proposed and a TXL solution is demonstrated, introducing the corresponding TXL solution paradigms and additional language features as we go along. We begin with the most important category: parsing.

4 Parsing Problems

Every transformation or analysis task begins with the creation or selection of a TXL grammar for the source language. The form of the language grammar has a big influence on the ease of writing transformation rules. In these first problems, we explore the creation of language grammars, pretty-printers and syntactic extensions using the parsing aspect of TXL only, with no transformations. The grammars we create here will serve as the basis of the transformation and analysis problems in the following sections. In many cases, a TXL grammar is already available for the language you want to process on the TXL website.

It is important to remember that the purpose of the TXL grammar for an input language is not, in general, to serve as a syntax checker (unless of course that is what we are implementing). We can normally assume that inputs to be transformed are well-formed. This allows us to craft grammars that are simpler and more abstract than the true language grammar, for example allowing all statements uniformly everywhere in the language even if some are semantically valid only in certain contexts, such as the *return* statement in Pascal, which is valid only inside procedures. In general, such uniformity in the grammar makes analyzing and transforming the language forms easier. In the case of TIL, the language is simple enough that such simplification of the grammar is unnecessary.

4.1 Basic Parser / Syntax Checker

In this first problem, our purpose is only to create a grammar for the language we plan to process, in this case TIL. Figure 8 shows a basic TXL grammar (file "TIL.grm") for TIL. The main nonterminal of a TXL grammar must always be called [program], and there must be a nonterminal definition for [program] somewhere in the grammar. Implementing a parser and syntax checker using this grammar is straightforward, simply including the grammar in a TXL program that does nothing but match its input (Figure 9, file "TILparser.txl").

Paradigm. *The grammar is the parser.* TXL "grammars" are in some sense misnamed - they are not really grammars in the usual BNF specification sense, to be processed and analyzed by a parser generator such as SDF or Bison. Rather, a TXL grammar is a directly interpreted recursive descent parser, written in grammatical style. Thus in TXL the grammar is really a program for parsing

```
File "TIL.grm"

% TXL grammar for Tiny Imperative Language

% When pretty-printing, we parse and output
% comments, controlled by this pragma
% #pragma -comment

% Keywords of TIL, a reserved-word language
keys
    var if then else while do for
    read write 'end
end keys

% Compound tokens to be recognized
% as a single lexical unit
compounds
    :=  !=  <=  >=
end compounds

% Commenting convention for TIL -
% comments are ignored unless -comment is set
comments
    //
end comments

% Direct TXL encoding of the TIL grammar.
% [NL], [IN] and [EX] on the right are
% optional pretty-printing cues
define program
    [statement*]
end define

define statement
        [declaration]
    |   [assignment_statement]
    |   [if_statement]
    |   [while_statement]
    |   [for_statement]
    |   [read_statement]
    |   [write_statement]
    |   [comment_statement]
end define

% Untyped variables
define declaration
    'var [name] ;               [NL]
end define

define assignment_statement
    [name] := [expression] ;    [NL]
end define

define if_statement
    'if [expression] 'then      [IN][NL]
        [statement*]            [EX]
    [opt else_statement]
    'end ';                     [NL]
end define

define else_statement
    'else                       [IN][NL]
        [statement*]            [EX]
end define

define while_statement
    'while [expression] 'do     [IN][NL]
        [statement*]            [EX]
    'end ';                     [NL]
end define

define for_statement
    'for [name] := [expression]
            'to [expression] 'do      [IN][NL]
        [statement*]                  [EX]
    'end ';                           [NL]
end define

define read_statement
    'read [name] ;                    [NL]
end define

define write_statement
    'write [expression] ;             [NL]
end define

define comment_statement
    % Only ever present if -comment is set
    [NL] [comment] [NL]
end define

% Traditional priority expression grammar
define expression
        [comparison]
    |   [expression] [logop] [comparison]
end define

define logop
    'and | 'or
end define

define comparison
        [term]
    |   [comparison] [eqop] [term]
end define

define eqop
    = | != | > | < | >= | <=
end define

define term
        [factor]
    |   [term] [addop] [factor]
end define

define addop
    + | -
end define

define factor
        [primary]
    |   [factor] [mulop] [primary]
end define

define mulop
    * | /
end define

define primary
        [name]
    |   [literal]
    |   ( [expression] )
end define

define literal
        [integernumber]
    |   [stringlit]
end define

define name
    [id]
end define
```

Fig. 8. TIL grammar in TXL

File "TILparser.txl"

```
% TXL parser for Tiny Imperative Language

% All TXL parsers are automatically also pretty-printers if the
% grammar includes the optional formatting cues, as in this case

% Use the TIL grammar
include "TIL.grm"

% No need to do anything except recognize the input, since the grammar
% includes the output formatting cues

function main
    match [program]
        _ [program]
end function
```

Fig. 9. TIL parser and pretty-printer

the input language, where the input is source text and the output is a parse tree. When crafting TXL grammars, one needs to be aware of this fact, and think (at least partly) like a programmer rather than a language specifier.

The creation of a TXL grammar begins with the specification of the lexical forms (tokens, or terminal symbols) of the language, using TXL's regular expression pattern notation. Several common lexical forms are built in to TXL, notably [id], which matches C-style identifiers, [number], which matches C-style integer and float constants, [stringlit], which matches double-quoted C-style string literals, and [charlit], which matches single-quoted C-style character literals.

The TIL grammar uses only the default tokens [id], [integernumber] and [stringlit] as its terminal symbols, thus avoiding defining any token patterns of its own. ([integernumber] is a built-in refinement of [number] to non-floating point forms.) More commonly, it would be necessary to define at least some of the lexical forms of the input language explicitly using TXL *tokens* statements.

The TIL keywords are specified in the grammar using the *keys* statement, which tells TXL that the given words are reserved and not to be mistaken for identifiers. The *compounds* section tells us that the TIL symbols := and != are to be treated as single tokens, and the *comments* section tells TXL that TIL comments begin with // and go to the end of line. Comments are by default ignored and removed from the parsed input, and do not appear in the parse tree or output. However, they can be preserved (see section 4.2).

Paradigm. *Use sequences, not recursions.* The fact that TXL grammars are actually parsing programs has a strong influence on the expression of language forms. For example, in general it is better to express sequences of statements or expressions directly as sequences ([X*] or equivalently [repeat X]) rather than right- or left-recursive productions. This is both because the parser will be more efficient, and because the TXL pattern matching engine is optimized for searching sequences. Thus forms such as this one, which is often seen in BNF grammars, should be converted to sequences (in this case [statement*]) in TXL:

```
statements -> statement
           |  statements statement
```

Paradigm. *Join similar forms.* In order to avoid backtracking, multiple similar forms are typically joined together into one in TXL grammars. For example, when expressed in traditional BNF, the TIL grammar shows two forms for the if statement, with and without an else clause, as separate cases.

```
if_statement -> "if" expression "then"
                    statement*
                "end" ";"
             |  "if" expression "then"
                    statement*
                "else"
                    statement*
                "end" ";"
```

While we could have coded this directly into the TXL grammar, because it is directly interpreted, when instances of the second form were parsed, TXL would have to begin parsing the first form until it failed, then backtrack and start over trying the second form to match the input. When many such similar forms are in the grammar, this backtracking can become expensive, and it is better to avoid it by programming the grammar more efficiently in TXL. In this case, both forms are subsumed into one by separating and making the else clause optional in the TXL define for [if_statement] (Figure 8).

Paradigm. *Encode precedence and associativity directly in the grammar.* As in all direct top-down parsing methods, left-recursive nonterminal forms can be a particular problem and should in general be avoided. However, sometimes, as with left-associative operators, direct left-recursion is required, and TXL will recognize and optimize such direct left-recursions. An example of this can be seen in the expression grammar for TIL (Figure 8), which encodes precedence and associativity of operators directly in the grammar using a traditional precedence chain. Rather than separate precedence and associativity into separate disambiguation rules, TXL normally includes them in the grammar in this way.

Figure 9 shows a TXL program using the TIL grammar that simply parses input programs, and the result of running it on the example program "multiples.til" of Figure 7, using the command:

```
txl -xml multiples.til TILparser.txl
```

is shown in Figure 10. The "-xml" output shows the internal XML form of the parse tree of the input program.

4.2 Pretty Printing

The next problem we tackle is creating a pretty-printer for the input language, in this case TIL. Pretty-printing is a natural application of source transformation systems since they all have to create source output of some kind.

Paradigm. *Using formatting cues to control output format.* TXL is designed for pretty-printing, and output formatting is controlled by inserting formatting cues for indent [IN], exdent [EX] and new line [NL] into the grammar. These cues look like nonterminal symbols, but have no effect on input parsing. Their

```
linux% txl multiples.til TILparser.txl -xml

<program>
 <repeat statement>
  <statement><for_statement> for
    <name><id>i</id></name> :=
    <expression><primary><literal><integernumber>1</integernumber></literal></primary></expression> to
    <expression><primary><literal><integernumber>9</integernumber></literal></primary></expression> do
    <repeat statement>
     <statement><for_statement> for
       <name><id>j</id></name> :=
       <expression><primary><literal><integernumber>1</integernumber></literal></primary></expression> to
       <expression><primary><literal><integernumber>10</integernumber></literal></primary></expression> do
       <repeat statement>
        <statement><write_statement> write
          <expression>
           <expression><primary><name><id>i</id></name></primary></expression>
           <op>*</op>
           <expression><primary><name><id>j</id></name></primary></expression>
          </expression> ;
         </write_statement>
        </statement>
       </repeat statement> end ;
      </for_statement>
     </statement>
    </repeat statement> end ;
   </for_statement>
  </statement>
 </repeat statement>
</program>
```

Fig. 10. Example XML parse tree output of TIL parser

only role is to specify how output is to be formatted. For example, in the TIL grammar of Figure 8, the definition for [while_statement] uses [IN][NL] following the while clause, specifying that subsequent lines should be indented, and that a new line should begin following the clause. The [EX] after the statements in the body specifies that subsequent lines should no longer be indented, and the [NL] following the end of the loop specifies that a new line should begin following the while statement.

Paradigm. *Preserving comments in output.* By default TXL ignores comments specified using the *comments* section as shown in Figure 8, where TIL comments are specified as from // to end of line. In order to preserve comments in output, we must tell TXL that we wish to do that using the *-comment* command-line argument or the equivalent #pragma directive,

```
#pragma -comment
```

Once we have done that, comments become first-class tokens and the grammar must allow comments anywhere they may appear. For well-formed input code this is not difficult, but in general it is tricky and can require significant tuning. It is a weakness of TXL that it has no other good way to preserve comments.

In the TIL case, we have only end-of-line comments and we will assume that they are used only at the statement level - if we observe other cases, they can be added to the grammar. This is specified in the grammar with the statement

form [comment_statement], (which has no effect when *-comment* is off because no [comment] token will be available to parse). [comment_statement] is defined to put a new line before each comment, in order to separate it in the output:

```
define comment_statement
    [NL] [comment] [NL]
end define
```

Figure 11 shows the result of pretty-printing *multiples.til* using the parsing program of Figure 9.

4.3 Language Extensions

Language extensions, dialects and embedded DSLs are a common application of source transformation systems. The next problem involves implementing a number of syntactic extensions to the TIL grammar. Syntactic extension is one of the things TXL was explicitly designed for, and the paradigm is straightforward.

Figure 12 shows four small language extensions to TIL, the addition of begin-end statements, the addition of arrays, the addition of functions, and the addition of modules (i.e., anonymous or singleton classes). New grammatical forms, tokens and keywords are defined using the usual tokens, keys and define statements of TXL, as for example with the [begin_statement] definition in the begin-end extension of TIL and the addition of the *function* keyword in the function extension of TIL (both in Figure 12).

Paradigm. *Extension of grammatical forms.* New forms are integrated into the existing language grammar using redefinitions of existing forms, such as [statement] in the begin-end dialect of TIL. TXL's *redefine* statement is explicitly designed to support language modifications and extensions. In the begin-end extension we can see the use of redefine to add a new statement form to an existing language:

```
redefine statement
        ...                 % refers to all existing forms
    |   [begin_statement]    % add alternative for our new form
end redefine
```

```
linux%  cat Examples/multiples.til

    // Output first 10 multiples of numbers 1 through 9
    for i:=1 to 9 do for j:=1 to 10 do
        // Output each multiple
    write i*j; end; end;

linux%  txl -comment multiples.til TILparser.txl

    // Output first 10 multiples of numbers 1 through 9
    for i := 1 to 9 do
        for j := 1 to 10 do
            // Output each multiple
            write i * j;
        end;
    end;
```

Fig. 11. Example output of the TIL pretty-printer

File "TILbeginend.grm"

```
% TXL grammar overrides for begin-end
% extension of the Tiny Imperative Language

% Add begin-end statements
redefine statement
        ...                 % existing forms
    |   [begin_statement]   % adds new form
end redefine

define begin_statement
    'begin                  [IN][NL]
        [statement*]        [EX]
    'end                    [NL]
end define
```

File "TILfunctions.grm"

```
% TXL grammar overrides for functions
% extension of the Tiny Imperative Language

% Add functions using grammar overrides

redefine declaration
        ...                 % existing
    |   [function_definition] % new form
end redefine

redefine statement
        ...                 % existing
    |   [call_statement]
end redefine

keys
    'function
end keys

define function_definition
    'function [name] '( [name,] ')
                [opt colon_id]    [IN][NL]
        [statement*]              [EX]
    'end;                         [NL][NL]
end define

define call_statement
  [opt id_assign]
       [name] '( [expression,] ') ';  [NL]
end define

define colon_id
        ': [name]
end define

define id_assign
        [name] ':=
end define
```

File "TILarrays.grm"

```
% TXL grammar overrides for array
% extension of the Tiny Imperative Language

% Add arrays using grammar overrides
redefine declaration
      'var [name] [opt subscript] '; [NL]
   |  ...
end redefine

redefine primary
      [name] [opt subscript]
   |  ...
end redefine

redefine assignment_statement
    [name] [opt subscript] ':=
                    [expression] ';  [NL]
end redefine

define subscript
    '[ [expression] ']
end define
```

File "TILmodules.grm"

```
% TXL grammar overrides for module
% extension of the Tiny Imperative Language

% Add modules using grammar overrides
% Requires functions extension

redefine declaration
        ...                 % existing forms
    |   [module_definition]  % add new form
end redefine

keys
    'module  'public
end keys

define module_definition
    'module [name]           [IN] [NL]
        [statement*]         [EX]
    'end ;                   [NL] [NL]
end define

redefine function_definition
    [opt 'public] ...
end redefine
```

Fig. 12. TXL overrides for four dialects of TIL

Such a grammar modification is called a *grammar override* in TXL since it "overrides" or replaces the original definition with the new one. The "..." notation in this example is not an elision, it is an actual part of the TXL syntax. It refers to all of the previously defined alternative forms for the nonterminal type, in this case [statement], and is a shorthand for repeating them all in the redefine. It also makes the redefinition largely independent of the base grammar, so if the definition of [statement] in TIL changes, the language extension will not require any change, it will just inherit the new definition.

Because TXL grammars are actually directly interpreted programs for parsing, any ambiguities of the extension with existing forms are automatically resolved - the first defined alternative that matches an input will always be the one recognized. So even if the base language changes such that some or all of the language extension's forms are subsumed by the base language grammar, the extension will continue to be valid.

Paradigm. *Grammar overrides files.* Language extension and dialect files are normally stored in a separate grammar file. Such a grammar modification file is called a *grammar overrides* file, and is included in the TXL program following the include of the base grammar, so that it can refer to the base grammar's defined grammatical types:

```
include "TIL.grm"
include "TILbeginend.grm"
```

For example, while the TIL begin-end extension is independent of the grammatical forms of TIL other than the [statement] form it is extending, in the arrays extension of Figure 12, [expression] and [name] refer to existing grammatical types of TIL.

Paradigm. *Preferential ordering of grammatical forms.* In the begin-end extension the new form is listed as the last alternative, indicating a simple extension that adds to the existing language. When the new forms should be used in preference to existing ones, as in the arrays example, the new form is given as the first alternative and the existing alternatives are listed below, as in the [declaration] and [primary] redefinitions in the arrays extension of TIL:

```
redefine declaration
        'var [name] [opt subscript] ';  [NL]
    |   ...
end redefine

redefine primary
        [name] [opt subscript]
    |   ...
end redefine
```

Because grammatical types are interpreted directly as a parsing program, this means that any input that matches will be parsed as the new form, even if existing old forms would have matched it. So, for example, every var declaration in the arrays extension, including those without a subscript (e.g., "var x;") will be parsed with an optional subscript in the extended language, even though the base grammar for TIL already has a form for it. Similarly, every [name] reference which appears as a [primary] in the extension will be parsed with an [opt subscript] even though there is an existing [name] form for [primary].

Pretty-printing cues for extended forms are specified in redefine statements in the usual way, by adding [NL], [IN] and [EX] output formatting nonterminals to the definitions of the new forms, as in the new [declaration] form above.

Paradigm. *Replacement of grammatical forms.* Grammar type redefinitions can also completely replace the original form in the base grammar. For example, the

[assignment_statement] form of the arrays extension of TIL ignores the definition in the base grammar, and defines its own form which completely replaces it. This means that every occurrence of an [assignment_statement] in the extended language must match the form defined in the dialect.

```
redefine assignment_statement
        [name] [opt subscript] ':= [expression] ';  [NL]
end redefine
```

Paradigm. *Composition of dialects and extensions.* Language extensions and dialects can be composed and combined to create more sophisticated dialects. For example, the module (anonymous class) extension of TIL shown in Figure 12 is itself an extension of the function extension. Extensions are combined by including their grammars in the TXL program in dependency order, for example:

```
include "TIL.grm"
include "TILarrays.grm"
include "TILfunctions.grm"
include "TILmodules.grm"
```

Paradigm. *Modification of existing forms.* Extended forms need not be completely separate alternatives or replacements. When used directly in a redefine rather than as an alternative, the "..." notation still refers to all of the original forms of the nonterminal, modified by the additional syntax around it. For example, in the module extension of TIL (Figure 12), the [function_declaration] form is extended to have an optional *public* keyword preceding it. In this way the module dialect does not depend on what a [function_definition] looks like, only that it exists. Figure 13 shows an example of a program written in the modular TIL language dialect described by the composition of the arrays, functions and modules grammar overrides in Figure 12.

4.4 Robust Parsing

Robust parsing [1] is a general term for grammars and parsers that are insensitive to minor syntax errors and / or sections of code that are unexplained by the input language grammar. Robust parsing is very important in production program analysis and transformation systems since production software languages are often poorly documented, making it difficult to get an accurate grammar, because language compilers and interpreters often allow forms not officially in the language definition, and because languages often have dialects or local custom extensions in practice. For all of these reasons, it is important that analysis and transformation tools such as those implemented in TXL be able to handle exceptions to the grammar so that systems can be at least partially processed.

Paradigm. *Fall-through forms.* The basic paradigm for robust parsing in TXL is to explicitly allow for unexplained input as a dialect of the grammar that adds a last, uninterpreted, alternative to the grammatical forms for which there may be such unofficial or custom variants. For example, we may want to allow for statements that are not in the official grammar by making them the last alternative when we are expecting a statement.

```
File "primes.mtil"

// this program determines the primes up to maxprimes using the sieve method
var maxprimes;
var maxfactor;
maxprimes := 100;
maxfactor := 50;        // maxprimes div 2

var prime;
var notprime;
prime := 1;
notprime := 0;

module flags
   var flagvector [maxprimes];

   public function flagset (f, tf)
       flagvector [f] := tf;
    end;

  public function flagget (f) : tf
      tf := flagvector [f];
   end;
end;

// everything begins as prime
var i;
i := 1;
while i <= maxprimes do
    flagset (i, prime);
    i := i + 1;
end;

. . .
```

Fig. 13. Part of an example program in the TIL arrays, functions, modules dialect

Figure 14 shows the grammar overrides for a dialect of TIL that allows for unexplained statement forms. The key idea is that all statements in TIL end with a semicolon - so if we have a form ending in a semicolon that does not match any of the known forms, it must be an unknown statement form. Because alternatives in TXL grammars are tried in order, we can encode this by adding the unknown case as the last form for [statement]:

```
redefine statement
        ...                     % existing forms for [statement]
    |   [unknown_statement]     % fall-through if not recognized
end redefine
```

Paradigm. *Uninterpreted forms.* When parsing, all other alternatives are tested, after which we fall through to the [unknown_statement] form. [unknown_ statement] is any sequence of input items that are not semicolons [not_semicolon*], ended with a semicolon. This ensures that we don't accidentally accept uninterpreted input over a statement boundary.

The [not_semicolon] nonterminal type is the key to flushing uninterpreted input, and uses a standard TXL paradigm for flushing input, [token_or_key]. [token] is a special TXL built-in type that matches any input token that is not a keyword of the grammar, and [key] is a special built-in type that matches any keyword. Thus the following definition describes a type that will accept any single item from the input:

```
define token_or_key
        [token]         % any input token that is not a keyword
    |   [key]           % any keyword
end define
```

Paradigm. *Guarded forms.* In the robust TIL dialect, we must be careful not to throw away a semicolon, and thus we have guarded [token_or_key] with a nonterminal guard. In the [not_semicolon] definition, [not ';] indicates that if the next input token is a semicolon, then we should not accept it as a [token_or_key].

```
define not_semicolon
        [not ';] [token_or_key]    % any item except semicolon
end define
```

[not X] is a generalized grammatical guard that can be used to limit what can be matched by the form following it to those inputs that cannot be recognized as an [X], which can be any grammatical type. Its semantics are simple: if an [X] can be parsed at the current point in the input, then the following form is not tested, otherwise it is. In either case, [not X] does not itself consume any input.

```
File "TILrobust.grm"

    % TXL grammar overrides for robust parsing extension of Tiny Imperative Language

    % Allow for unrecognized statement forms
    redefine statement
            ...                         % refers to all existing forms for [statement]
        |   [unknown_statement]         % add fall-through if we don't recognize a statement
    end redefine

    define unknown_statement
            [not_semicolon*] ;  [NL]
    end define

    define not_semicolon
            [not ';] [token_or_key]     % any input item that is not a semicolon
    end define

    define token_or_key
            [token]                     % any input token that is not a keyword
        |   [key]                       % any keyword
    end define
```

Fig. 14. TXL overrides for robust statement parsing in TIL

4.5 Island Grammars

Island grammars [14,19] address a related problem to robust parsing, the problem of embedded code we wish to process in a sea of other text we don't want to process. For example, we may want to analyze only the embedded C code examples in the chapters of a textbook or a set of HTML pages, or only the EXEC SQL blocks in a large set of Cobol programs.

The basic strategy for island grammars in TXL is to invert the robust parsing strategy - we treat the input as a sequence of meaningful things ("islands") and unmeaningful things ('water") (Figure 15). The meaningful things are, for example, TIL programs, and the unmeaningful things are any sequence of input items not beginning with a TIL program.

File "Islands.grm"

```
% Generic grammar for parsing documents
% with embedded islands

% The input is a sequence of interesting
% islands and uninteresting water
redefine program
    [island_or_water*]
end redefine

define island_or_water
        [island]
    |   [water]
end define

% Water is any input that is not an island
define water
    [not_island+]
end define

define not_island
    % any item that does not begin an island
    [not island] [token_or_key]
end define

define token_or_key
        [token]    % any token not a keyword
    |   [key]      % any keyword
end define
```

File "TILislands.txl"

```
% TXL program for parsing documents
% with embedded TIL programs

% Begin with the TIL grammar
include "TIL.grm"

% And the generic island grammar
include "Islands.grm"

% In this case the islands are TIL programs
define island
    [til_program]
end define

define til_program
    [statement+]
end define

% We can now target rules at embedded TIL
% [island]s. But in this case, we just
% delete the non-TIL, to yield code only
rule main
    replace [island_or_water*]
        Water [water]
        Rest [island_or_water*]
    by
        Rest
end rule
```

Fig. 15. TXL generic island grammar (left), and an island parser for embedded TIL programs based on it (right)

Paradigm. *Preferential island parsing.* Figure 15 shows a generic TXL grammar for implementing island grammars to parse documents such as this one, recognizing the embedded islands (such as TIL programs) and ignoring the rest of the text (such as this paragraph). As usual, the trick is that the first alternative [island] is preferred, and the second [water] is tried only if the first fails. Parameterized generic grammars such as this one are frequently used in TXL to encode reusable parsing paradigms such as island grammars.

The generic island grammar is used by defining [island], the interesting form, in the TXL program that includes the generic grammar. The second half of Figure 15 is a TXL program that uses the generic island grammar to make an island grammar for embedded TIL programs in documents such as this tutorial. [island] is defined as [til_program], which uses the included TIL grammar's [statement] form. The analysis or transformation rules can then target the [island] forms only, ignoring the uninterpreted water. In this case, the program simply replaces all occurrences of [water] by the empty sequence, leaving only the embedded TIL programs in the output.

4.6 Agile Parsing

Agile parsing [13] refers to the use of grammar tuning on an individual analysis or transformation task basis. By using the parser to change the parse to better isolate the parts of the program of interest or make them more amenable to the particular transformation or analysis, we can greatly simplify the rules necessary to perform the task.

Paradigm. *Transformation-specific forms.* Agile parsing is implemented in TXL using grammar overrides (redefines) in exactly the same way as we have done for language extensions and dialects. In essence, we create a special dialect grammar for the language in support of the particular task.

The remainder of this paper consists of a sampling of example problems in various applications of source transformation, highlighting the TXL paradigms that are used in each solution.

5 Restructuring Problems

Once we have crafted grammars for our input languages, we can begin using them to support the real work - the transformation and analysis tasks that support software understanding, maintenance, renovation, migration and evolution. The flexibility of the TXL parser is a key to its application in many domains - for example, we exploit agile parsing in many solutions. But the real work is in the transformation and analysis rules.

In the remaining problems from the TXL Cookbook, we concentrate on source code transformation and analysis problems in three categories: restructuring problems, optimization problems, and static and dynamic analysis problems. In each category, we will look at a set of small but real challenges, each couched in terms of TIL and its extensions. We only have space for a few representative examples in each category, chosen not because they are the most useful, but because they introduce new recipes and paradigms.

As we have seen, a TXL "grammar" is not really a grammar - rather it is a functional program for parsing the input, which gives us direct control over the parse, yielding both flexibility and generality. Similarly, a TXL transformation "rule set" is not really a term rewriting system - rather, the rules form a functional program for transforming the input, with similar direct control over tree traversal and strategy, again yielding flexibility and generality.

We begin with problems in basic program restructuring, the heart of applications in refactoring and code improvement. As with our parsing examples, all of our example problems are based on the Tiny Imperative Language (TIL) and its extensions. We will use the grammars and parsing techniques we developed in Section 4 to support all our solutions.

Paradigm. *Programmed functional control.* Transformations and analyses are coded in TXL using *rules* and *functions*. The basic difference between the two is that rules repeatedly search for and transform instances of their pattern until no more can be found, whereas functions transform exactly one instance of their pattern. TXL is a functional language, and the transformation is driven by the application of one rule or function, the *main* rule, to the parse tree of the entire input. All other rules and functions must be explicitly invoked, either in the main rule or in other rules invoked by it.

In contrast to pure term rewriting systems, this functional style gives the programmer fine-grained programmed control over the application of transformation rules on an invocation-by-invocation basis, and tree traversals and strategies can

be customized to each task. Of course, the downside of this flexibility and control is that you *must* do so, the price we pay in TXL for detailed programmability. As we shall see, in practice the common traversals and strategies are simply TXL coding paradigms, which we can learn quickly and reuse as need be. It is these functional paradigms that we will be exploiting in our solutions.

Paradigm. *Transformation scopes.* The result of a TXL rule or function invocation is a transformed copy of the *scope* (parse tree) it is directly applied to. In TXL, scopes of application are explicitly programmed - rules are not global, but transform only the subtree they are applied to. The result of a rule application is (semantically) a completely separate copy from the scope itself - the original TXL variable bound to the scope is unchanged by a rule invocation on it, and retains its original value (parse tree), as in all functional languages. For example, if the TXL variable X is bound to the [number] 1, the rule invocation X [+ 1] yields 2, but does not change X, which retains its original value, 1.

5.1 Feature Reduction

Applications in code analysis and transformation often begin by normalizing the code to reduce the number of features in the code to be analyzed in order to expose basic semantics and reduce the number of cases to analyze. Figure 16 is a simple example of such a feature reduction transformation, the elimination of TIL *for* loops by translation to an equivalent *while*. The transformation has only one rule, [main], which searches for sequences of [statement] beginning with a for statement and replaces the for with an equivalent while statement. While small, this simple example introduces us to a number of TXL paradigms.

It may surprise you to see that the rule is targeted at the type [statement*], a sequence of statements, rather than just [statement], since it is a single *for* statement that we are replacing. The reason for this is that we need to replace the for loop with not one statement but several - the initialization of the iteration variable, the declaration and computation of the upper limit, and the while loop itself. If we had tried to replace a single [statement] with this sequence, we would get a syntax error in the replacement, because TXL rules are constrained to preserve grammatical type in order to guarantee a well-formed result. A sequence of statements [statement*] is not an instance of the type [statement], and thus a replacement of several statements would violate the type constraint.

Paradigm. *Raising the scope of application.* This situation is an example of a general paradigm in TXL - transforming a pattern that is further up the parse tree than what we really want to match, in order to be able to create a result that is significantly different. The saying in TXL is: if you can't create the replacement you want, target further up the tree. In this specific case, we need to create several [statement]s out of one - so we must target the statement sequence [statement*] of which the for statement is a part.

Note that the statements following the for are also captured in the pattern (*MoreStatements*) and preserved in the result. This is a part of the paradigm - if we had not allowed for these, the pattern could match only sequences containing

File "TILfortowhile.txl"

```
% Convert Tiny Imperative Language "for" statements to "while" form

% Based on the TIL grammar
include "TIL.grm"

% Preserve comments in output
#pragma -comment

% Rule to convert every "for" statement
rule main
    % Capture each "for" statement, in its statement sequence context
    % so that we can replace it with multiple statements
    replace [statement*]
        'for Id [id] := Expn1 [expression] 'to Expn2 [expression] 'do
            Statements [statement*]
        'end;
        MoreStatements [statement*]

    % Need a unique new identifier for the upper bound
    construct UpperId [id]
        Id [_ 'upper] [!]

    % Construct the iterator
    construct IterateStatement [statement]
        Id := Id + 1;

    % Replace the whole thing
    by
        'var Id;
        Id := Expn1;
        'var UpperId;
        UpperId := (Expn2) + 1;
        'while Id - UpperId 'do
            Statements [. IterateStatement]
        'end;
        MoreStatements
end rule
```

Fig. 16. TXL transformation to convert for statements to while statements

exactly one statement - that is, the last statement of a sequence. There is no cost to copying these from the pattern to the result, since like many functional languages TXL optimizes flow-through copies.

Paradigm. *Explicit patterns.* The pattern for the for loop is fully explicated, that is, it matches all of its parts right away rather than just a [for_statement] which we could then take apart. Similarly, the replacement contains all the parts of the result explicitly rather than constructing a [while_statement] and replacing it whole. This example-like way of expressing rules is a TXL style - making the pattern and replacement show as much as possible of the form of the actual intended pattern and result target code rather than the constructed terms.

TXL uses the same parser (i.e., the TXL grammar you specify) to parse patterns and replacements in rules as it does for input. Thus it constructs all of the intermediate terms for you. This means that there is no cost to explicating details in a pattern, and it would be no more efficient to have a pattern searching for a [for_statement] only than for the entire pattern we have coded in the [main] rule, because the parsed pattern is in fact a [for_statement] anyway.

Given this preference for an example-like style, it may also surprise you to see that the iteration statement (*IterateStatement*) is separately constructed and appended [.] to the sequence of statements in the body of the loop rather than appearing in the replacement directly. The reason for this is the definition of sequence in TXL - the sequence type [X*] has a recursive definition, deriving [X] [X*] or [empty]. Thus although a statement at the head of a sequence ([statement] [statement*] as in the pattern of this rule) is a valid [statement*], a statement at the end, [statement*] [statement], is not. Therefore the TXL [.] (sequence append) built-in function is provided to allow for this, and the rule uses it to append the new statement to those in the loop body.

It may also surprise you to see that the literal identifiers and keywords in both the pattern and the replacement have been quoted using a single quote ' in all cases. While this is not necessary (except for the TXL keyword "end"), TXL programmers often choose to quote literal identifiers to remind the reader that they are not TXL variable references but part of the output text.

Paradigm. *Generating unique new identifiers.* The rule uses two built-in functions, [_] and [!], to generate a unique new identifier for the introduced upper bound variable. In the construct of *UpperId*, a new identifier is constructed from the original for iteration variable name *Id*, to which the literal identifier "upper" is appended with underscore using the [_] built-in function to form a new identifier (for example, if Id is "i", then we have "i_upper"). The new identifier is then made globally unique using the unique built-in function [!], which appends a number to it to create a new identifier unused anywhere else in the input (for example, "i_upper27").

This first example did its transformations in place - let's look at one that moves things around a bit.

5.2 Declarations-to-Global

One of the standard challenges for transformation tools is the ability to move things about, and in particular to make transformations at an outer level that depend on things deeply embedded in an inner level and vice-versa. In the next two examples, we will look at each of these kinds of problems in turn.

In the first problem, we are simply going to move all declarations in the TIL program to the global scope. Even though TIL declarations seem to be able to appear anywhere according to the TIL grammar, their meaning is apparently global, since no scope rules are defined. In this transformation, we make the true meaning of embedded declarations explicit by promoting all declarations to one global list at the beginning of the program.

The simplest solution to this problem (Figure 17) uses two common paradigms of TXL, *type extraction* and *type filtering*. The basic strategy is shown in the main rule, which has three steps: construct a copy of all the declarations in the program as a sequence, construct a copy of the program with all declarations removed, and concatenate the one to the other to form the result.

```
File "TILtoglobal.txl"

% Make all TIL declarations global

% Based on the TIL base grammar
include "TIL.grm"

% Preserve comments in output
#pragma -comment

% The main rule - in this case a function,
% applies only once

function main
    replace [program]
        Program [statement*]

    % Extract all statements,
    % then filter for declarations only
    construct Declarations [statement*]
        _ [^ Program] [removeNonDeclarations]

    % Make a copy of the program
    % with all declarations removed
    construct ProgramSansDeclarations [statement*]
        Program [removeDeclarations]

    % The result consists of the declarations
    % concatenated with the non-declarations
    by
        Declarations [. ProgramSansDeclarations]
end function

rule removeDeclarations
    % Rule to remove every declaration
    % at every level from statements
    replace [statement*]
        Declaration [declaration]
        FollowingStatements [statement*]
    by
        FollowingStatements
end rule

rule removeNonDeclarations
    % Rule to remove all statements that
    % are not declarations from statements
    replace [statement*]
        NonDeclaration [statement]
        FollowingStatements [statement*]

    % Check the statement isn't a declaration
    deconstruct not NonDeclaration
        _ [declaration]

    % If so, take it out
    by
        FollowingStatements
end rule
```

Fig. 17. TXL transformation to move all declarations to the global scope

Extracting all the declarations from the program is done in two steps, using the extract [^] built-in rule to get a sequence of all the statements of the program, and then removing all those that are not declarations.

```
construct Declarations [statement*]
        _ [^ Program] [removeNonDeclarations]
```

Paradigm. *Extracting all instances of a type.* The extract built-in function [^] is applied to a scope of type [T*] for any type [T], and takes as parameter a bound variable V of any type. The rule constructs a sequence containing a copy of every occurrence of an item of type [T] in V and replaces its scope with the result. In our case, a sequence containing a copy of every [statement] in the program is constructed. Extract ignores its original scope, so it is normally empty to begin with. In this case, we have used the empty variable "_", a special TXL variable denoting an empty item, as the scope of the rule. This is the usual way that extract is used.

Paradigm. *Filtering all instances of a type.* The second step in this construct uses the subrule [removeNonDeclarations] to remove all non-declarations from the constructed sequence of all statements. (The constructor could have extracted all [declaration]s directly, but this would cause problems later when we tried to concatenate them to the beginning of the program.) The subrule uses a common filtering paradigm in TXL, looking for any occurrence of a sequence of statements beginning with a statement that is not a declaration, and replacing

it with the sequence without the beginning statement. The rule continues until it can find no remaining instances in its scope.

Paradigm. *Negative patterns.* Determining that a statement is not a declaration involves another common paradigm in TXL - a negated deconstructor. A normal deconstructor simply matches a bound variable to a pattern for example:

```
deconstruct Statement
       Assignment [assignment_statement]
```

which succeeds and binds Assignment if the [statement] to which Statement is bound consists entirely of an assignment statement.

In this case, however, we are interested in statements that are *not* a [declaration], so we use *deconstruct not* to say that our match succeeds only if the deconstructor fails (that is, the [statement] bound to NonDeclaration is not a [declaration]. Although it has a pattern, a *deconstruct not* does not bind any pattern variables, since to succeed it must not match its pattern. Thus any variable names in the negated deconstructor's pattern are irrelevant, and in this case we have explicitly indicated that by using the anonymous name "_" in the pattern.

```
deconstruct not NonDeclaration
        _ [declaration]
```

The same filtering paradigm is used in the second constructor of the main rule to remove all declarations from the copy of the program used in the result of the rule. This general removal paradigm can be used with any simple, complex or guarded pattern to remove items matching any criterion from a scope.

Finally, the replacement of the rule simply appends the copy of the program without declarations to the extracted declarations, yielding a result with all declarations at the beginning of the program.

5.3 Declarations-to-Local

The other half of the movement challenge is the ability to make transformations on an inner level that depend on things from an outer level. One such problem is localization, in which things at an outer level are to be gathered and moved to an inner level. It can be used to support clustering of related methods, refactoring to infer methods, creation of inferred classes, and so on.

In this next problem, we assume that TIL is a scoped language rather than unscoped. The idea is to find all declarations of variables that are artificially global, and localize them as much as possible to the deepest inner scope in which they are used. In some sense it is the inverse of the previous problem.

Figure 18 shows a TXL solution to this problem. The main rule for this transformation uses two steps - "immediatize" and "localize". The [immediatizeDeclarations] rule moves declarations as far down in their scope as possible, to immediately before the first statement that uses their declared variable.

For example, if we have the scope shown on the left below (a), then the first step, [immediatizeDeclarations], will yield the intermediate result in the middle (b). The second step, [localizeDeclarations], then looks for compound statements

```
File "TILtolocal.txl"

% Move all declarations in a TIL program
% to their most local location

% Based on the TIL base grammar
include "TIL.grm"

% Preserve comments
#pragma -comment

% Transformation to move all declarations
% to their most local location -
% immediately before their first use,
% in the innermost block they can be.

rule main
    % This rule's pattern matches its result,
    % so it has no natural termination point
    replace [program]
        Program [program]

    % So we add an explicit fixed-point
    % guard - after each application of the
    % two transformations, we check to see
    % that something more was actually done
    construct NewProgram [program]
        Program [immediatizeDeclarations]
                [localizeDeclarations]
    deconstruct not NewProgram
            Program
    by
        NewProgram
end rule

rule immediatizeDeclarations
    % Move declarations past statements
    % that don't depend on them.
    % Use a one pass ($) traversal
    replace $ [statement*]
        'var V [id];
        Statement [statement]
        MoreStatements [statement*]

    % We can move the declaration past a
    % statement if the statement does not
    % refer to the declared variable
    deconstruct not * [id] Statement
            V
    by
        Statement
        'var V;
        MoreStatements
end rule

rule localizeDeclarations
    % Move declarations outside a structured
    % statement inside if following statements
    % do not depend on the declared variable.
    % Again, use a one pass ($) traversal
    replace $ [statement*]
        Declaration [declaration]
        CompoundStatement [statement]
        MoreStatements [statement*]

    % Check that it is some kind of compound
    % statement (one with a statement list inside)
    deconstruct * [statement*] CompoundStatement
            _ [statement*]

    % Check that the following statements
    % don't depend on the declaration
    deconstruct * [id] Declaration
            V [id]
    deconstruct not * [id] MoreStatements
            V

    % Alright, we can move it in.
    % Another solution might use agile parsing
    % to abstract all these similar cases into one
    by
        CompoundStatement
            [injectDeclarationWhile Declaration]
            [injectDeclarationFor Declaration]
            [injectDeclarationIfThen Declaration]
            [injectDeclarationIfElse Declaration]
        MoreStatements
end rule

function injectDeclarationWhile
                    Declaration [declaration]
    % There is no legal way that the while
    % Expn can depend on the declaration,
    % since there are no assignments between
    % the declaration and the Expn
    replace [statement]
        'while Expn [expression] 'do
            Statements [statement*]
        'end;
    by
        'while Expn 'do
            Declaration
            Statements
        'end;
end function

  . . .    (other injection rules similar)
```

Fig. 18. TXL transformation to localize all declarations

into which an immediately preceding declaration can be moved, and moves the declaration ("var x;" in the example) inside, yielding the result (c) on the right.

```
var y;
var x;
read y;
y := y + 6;
if y > 10 then
    x := y * 2;
    write x;
end;
```

(a)

```
var y;
read y;
y := y + 6;
var x;
if y > 10 then
    x := y * 2;
    write x;
end;
```

(b)

```
var y;
read y;
y := y + 6;
if y > 10 then
    var x;
    x := y * 2;
    write x;
end;
```

(c)

Paradigm. *Transformation to a fixed point.* Because declarations may be more than one level too global, the process must be repeated on the result until a fixed point is reached. This is encoded in the main rule, which is an instance of the standard fixed-point paradigm for TXL rules.

Although its only purpose is to invoke the other rules, [main] is a rule rather than a function because we expect it to continue to look for more opportunities to transform its result after each application. But unless we check that something was actually done on each application, the rule will never halt since its replacement NewProgram is a [program] and therefore matches its pattern. To terminate the rule, we use a deconstructor as an explicit fixed-point test:

```
deconstruct not NewProgram
        Program
```

The deconstructor simply tests whether the set of rules has changed anything on each repeated application, that is, if the NewProgram is exactly the same as the matched Program. If nothing has changed, we are by definition at a fixed point. This rule is a complete generic paradigm for fixed-point application of any rule set - only the set of rules applied in the constructor changes.

Paradigm. *Dependency sorting.* The rule [immediatizeDeclarations] works by iteratively moving declarations over statements that do not depend on them. In essence, this is a dependency sort of the code. The rule continues to move declarations down until every declaration is immediately before the first statement that uses its declared variable. (This could be done more efficiently by moving declarations directly, but our purpose here is to demonstrate as many paradigms as possible in the clearest and simplest way.) Dependency sorting in this way is a common paradigm in TXL, and we will see it again in other solutions.

Paradigm. *Deep pattern match.* The dependency test uses another common paradigm in TXL - a *deep deconstruct*. This is similar to the negated deconstruct used in the previous problem, but this time we are not just interested in whether Statement does not match something, we are interested in whether it does not *contain* something. Deep deconstructs test for containment by specifying the type of the pattern they are looking for inside the bound variable, and a pattern of that type to find. In this case, we are looking to see if there is an instance of an identifier (type [id]) exactly like the declared one (bound to V).

Paradigm. *One pass rules.* The [immediatizeDeclarations] rule also demonstrates another paradigm of TXL - the "one-pass" rule. If there are two declarations in a row, this rule will contually move them over one another, never coming to a fixed point. For this reason, the rule is marked as one-pass using *replace $*. This means that the scope should be searched in linear fashion for instances of the pattern, and replacements should not be directly reexamined for instances of the pattern. In this case, if we move a declaration over another, we don't try to move the other over it again because we move on to the next sequence of statements in one-pass rather than recursive fashion.

The second rule in this transformation, [localizeDeclarations], looks for instances of a declaration that has been moved to immediately before a compound

statement (such as if, while, for) and checks to see whether it can be moved inside the statement's scope. The rule uses all of the paradigms outlined above - it is one-pass (*replace $*) so that it does not try the same case twice, and it uses deep pattern matching both to get the declared identifier V from the Declaration and to check that the following statements MoreStatements do not depend on the declaration we want to move inside, by searching for uses of V in them.

A new use of deconstruct in this rule is the deep deconstruct of CompoundStatement, which is simply used to check that we actually have an inner scope in the statement in which to move the declaration.

Paradigm. *Multiple transformation cases.* The replacement of this rule demonstrates another paradigm, the programming of cases in TXL. There are several different compound statements into which we can move the declaration: while statements, for statements, then clauses and else clauses. Each one is slightly different, and so they have different patterns and replacements. In TXL such multiple cases use one function for each case, all applied to the same scope.

In essence this is the paradigm for case selection or if-then-else in TXL - application of one function for each case. Only one of the functions will match any particular instance of CompoundStatement, and the others that do not match will leave the scope untouched. TXL functions and rules are *total*, that is, they have a defined result, the identify transformation, when they do not match.

Paradigm. *Context-dependent transformation rules.* In each case, the Declaration to be inserted into the CompoundStatement is passed into the function for the case using a rule parameter. Rule parameters allow us to carry context from outer scopes into rules that transform inner scopes, and this is the paradigm for context-dependent transformation in TXL. In this case we pass the Declaration from the outer scope into the rule that transforms the inner scope.

The context carried in can be arbitrarily large or complex - for example, if the inner transformation rule wanted to change small things inside its scope but depended on global things, we could pass a copy of the entire program into the rule. Outer context can also be passed arbitrarily deeply into subrules, so if a small change deeply inside a sub-sub-subrule depended on something in the outer scope, we could pass a copy all the way in.

5.4 Goto Elimination

The flagship of all restructuring problems is goto elimination - the inference of structured code such as while loops and nested if-then-else from spaghetti-coded goto statements in legacy languages such as Cobol. In this example we imagine a dialect of TIL that has goto statements, and infer equivalent while statements where possible. Figure 19 gives the grammar for a dialect of TIL that adds goto statements and labels, so that we can write programs like the one shown on the left below (a). Our goal is to recognize and transform loop-equivalent goto structures into their while loop form, like the result (b) on the right.

```
// Factor an input number
var n;
var f;
write "Input n please";
read n;
write "The factors of n are";
f := 2;
// Outer loop over potential factors
factors:
    if  n = 1 then
        goto endfactors;
    end;
// Inner loop over multiple instances
// of same factor
multiples:
    if (n / f) * f != n then
        goto endmultiples;
    end;
    write f;
    n := n / f;
    goto multiples;
endmultiples:
    f := f + 1;
    goto factors;
endfactors:
```

(a)

```
// Factor an input number
var n;
var f;
write "Input n please";
read n;
write "The factors of n are";
f := 2;
// Outer loop over potential factors
while n != 1 do
    // Inner loop over multiple instances
    // of same factor
    while (n / f) * f = n do
        write f;
        n := n / f;
   end;
   f := f + 1;
end;
```

(b)

An example TXL solution to the problem of recognizing and transforming while-equivalent goto structures is shown in Figure 19. The basic strategy is to catalogue the patterns of use we observe, encode them as patterns, and use one rule per pattern to replace them with their equivalent loop structures. In practice we would first run a goto normalization (feature reduction) transformation to reduce the number of cases.

The program presently recognizes two cases: "forward' while structures, which begin with an if statement guarding a goto and end with a goto back to the if statement, and "backward" whiles, which begin with a labelled statement and end with an if statement guarding a goto branching back to it.

By now most of the TXL code will be looking pretty familiar. However, this example has two new paradigms to teach us. The first is the match of the pattern in the rule [transformForwardWhile]. Ideally, we are looking for a pattern of the form:

```
replace [statement*]
    L0 [label] ':
        'if C [expression] 'then
            'goto L1 [label] ';
        'end;
        Statements [statement*]
        'goto L0 ';
    L1 ':
        Follow [statement]
    Rest [statement*]
```

Paradigm. *Matching a subsequence.* The trailing Rest [statement*] is necessary since we are matching a subsequence of an entire sequence. If the pattern were to end without including the trailing sequence (i.e., without Rest), then it would only match when the pattern appeared as the last statements in the sequence of statements, which is not what we intend.

File "TILgotos.grm"

```
% Dialect of TIL that adds goto statements

redefine statement
       ...
    |  [labelled_statement]
    |  [goto_statement]
    |  [null_statement]
end redefine

define labelled_statement
    [label] ': [statement]
end define

define goto_statement
    'goto [label] ';  [NL]
end define

% Allow for trailing labels
define null_statement
    [NL]
end define

define label
    [id]
end define

% Add missing "not" operator to TIL
redefine primary
       ...
    |  '! [primary]
end redefine
```

File "TILgotoelim.txl"

```
% Goto elimination in TIL programs

% Recognize and resolve while-equivalent
% goto structures.

% Using the goto dialect of basic TIL
include "TIL.grm"
include "TILgotos.grm"

% Preserve comments in this transformation
#pragma -comment

% Main program - just applies the rules
% for cases we know how to transform.

function main
    replace [program]
        P [program]
    by
        P [transformForwardWhile]
          [transformBackwardWhile]
end function

% Case 1 - structures of the form
%        loop:
%            if Cond then goto endloop; end
%            LoopStatements
%            goto loop;
%        endloop:
%            TrailingStatements

rule transformForwardWhile
    % We're looking for a labelled if guarding
    % a goto - it could be the head of a loop
    replace [statement*]
        L0 [label] ':
            'if C [expression] 'then
         'goto L1 [label] ';
            'end;
        Rest [statement*]
    % If we have a goto back to the labelled if,
    % we have a guarded loop (i.e., a while)
    % The "skipping" makes sure we look only
    % in this statement sequence, not deeper
    skipping [statement]
    deconstruct * Rest
        'goto L0 ';
        L1 ':
            Follow [statement]
        FinalRest [statement*]
    % The body of the loop is the statements
    % after the if and before the goto back
    construct LoopBody [statement*]
        Rest [truncateGoto L0 L1]
    by
        'while '! (C) 'do
            LoopBody
        'end;
        Follow
        FinalRest
end rule

rule transformBackwardWhile

    . . . (similar to above for backward case)

end rule

% Utility rule used by all cases

function truncateGoto L0 [label] L1 [label]
    skipping [statement]
    replace * [statement*]
        'goto L0 ';
        L1 ':
            Follow [statement]
        FinalRest [statement*]
    by
        % nothing
end function
```

Fig. 19. TXL dialect grammar to add goto statements and labels to TIL, and transformation to eliminate gotos (showing first case only)

What is not so obvious is why we could not simply write the pattern above directly in the rule. The reason again has to do with the definition of [X*], which as we recall is recursively defined as [X] [X*] or empty. The pattern above is trying to match [statement] [statement*] [statement] [statement] [statement*], which can't be parsed using that definition no matter how we group it.

Paradigm. *Matching a gapped subsequence.* The TXL paradigm to match such "gapped" sequences is the one used in the [transformForwardWhile] rule. In it, we first match the head of the pattern we are looking for, that is, the leading if statement and the statements following it. We then search in the statements following it for the trailing pattern, the goto back and the ending forward label. The trick of the paradigm is that we must not look inside the statements of the sequence, because we want the trailing pattern to be in the same statement sequence. This is achieved using a *skipping deep deconstruct.*

```
skipping [statement]
deconstruct * Rest
        'goto L0 ';
    L1 ':
        Follow [statement]
        FinalRest [statement*]
```

This deconstructor says that we only have a match if we can find the goto back and the ending forward label without looking inside any of the statements in the sequence (that is, if they are both at the same level, in the statement sequence itself). "skipping [T]" limits a search to the parse tree nodes above any embedded [T]s - in our case, above any statements, so that the goto back is in the same sequence as the heading if statement, completing the pattern we are looking for.

Paradigm. *Truncating the tail of a sequence.* The other new paradigm this example shows us is the truncation of a trailing subsequence, achieved by the function [truncateGoto], which removes everything from the goto on from the statements following the initial if statement. The trick in this function is to look for the pattern heading the trailing subsequence we want to truncate, and replacing it and the following items by an empty sequence. Once again we use the *skipping* notation, since we don't want to accidentally match a similar instance in a deeper statement.

6 Optimization Problems

Source transformation tools are often used in source code optimization tasks of various kinds, and TXL is no exception. In this section we attack some traditional source code optimizations, observing the TXL paradigms that support these kinds of tasks. Once again, our examples are based on the Tiny Imperative Language (TIL) and its extensions.

6.1 Statement-Level Code Motion

The first example problem is on the border between restructuring and optimization: moving invariant assignments and computations out of while loops. In the first solution, we simply look for assignment statements in while loops that are independent of the loop (that is, that don't change over the iterations of the loop). For example, in this loop, the assignment to x does not depend on the loop and can be moved out:

```
var j; var x; var y; var z;
j := 1; x := 5; z := 7;
while j != 100 do
    y := y + j -1;
    x := z * z;
    j := j + 1;
end;
```

Figure 20 shows a solution to this problem for TIL programs. The key to the solution is the function [loopLift], which, given a while loop and an assignment statement in it, checks to see whether the assigned expression of the assignment contains only variables that are not assigned in the loop, and that the assigned variable of the assignment is assigned exactly once in the loop. If both these conditions are met, then the assignment is moved out by putting a copy of it before the loop and deleting it from the loop.

The function uses a number of TXL paradigms. It begins by deconstructing the assignment statement it is passed to get its parts, then uses the extract paradigm to get all of the variable references in the assigned expression. Both of these paradigms we have seen before. The interesting new paradigm is the guarding of the transformation using *where* clauses:

```
% We can only lift the assignment out if all the identifiers in its
% expression are not assigned in the loop ...
where not
    Loop [assigns each IdsInExpression]

% ... and X itself is assigned only once
deconstruct * Body
    X := _ [expression];
    Rest [statement*]
where not
    Rest [assigns X]

% ... and the effect of it does not wrap around the loop
construct PreContext [statement*]
    Body [deleteAssignmentAndRest X]
where not
    PreContext [refers X]
```

Paradigm. *Guarding a transformation with a complex condition.* Where clauses guard the pattern match of a rule or function with conditions that are tested by a subrule or set of subrules. If the *where* clause is positive (i.e., has no *not* modifier), then the subrule must match its pattern for the rule to proceed. If it is a *where not*, as in these cases, then it must not match its pattern.

Paradigm. *Condition rules.* The subrules used in a *where* clause are of a special kind called *condition rules*, which have only a pattern and no replacement. The pattern may be simple, as in the [assigns] and [refers] subrules of this example, which simply check to see if their parameter occurs in the context of their scope, or they may be complex, involving other deconstructors, where clauses and subrules. In either case, a condition subrule simply matches its pattern or not, and the where clause using it succeeds or not depending on whether it matches. If multiple subrules are used in the condition, the where clause succeeds if any one of them matches, and fails only if all do not match (or conversely for *where not*, succeeds only if none match).

```
File "TILcodemotion.txl"

% Lift independent TIL assignments outside
% of while loops

% Based on the TIL grammar
include "TIL.grm"

% Lift all independent assignments out of loops
rule main
    % Find every loop
    replace [statement*]
        while Expn [expression] do
            Body [statement*]
        'end;
        Rest [statement*]

    % Get all the top-level assignments in it
    construct AllAssignments [statement*]
        Body [deleteNonAssignments]

    % Make a copy of the loop to work on
    construct LiftedLoop [statement*]
        while Expn do
            Body
        'end;

    % Only proceed if there are assignments
    % left that can be lifted out.
    % The [?loopLift] form tests if the
    % [loopLift] rule can be matched -
    % "each AllAssignments" tests this
    % for any of the top-level internal
    % assignments
    where
        LiftedLoop
            [?loopLift Body each AllAssignments]

    % If the above guard succeeds,
    % some can be moved out, so go ahead
    % and move them, replacing the original
    % loop with the result
    by
        LiftedLoop
            [loopLift Body each AllAssignments]
            [. Rest]
end rule

% Attempt to lift a given assignment
% outside the loop
function loopLift Body [statement*]
                  Assignment [statement]
    deconstruct Assignment
        X [id] := E [expression];

    % Extract a list of all the identifiers
    % used in the expression
    construct IdsInExpression [id*]
        _ [^ E]

    % Replace the loop and its contents
    replace [statement*]
        Loop [statement*]

    % We can only lift the assignment out
    % if all the identifiers in its
    % expression are not assigned in the loop ...
    where not
        Loop [assigns each IdsInExpression]

    % ... and X itself is assigned only once
    deconstruct * Body
        X := _ [expression];
        Rest [statement*]
    where not
        Rest [assigns X]

    % ... and the effect of it
    % does not wrap around the loop
    construct PreContext [statement*]
        Body [deleteAssignmentAndRest X]
    where not
        PreContext [refers X]

    % Now lift out the assignment
    by
        Assignment
        Loop [deleteAssignment Assignment]
end function

% Utility rules used above

% Delete a given assignment from a scope
function deleteAssignment Assignment [statement]
    replace * [statement*]
        Assignment
        Rest [statement*]
    by
        Rest
end function

% Delete all non-assignments in a scope
rule deleteNonAssignments
    replace [statement*]
        S [statement]
        Rest [statement*]
    deconstruct not S
        _ [assignment_statement]
    by
        Rest
end rule

% Delete everything in a scope from
% the first assignment to X on
function deleteAssignmentAndRest X [id]
    replace * [statement*]
        X := E [expression];
        Rest [statement*]
    by
        % nada
end function

% Does a scope assign to the identifier?
function assigns Id [id]
    match * [assignment_statement]
        Id := Expn [expression];
end function

% Does a scope refer to the identifier?
function refers Id [id]
    match * [id]
        Id
end function
```

Fig. 20. TXL transformation to lift independent assignments out of while loops

Paradigm. *Each element of a sequence.* The first where condition in the [loopLift] function also uses another paradigm - the *each* modifier.

```
where not
    Loop [assigns each IdsInExpression]
```

each takes a sequence of type [X*] for any type [X], and calls the subrule once with each element of the sequence as parameter. So for example, if IdsInExpression is bound to the sequence of identifiers "a b c", then "where not Loop [assigns each IdsInExpression]" means "where not Loop [assigns 'a] [assigns 'b] [assigns 'c]", and the guard succeeds only if none of the [assigns] calls matches its pattern. This is a common TXL paradigm for checking multiple items at once.

The main rule in this example simply finds every while loop, extracts all the assignment statements in it by making a copy of the statements in the loop body and deleting those that are not assignments, and then calls [loopLift] with *each* to try to move each of them outside the loop. Rather than use the fixed-point paradigm, this main rule uses a *where* clause as a guard to check whether there are any assignments to move in advance. To do this, it actually uses the [loopLift] function itself to check - by converting it to a condition using [?].

```
where
    LiftedLoop [?loopLift Body each AllAssignments]
```

Paradigm. *Using a transformation rule as a condition.* [?loopLift] means that [loopLift] should not do any replacement - rather, it should act as a condition rule, simply checking whether its complex pattern matches or not. Thus the where clause above simply checks whether [loopLift] will succeed for any of the assignments, and the rule only proceeds if at least one will match.

6.2 Common Subexpression Elimination

Common subexpression elimination is a traditional optimization transformation that searches for repeated subexpressions whose value cannot have changed between two instances. The idea is to introduce a new temporary variable to hold the value of the subexpression and replace all instances with a reference to the temporary. For example, if the input contains the code on the left (a) below, then the output should be the code (b) shown on the right.

```
var a; var b;
read a;
b := a * (a + 1);
var i;
i := 7;
c :=  a * (a + 1);
```

(a)

```
var a; var b;
read a;
var t;
t := a * (a + 1);
b := t;
var i;
i := 7;
c := t;
```

(b)

A TXL solution to this problem for TIL programs is shown in Figure 21. The solution uses a number of new paradigms for us to look at. To begin, the program uses *agile parsing* to modify the TIL grammar in two ways.

Paradigm. *Grammatical form abstraction.* First, it overrides the definition of [statement] to gather all compound statements into one statement type. This

File "TILcommonsubexp.txl"

```
% Recognize and optimize common subexpressions

% Based on the TIL base grammar
include "TIL.grm"

% Preserve comments
#pragma -comment

% Override to abstract compound statements
redefine statement
        [compound_statement]
    |   ...
end redefine

define compound_statement
        [if_statement]
    |   [while_statement]
    |   [for_statement]
end define

% Allow statements to be attributed
% so we don't mistake one we've
% generated for one we need
% to process
redefine statement
        ...
    |   [statement] [attr 'NEW]
end redefine

% Main rule
rule main
    replace [statement*]
        S1 [statement]
        SS [statement*]
    % Don't process statements we generated
    deconstruct not * [attr 'NEW] S1
        'NEW
    % We're looking for an expression ...
    deconstruct * [expression] S1
        E [expression]
    % ... that is nontrivial ...
    deconstruct not  E
        _ [primary]
    % ... and repeated
    deconstruct * [expression] SS
        E
    % See if we can abstract it
    % (checks if variables assigned between)
    where
        SS [?replaceExpnCopies S1 E 'T]
    % If so, generate a new temp name ...
    construct T [id]
        _ [+ "temp"] [!]
    % ... declare it, assign it the expression,
    % and replace instances with it
    by
        'var T;    'NEW
        T := E;    'NEW
        S1 [replaceExpn E T]
        SS [replaceExpnCopies S1 E T]
end rule

% Recursively replace copies of a given
% expression with a given temp variable id,
% provided the variables used in the
% expression are not assigned in between

function replaceExpnCopies S1 [statement]
                           E [expression] T [id]
    construct Eids [id*]
        _ [^ E]

    % If the previous statement did not assign
    % any of the variables in the expression
    where not
        S1 [assigns each Eids]

    % Then we can continue to substitute the
    % temporary variable for the expression
    % in the next statement ...
    replace [statement*]
        S [statement]
        SS [statement*]

    % ... as long as it isn't a compound
    % statement that internally assigs one of
    % the variables in the expression
    where not all
        S [assignsOne Eids]
          [isCompoundStatement]
    by
        S [replaceExpn E T]
        SS [replaceExpnCopies S E T]
end function

% Check to see if a statement assigns
% any of a list of variables

function assignsOne Eids [id*]
    match [statement]
        S [statement]
    where
        S [assigns each Eids]
end function

function assigns Id [id]
    match * [statement]
        Id := _ [expression] ;
end function

function isCompoundStatement
    match [statement]
        _ [compound_statement]
end function

rule replaceExpn E [expression] T [id]
    replace [expression]
        E
    by
        T
end rule
```

Fig. 21. TXL transformation to recognize and optimize common subexpressions

redefinition takes advantage of TXL's programmed parsing to prefer that if, while and for statements be parsed as [compound_statement]. The original forms are still in the definition of [statement] (denoted by "..."), but since our new form appears first, all of them will be parsed as [compound_statement]. This paradigm is often used to gather forms so that we can use one rule to target all of the forms at once rather than having several rules for the different grammatical types.

Paradigm. *Marking using attributes.* The second technique used here is grammar attributes, denoted by the [attr] modifier. TXL grammar attributes denote optional parts of the grammar that will not appear in the unparsed output text. They can be of any grammatical type, including complex types with lots of information in them. In this case, the attribute is simply the identifier "NEW", and it is added to allow us to mark statements that are inserted by the transformation so that we don't mistake them for a statement to be processed.

Marking things that have been generated or already processed using attributes is a common technique in TXL, and is often the easiest way to distinguish things that have been processed from those that have not. The new attributed form is recursive, allowing any statement to be marked as "NEW".

The main rule finds any statement containing a nontrivial expression, determined by deconstructing it to ensure that it is not simply a [primary]. It then deconstructs the following statements to determine if the expression is repeated in them. If so, then it uses the conditional guard paradigm to check that the repetition will be legally transformable [?replaceExpnCopies]. A new unique temporary name of the form "temp27" is then created using the unique identifier paradigm, and finally, statements are generated to declare and assign the expression to the new temporary.

This is where the NEW attribute comes in. By marking the newly generated statements with the NEW attribute, we are sure that they will not be matched by the main rule and reprocessed. The remainder of the replacement copies the original statement and following statements, substituting the new temporary name for the expression in the original statement [replaceExpn E T] and any subsequent uses in following statements [replaceExpnCopies S1 E T].

Paradigm. *Tail-recursive continuation.* Rule [replaceExpnCopies] (Figure 21) introduces us to another new paradigm - continuing a transformation through a sequence as long as some condition holds. In this case, we can continue to substitute the temporary name for the common expression as long as the variables in the expression are not assigned to.

In TXL such situations are encoded as a tail-recursive function, which processes each statement one by one checking that the conditions still hold, until it fails and terminates the recursion. In each recursion we pass the previous statement as parameter, and first check that it has not assigned any of the identifiers used in the expression, again using the where-not-each paradigm of the previous problem. We then match the next statement in the sequence, and check that it is not a compound statement that assigns any of the identifiers in the expression.

Paradigm. *Guarding with multiple conditions.* This check uses a new paradigm - *where not all*. As we've seen in previous paradigms, where clauses normally check whether *any* of the condition rules matches. When *all* is specified, the check is whether *all* of the condition rules match. Thus the where clause here:

```
where not all
    S [assignsOne Eids]
      [isCompoundStatement]
```

checks whether it is both the case that one of the identifiers used in the expression is assigned by the statement, and that the statement is a compound statement (in which case our simple algorithm choose to give up and stop).

If the check succeeds and either the statement is not a compound statement or does not assign any of the variables in the original expression, then instances of the expression are substituted in the matched statement and we recursively move on to the next one.

6.3 Constant Folding

Constant folding, or optimizing by recognizing and precomputing compile-time known subexpressions, is another traditional optimization technique. In essence, the solution is a partial evaluation of the program, replacing named constants by their values and interpreting resulting operations on those values. Thus a constant folding algorithm must have rules to evaluate much of the expression sublanguage of the target language.

The solution for TIL (Figure 22) is in two parts: recognition and substitution of constant assignments to variables that are not destroyed, and interpretation of constant subexpressions. Of course, these two processes interact, because substitution of constant values for variables yields more constant subexpressions to evaluate, and evaluation of constant subexpressions yields more constant values for variables. So in the main rule we see the now familiar paradigm for a fixed point, continuing until neither rule changes anything.

The [propagateConstants] rule handles the first half of the problem, searching for assignments of constant values to variables (e.g., "x := 5;") that are not destroyed by a subsequent assignment in the same statement sequence. The two deep deconstructs of Rest are the key to the rule. The first one ensures that the following statements do not subsequently assign to the variable, destroying its constant value. The second one makes sure that there is a reference to the variable to substitute. When both conditions are met, the value is substituted for all references to the variable in the following statements.

The second half of the transformation is the interpretation of constant subexpressions (possibly created by the first half substituting constant variable values). The rule [foldConstantExpressions] simply applies a set of rules each of which knows how to evaluate an operator with constant operands. In addition to the simple cases, a number of special cases, such as multiplying any expression by zero, are also handled. [foldConstantExpressions] continues applying the set of evaluation rules until none of them changes anything and a fixed point is reached.

```
File "TILconst.txl"

% Constant propagation and folding for TIL

% Begin with the TIL base grammar
include "TIL.grm"

% Preserve comments in this transformation
#pragma -comment

% Main function
rule main
    replace [program]
        P [program]
    construct NewP [program]
        P [propagateConstants]
          [foldConstantExpressions]
    deconstruct not NewP
        P
    by
        NewP
end rule

% Constant propagation - find each
% constant assignment to a variable,
% and if it is not assigned again then
% replace references with the constant
rule propagateConstants
    replace [statement*]
        Var [id] := Const [literal] ;
        Rest [statement*]
    deconstruct not * [statement] Rest
        Var := _ [expression] ;
    deconstruct * [primary] Rest
        Var
    by
        Var := Const;
        Rest [replaceExpn Var Const]
end rule

rule replaceExpn Var [id] Const [literal]
    replace [primary]
        Var
    by
        Const
end rule

% Constant folding - find and evaluate
% constant expressions
rule foldConstantExpressions
    replace [expression]
        E [expression]
    construct NewE [expression]
        E % Generic folding of pure
          % constant expressions
          [resolveAddition]
          [resolveSubtraction]
          [resolveMultiplication]
          [resolveDivision]
          % Other special cases
          [resolveAdd0]
          [resolveSubtract0]
          [resolveMultiply1Right]
          [resolveMultiply1Left]
          [resolveParentheses]
    % Continue until we don't
    % find anything to fold
    deconstruct not NewE
        E
    by
        NewE
end rule

% Utility rules to do the arithmetic
rule resolveAddition
    replace [expression]
        N1 [integernumber]
           + N2 [integernumber]
    by
        N1 [+ N2]
end rule

rule resolveSubtraction
    replace [expression]
        N1 [integernumber]
           - N2 [integernumber]
    by
        N1 [- N2]
end rule

% ... other operator folding rules
  . . .
```

Fig. 22. TXL transformation to fold constant subexpressions

6.4 Statement Folding

Our last optimization example is statement folding, the elimination of statements that cannot be reached because the conditions that guard them are known at compile time, for example, when an if condition is known to be true or false. In practice, constant folding and statement folding go together - constant folding precomputes conditional expressions, some of which are then known to be true or false, allowing for statement folding. These problems are closely related to conditional compilation. Transformations to implement preprocessors and conditional compilation are essentially the same as constant and statement folding.

Figure 23 shows a TXL solution to the statement folding problem for TIL if and while statements with known conditions. In this case the main rule is a

```
File "TILstmtfold.txl"

% Statement folding using TIL

% Look for opportunities to reduce code
% footprint by optimizing out unreachable code

% Begin with the TIL base grammar
include "TIL.grm"

% Preserve comments
#pragma -comment

 % Main function
function main
    replace [program]
        P [program]
    by
        P [foldTrueIfStatements]
          [foldFalseIfStatements]
end function

% Folding rules for constant condition ifs

rule foldTrueIfStatements
    % Find an if statement
    replace [statement*]
        'if Cond [expression] 'then
            TrueStatements [statement*]
            ElseClause [opt else_statement]
        'end;
        Rest [statement*]

    % with a constant true condition
    where
        Cond [isTrueEqual] [isTrueNotEqual]

    % and replace it with the true part
    by
        '// Folded true if
        TrueStatements [. Rest]
end rule

rule foldFalseIfStatements
    % Find an if statement
    replace [statement*]
        'if Cond [expression] 'then
            TrueStatements [statement*]
            ElseClause [opt else_statement]
        'end;
        Rest [statement*]

    % with a constant false condition
    where not
        Cond [isTrueEqual]
             [isTrueNotEqual]

    % and replace it with the false part
    construct FalseStatements [statement*]
        _ [getElseStatements ElseClause]
    by
        '// Folded false if
        FalseStatements [. Rest]
end rule

function getElseStatements
            ElseClause [opt else_statement]
    deconstruct ElseClause
        'else
            FalseStatements [statement*]
    replace [statement*]
        % default none
    by
        FalseStatements
end function

% Utility functions to detect statically
% true conditions - these can be as
% smart as we wish

  . . .
```

Fig. 23. TXL transformation to fold known if statements

function, since none of the rules changes anything that may create new instances of the others, and thus the fixed point paradigm is not needed.

Paradigm. *Handling optional parts.* In the false if condition case (rule [foldFalseIfStatements]) there is a new paradigm used to get the FalseStatements from the else clause of the if statement. Beginning with an empty sequence using the empty variable "_", a separate function is used to get the FalseStatements from the else clause. The reason for this construction is that the [else_statement] is optional - there may not be one. So beginning with the assumption there is none (i.e., the empty sequence) we used the [getElseStatements] function to both check if there is one (by deconstructing the ElseClause) and if so to replace the empty sequence by the FalseStatements.

Paradigm. *Creating output comments.* Both cases illustrate another TXL paradigm - the creation of target language comments. Besides explicitly marking identifiers intended to be literal output, quoting of items in TXL marks something to be lexically interpreted in the target language rather than TXL. Thus

a target language comment can be created in a TXL replacement simply by pre-quoting it (Figure 23, rule [foldFalseIfStmts]). This can be handy when marking sections of code that have been transformed in output.

7 Static and Dynamic Analysis Problems

Now that we've tried some of the simpler problems and introduced many of the standard paradigms for using TXL, it's time to attack some more realistic challenges. Static and dynamic analysis tasks, including program comprehension, security analysis, aspect mining and other analyses, are commonly approached using parsing and source transformation tools. In this section we demonstrate the use of TXL in several example static and dynamic analyses, including static metrics, dynamic tracing, type inference, slicing, clone detection, code markup, unique renaming and fact extraction.

7.1 Program Statistics

In our first analysis example, we demonstrate TXL's use in computing static program metrics. Figure 24 shows a TXL program designed to gather statement usage statistics for TIL programs. The input is any TIL program, and the output is empty. However, the program uses the TXL message built-in functions to print out several statement statistics about the program on the standard error stream as it matches the measured features. The error stream output of this program when processing the "factors.til" program of Figure 7 looks like this:

```
Total: 11
Declarations: 2
Assignments: 3
Ifs: 0
Whiles: 2
Fors: 0
Reads: 1
Writes: 3
```

The program uses the TXL type extraction paradigm that we have seen before to collect all statements of each type into sequences, and then counts them using the sequence [length] built-in function to give the statistic.

Paradigm. *Counting feature instances.* This is a general paradigm that when combined with agile parsing (to gather our desired grammatical forms) and the filtering paradigm we have seen previously (to refine to the exact subset we are interested in) can be used to count instances of any feature or pattern in the program, including most standard static metrics. An important point here is the use of the empty variable "_" as the [number] scope of the counting constructors. When used in a [number] context, the empty variable plays the role of the value zero, which is often a good place to start a numeric computation.

Paradigm. *Dynamic error stream output.* The program uses the TXL error stream built-in function [putp] to output messages reporting the statistics as they are computed. [putp] is modeled after the C *printf* function, and acts in

```
File "TILstats.txl"

% Gather TIL statement statistics

% Begin with the TIL base grammar
include "TIL.grm"

% Compute and output statement kind counts,
% and replace program with an empty one.
% There are many different ways to do this -
% this naive way is simple and obvioulsy
% correct, but exposes TXL's need for generics.
% Another less clear solution could use
% polymorphism to avoid the repetition.

function main
    replace [program]
        Program [program]

    % Count each kind of statement we're
    % interested in by extracting all of
    % each kind from the program

    construct Statements [statement*]
        _ [^ Program]
    construct StatementCount [number]
        _ [length Statements]
          [putp "Total: %"]

    construct Declarations [declaration*]
        _ [^ Program]
    construct DeclarationsCount [number]
        _ [length Declarations]
          [putp "Declarations: %"]

    construct Assignments [assignment_statement*]
        _ [^ Program]
    construct AssignmentsCount [number]
        _ [length Assignments]
          [putp "Assignments: %"]

    construct Ifs [if_statement*]
        _ [^ Program]
    construct IfCount [number]
        _ [length Ifs] [putp "Ifs: %"]

    construct Whiles [while_statement*]
        _ [^ Program]
    construct WhileCount [number]
        _ [length Whiles] [putp "Whiles: %"]

    construct Fors [for_statement*]
        _ [^ Program]
    construct ForCount [number]
        _ [length Fors] [putp "Fors: %"]

    construct Reads [read_statement*]
        _ [^ Program]
    construct ReadCount [number]
        _ [length Reads] [putp "Reads: %"]

    construct Writes [write_statement*]
        _ [^ Program]
    construct WriteCount [number]
        _ [length Writes] [putp "Writes: %"]
    by
        % nothing
end function
```

Fig. 24. TXL transformation to collect and report statement statistics

much the same way, printing out its string parameter with the output text of the scope it is applied to substituted for the "%" in the string. In this case, the scope is a [number], and the corresponding number value is printed in the message.

In general, the scope of [putp] can be any type at all, and both [putp] and its simpler form [put], which takes no parameter and simply prints out the text of its scope, can be used to instrument and debug TXL programs as they execute.

7.2 Self Tracing Programs

The addition of auxiliary monitoring code to a program is a common transformation task, and in this example we demonstrate the paradigms for adding such code using TXL. The problem is to transform a TIL program to a self-tracing version of itself, one that prints out each statement just before it is executed. This is a model for a large number of transformations used in instrumentation and dynamic analysis tasks such as test coverage and dynamic flow analysis.

To distinguish statements that are generated or have already been processed by the transformation, the program uses the same attribute marking paradigm we have seen before to mark statements that have been generated or already processed, in this case marking with the attribute "TRACED".

Paradigm. *Eliding detail.* In order that we don't print out entire multi-line messages for compound statements, the program also allows for an elision marker

```
File "TILtrace.txl"

% Make a TIL program self-tracing

% Replaces every statement with a write
% statement of its text followed by itself

% Begin with the TIL base grammar
include "TIL.grm"

% Don't bother preserving comments,
% we only want to run the result

% Pragma to tell TXL that our string
% escape convention uses backslash
#pragma -esc "¨

% Allow elided structured statements
redefine statement
        ...
    |   [SP] '... [SP]
end redefine

% Allow for traced statements - the TRACED
% attribute marks statements already done
redefine statement
        ...
    |   [traced_statement]
end redefine

define traced_statement
    [statement] [attr 'TRACED]
end define

% Main rule
rule main
    % Result has two statements where one
    % was before, so work on the sequence
    replace [statement*]
        S [statement]
        Rest [statement*]
    % Semantic guard: if it's already
    % done don't do it again
    deconstruct not S
        _ [statement] 'TRACED
    % Make a concise version of
    % structured statements
    construct ConciseS [statement]
        S [deleteBody]
    % Get text of the concise statement
    construct QuotedS [stringlit]
        _ [+ "Trace: "] [quote ConciseS]
    by
        'write QuotedS;    'TRACED
        S                  'TRACED
        Rest
end rule

% Utility function - replace the body
% of a structured statement with ...
function deleteBody
    replace * [statement*]
        _ [statement*]
    by
        '...
end function
```

Fig. 25. TXL transformation to transform TIL program to self-tracing

"..." as a [statement]. This is used in the function [deleteBody], which makes a copy of a statement in which the body has been replaced by "..." so that the trace will be more terse. The function uses a deep search (*replace **) to find the outermost sequence of statements embedded in the statement which is its scope.

The main rule does all of the work, searching for every statement that has not yet been transformed (i.e., that is not yet marked with the attribute TRACED) and inserting a *write* statement to print out its quoted text before it is executed. Both the write statement and the original are attributed with TRACED in the replacement so that they are not themselves transformed again.

Once again we see the paradigm for replacing one element of a sequence with more than one by targeting the higher level sequence [statement*] rather than the element [statement] - and again the pattern and replacement of the rule must preserve the Rest of the statements following the one we are transforming.

Paradigm. *Converting program text to strings.* The construction of the quoted string version of the statement's text to be printed in the trace uses the TXL string manipulation built-in functions [+] and [quote]. Beginning with an empty [stringlit], once again denoted by the empty variable "_", the constructor concatenates the string literal "Trace: " to the quoted text of the statement.

```
construct QuotedS [stringlit]
    _ [+ "Trace: "] [quote ConciseS]
```

The [quote] built-in function creates a string literal containing the text of its parameter, which may be any grammatical type, and concatenates it to its scope, in this case the string "Trace: ". The output of a run of the traced version of the "factors.til" TIL program of Figure 7, when executed, looks like this:

```
Trace: var n;
Trace: write "Input n please";
Input n please
Trace: read n;
read: 6
Trace: write "The factors of n are";
The factors of n are
Trace: var f;
Trace: f := 2;
Trace: while n != 1 do ... end;
Trace: while (n / f) * f = n do ... end;
Trace: write f;
2
Trace: n := n / f;
Trace: f := f + 1;
Trace: while (n / f) * f = n do ... end;
Trace: write f;
3
```

7.3 Type Inference

Calculation of derived or inferred attributes of items in a program is a common analysis task, forming part of type checking, optimization of dynamically typed programs, translation between languages, and business type analysis such as the Y2K problem. In this example we demonstrate the TXL paradigms for concrete and abstract type inference, using a transformation to infer static types for the untyped variables in a TIL program from the contexts in which they are used.

TIL declares untyped variables, originally intended to be all integer. However, the addition of string values to the language led to string variables, making the language effectively dynamically typed. However, perhaps TIL variables could be statically typed if they are used consistently. This transformation infers the type of every variable in a TIL program from its uses and explicitly adds types to declarations using the new form: "var x: integer;" where the valid types are "integer" and "string". Variables of inconsistent type are flagged as an error.

Figure 26 shows a solution to this problem. Using the precedence (PRIORITY) version of the TIL grammar, the program begins with several grammar overrides. First, the new form of declarations is added, by allowing for an optional type specification on each variable declaration. Types "int", "string" and "UNKNOWN" are allowed. The special type UNKNOWN is included so that we can mark variables whose type is inconsistent or that we cannot infer, so that error messages can be generated when we are done.

Next, we override the definition of [primary] to allow for a type attribute on every variable reference, literal constant and parenthesized expression in the program. We will use these attributes to record local inferences we make about types of variables and expressions.

Finally, to make the transformation more convenient, we use agile parsing to make the grammar easier to deal with for this particular problem. Everywhere in the TIL grammar where a variable appears as an [id] (i.e., "left hand side" references outside of expressions), we allow instead a [primary], so that it can be type attributed in the same way as in expressions.

```
redefine assignment_statement
   [primary] ':= [expression] ';   [NL]
end redefine
```

File "TILtypeinfer.txl"

```
% Infer types for variables and expressions

% Infer all expression and variable types,
% add types to variable declarations,
% and flag type conflicts.

% Based on the TIL base grammar
include "TIL.grm"

% Preserve comments
#pragma -comment

% Allow type specs on declarations.

redefine declaration
   'var [primary]
        [opt colon_type_spec] '; [NL]
end redefine

define colon_type_spec
   ': [type_spec]
end define

define type_spec
   'int | 'string | 'UNKNOWN
end define

% Allow type attributes on primaries.

redefine primary
   [subprimary] [attr type_attr]
end redefine

define subprimary
   [id] | [literal] | '( [expression] ')
end define

define type_attr
   '{ [opt type_spec] '}
end define

% Conflate all [id] refs to [primary],
% to make attribution rules simpler.

redefine assignment_statement
   [primary] ':= [expression] ';   [NL]
end redefine

redefine for_statement
    'for [primary] := [expression]
          'to [expression] 'do   [IN][NL]
       [statement*]              [EX]
    'end ';                      [NL]
end redefine

redefine read_statement
   'read [primary] ';  [NL]
end redefine

% The typing process has several steps:

%    1. introduce complete parenthesization,
%    2. enter default empty type attributes,
%    3. attribute literal expressions,
%    4. infer attributes from context until
%       a fixed point is reached,
%    5. set type attribute of uninferred
%       items to UNKNOWN,
%    6. add declaration types from variables'
%       inferred type attribute,
%    7. report errors (i.e., UNKNOWN types),
%    8. undo complete parenthesization.

function main
    replace [program]
        P [program]
    by
        P [bracket]
          [enterDefaultAttributes]
          [attributeStringConstants]
          [attributeIntConstants]
          [attributeProgramToFixedPoint]
          [completeWithUnknown]
          [typeDeclarations]
          [reportErrors]
          [unbracket]
end function

% Rules to introduce and undo complete
% parenthesization to allow for detailed
% unambiguous type attribution

function bracket
    replace [program]
        P [program]
    by
        P [bracketExpressions]
          [bracketComparisons]
          [bracketTerms] [bracketFactors]
end function

rule bracketExpressions
    skipping [expression]
    replace [expression]
        E [expression] Op [logop] C [comparison]
    by
       '( E [bracketExpressions]
          Op C [bracketExpressions] ')
end rule

 . . . (bracketComparisons, bracketTerms,
        bracketFactors similar)

function unbracket
    replace [program]
        P [program]
    by
        P [unbracketExpressions]
          [unbracketComparisons]
          [unbracketTerms] [unbracketFactors]
end function

rule unbracketExpressions
    replace [expression]
        '( E [expression] ')
            { Type [type_spec] }
    by
        E
end rule

 . . . (unbracketComparisons, unbracketTerms,
        unbracketFactors similar)

% Rule to add empty type attributes
% to every primary expression and variable

rule enterDefaultAttributes
    replace [attr type_attr]
    by
        { }
end rule
```

Fig. 26. TXL transformation to infer types of TIL variables

```
% The meat of the type inference algorithm.
% Infer empty type attributes from the types
% in the context in which they are used.
% Continue until no more can be inferred.

rule attributeProgramToFixedPoint
    replace [program]
        P [program]
    construct NP [program]
        P [attributeAssignments]
          [attributeExpressions]
          [attributeComparisons]
          [attributeTerms]
          [attributeFactors]
          [attributeForIds]
          [attributeDeclarations]
    deconstruct not NP
         P
    by
         NP
end rule

rule attributeStringConstants
    replace [primary]
        S [stringlit] { }
    by
        S { string }
end rule

rule attributeIntConstants
    replace [primary]
        I [integernumber] { }
    by
        I { int }
end rule

rule attributeAssignments
    replace [assignment_statement]
        X [id] { } := SP [subprimary]
            {Type [type_spec] };
    by
        X { Type } := SP { Type };
end rule

. . . (attributeForIds similar)

rule attributeExpressions
   replace [primary]
      ( P1 [subprimary] {Type [type_spec]}
        Op [logop] P2 [subprimary] {Type} ) { }
   by
      ( P1 {Type} Op P2 {Type} ) {Type}
end rule

. . . (attributeComparisons, attributeTerms,
       attributeFactors similar)

rule attributeDeclarations
    replace [statement*]
       'var Id [id] { } ;
        S [statement*]
    deconstruct * [primary] S
        Id { Type [type_spec] }
    by
       'var Id { Type };
        S [attributeReferences Id Type]
end rule

rule attributeReferences
        Id [id] Type [type_spec]
    replace [primary]
        Id { }
    by
        Id { Type }
end rule

% Once a fixed point has been reached,
% set all such remaining empty type
% attributes to UNKNOWN.

rule completeWithUnknown
    replace [attr type_attr]
       { }
    by
       { UNKNOWN }
end rule

% Add an explicit type to every untyped
% variable declaration, from the
% variable's inferred type attribute.

rule typeDeclarations
    replace [declaration]
        'var Id [id] { Type [type_spec] };
    by
        'var Id { Type } : Type;
end rule

% Report type errors.  An UNKNOWN
% attribute indicates either a conflict or
% not enough information to infer a type.

rule reportErrors
    replace $ [statement]
       S [statement]
    skipping [statement*]
    deconstruct * [type_spec] S
        'UNKNOWN

    % Issue an error message.
    % [pragma "-attr"] allows attributes
    % to be printed in the message.

    construct Message [statement]
        S [pragma "-attr"] [message
    "*** ERROR: Unable to resolve types in:"]
          [stripBody] [putp "%"]
          [pragma "-noattr"]
    by
         S
end rule

function stripBody
    replace * [statement*]
       _ [statement*]
    by
        % nothing
end function
```

Fig. 27. TXL transformation to infer types of TIL variables (*continued*)

```
redefine for_statement
    'for [primary] := [expression] 'to [expression] 'do  [IN][NL]
       [statement*]                                      [EX]
    'end                                                 [NL]
end redefine

redefine read_statement
   'read [primary] ';  [NL]
end redefine
```

Paradigm. *Grammatical form generalization.* Technically this allows for many forms that are not legal in TIL - for example, the form "4 := 7;". But since this is a program analysis transformation, we can assume that our input is well-formed. This is a general paradigm in TXL - using a more lenient grammar than the target language in order to subsume forms that will be handled in the same way into a single grammatical type in order to simplify transformation rules. This is the core idea in agile parsing [13].

The transformation rules use a number of new paradigms: normalization of the program so that all cases are the same, inference of attributes to a fixed point using a set of local inference rules, promotion of locally inferred attributes to the global scope, and denormalization of the final result.

Paradigm. *Program normalization.* In this case the normalization is simple - the normalizing rule [bracketExpressions] converts every [expression] in the program to a fully parenthesized version. Full parenthesization both makes every expression into a [primary], which allows it to be attributed with a type due to the overrides above, and limits every [expression] to one operator, since subexpression operands will be also be fully parenthesized. This reduces our inference problem to only one case - that of a single operator, simplifying and clarifying the inference rules. This kind of simplifying normalization is typical of many source analysis tasks, and is essential to any complex inference problem in TXL.

The denormalizing rule [unbracketExpressions] both unparenthesizes and removes the inferred type attribute of the expression, since the result of type inference is in the explicit types on variable declarations in the result.

Paradigm. *Default analysis results.* Following normalization, a default empty type attribute is added to every [primary] in the input program using the rule [enterDefaultAttributes]. This secondary normalization again reduces the number of cases, since rules can handle both attributed and unattributed primaries in the same way. Such defaulting is also typical of inference tasks in TXL.

Once these normalizations are complete, the actual type inference algorithm is simple - we just look for opportunities to infer the type of as yet untyped items in a context where other types are known. This begins with simple typing of literal primaries, whose type is native to their value, using the rules [attributeStringConstants] and [attributeIntConstants]. This is the base case of the inductive inference algorithm.

Paradigm. *Inductive transformation.* The process then proceeds using a small set of contextual inference rules, using the fixed-point paradigm to halt when not more types can be inferred. The key rule is [attributeOperations], which infers

the type of an operator expression from the types of its operands, which looks for operations with two operands of the same type, and infers that the result must also be of that type. Of course, such inference rules depend on the programming language, but the basic strategy remains the same.

Another key inference rule is [attributeDeclarations], which infers the type of a variable declaration from any one of its references, and then marks all other references with the same type. [attributeDeclarations] uses a deep deconstructor of the statements following the declaration to see if a type has been inferred for any reference to the variable, and if so, gives the declaration that type and marks all other references in the following statements with it. (Using the local-to-global paradigm we saw in the restructuring examples.) This new typing can in turn can give more information for the next iteration of the operator inference rule above, and so on. Once the inference rules have come to a fixed point, any remaining unknown types are given the special type UNKNOWN.

Finally, we insert the inferred types into all variable declarations, and then report errors by printing out all statements containing types we could not infer - those attributed as UNKNOWN.

```
construct Message [statement]
    S [pragma "-attr"]
      [message "*** ERROR: Unable to resolve types in:"] [stripBody] [putp "%"]
      [pragma "-noattr"]
```

Paradigm. *Making attributes visible.* The message constructor illustrates the ability of TXL to include attributes in the output text - by turning on the "-attr" option, attributes are printed in the output text, in this case of the [putp] function, so that they can be seen in the error message:

```
"*** ERROR: Unable to resolve types in:
x {UNKNOWN} := ( y {string} + 1 {int} ) {UNKNOWN};
```

The [pragma] function allows us to turn TXL options on and off dynamically.

7.4 Static Slicing

Dependency analysis is an important and common static analysis technique, and one of the most common dependency analyses is the static slice. As defined by Weiser [22], a (backward) *slice* is the executable subset of the statements in a program that preserves the behavior observable at a particular statement. If the slice is executed with the same input as the program, then all variable values when the slice reaches the statement will be the same as if the original program were to be executed to the same point. Often the value of one particular variable is designated as the one of interest, in which case values of others can be ignored.

Slicing algorithms are usually carried out by building a dependency graph for the program and then using graph algorithms to reduce it to the slice, which is mapped back to source statements afterward. However, as we have seen in the type inference example, in TXL we can compute dependency chains directly, using the inductive transformation paradigm.

Figure 28 shows a TXL program for backward slicing of TIL programs. The program uses a related TXL paradigm called *cascaded markup*, in which, beginning with one statement marked as the one of interest, statements which directly

influence that statement are marked, and then those that influence those statements, and so on until a fixed point is reached.

The program begins with grammar overrides to allow for XML-like markup of TIL statements. The input to the program will have one such statement marked as the one of interest, as shown on the left (a) below. The output slice for this input is shown on the right (b).

```
        var chars;
        var n;
        read n;
        var eof_flag;
        read eof_flag;
        chars := n;
        var lines;
        lines := 0;
        while eof_flag do
            lines := lines + 1;
            read n;
            read eof_flag;
            chars := chars + n;
        end;
        write (lines);
<mark> write (chars); </mark>
```

(a)

```
var chars;
var n;
read n;
var eof_flag;
read eof_flag;
chars := n;


while eof_flag do

    read n;
    read eof_flag;
    chars := chars + n;
end;

write (chars);
```

(b)

Here the statement "write (chars);" has been marked. The challenge for the slicer is to trace dependencies backwards in the program to mark only those statements that can influence the marked one, yielding the backward slice for the program (b).

Paradigm. *Cascaded markup.* The basic strategy is simple: an assignment to a variable is in the backward slice if any subsequent use of the variable is already in the slice. The rule that implements the strategy is [backPropogateAssignments] (Figure 28). We have previously seen the "skipping" paradigm - here it prevents us from remarking statements inside an already marked statement.

The other markup propagation rules are simply special cases of this basic rule that propagate markup backwards into loop and if statements and around loops, and out to containing statements when an inner statement is marked. The whole set of markup propagation rules is controlled by the usual fixed-point paradigm that detects when no more propagation can be done.

Once a fixed point is reached, the program simply removes all unmarked statements [removeUnmarkedStatements] and unused declarations [removeRedundantDeclarations], then removes all markup to yield the program slice. The result fot the example (a) above is shown on the right (b). (Line spacing is shown to align with the original code, and is not part of the output.)

7.5 Clone Detection

Clone detection is a popular and interesting source analysis problem with a wide range of applications, including code reduction and refactoring. In this problem, we demonstrate the basic techniques for clone detection using TXL. Clone detection can vary in granularity from statements to functions or classes.

```
File "TILbackslice.txl"

% Backward static slicing of TIL programs

% Backward slice from a statement marked up
% using <mark> </mark>

% Begin with the TIL base grammar
include "TIL.grm"

% Allow for XML markup of TIL statements
redefine statement
        ...
    |   [marked_statement]
end redefine

define marked_statement
            [xmltag] [statement] [xmlend]
end define

define xmltag
        < [SPOFF] [id] > [SPON]
end define

define xmlend
        < [SPOFF] / [id] > [SPON]
end define

% Conflate while and for statements
% into one form to optimize handling
% of both forms in one rule
redefine statement
        [loop_statement]
    |   ...
end redefine

define loop_statement
        [loop_head]                 [NL][IN]
            [statement*]            [EX]
        'end;                       [NL]
end define

define loop_head
        while [expression] do
    |   for [id] := [expression]
            to [expression] do
end define

% The main function gathers the steps
% of the transformation: induce markup
% to a fixed point, remove unmarked
% statements, remove declarations for
% variables not used in the slice,
% and strip markups to yield the
% sliced program

function main
    replace [program]
        P [program]
    by
        P [propagateMarkupToFixedPoint]
          [removeUnmarkedStatements]
          [removeRedundantDeclarations]
          [stripMarkup]
end function

% Back propagate markup of statements
% beginning with the initially marked
% statement of interest.
% Continue until a fixed point

rule propagateMarkupToFixedPoint
    replace [program]
        P [program]

    construct NP [program]
        P [backPropogateAssignments]
          [backPropogateReads]
          [whilePropogateControlVariables]
          [loopPropogateMarkup]
          [loopPropogateMarkupIn]
          [ifPropogateMarkupIn]
          [compoundPropogateMarkupOut]

    % We're at a fixed point when P = NP
    deconstruct not NP
        P
    by
        NP
end rule

% Rule to back-propagate markup of
% assignments.  A previous assignment is
% in the slice if its assigned variable
% is used in a following marked statement

rule backPropogateAssignments
    skipping [marked_statement]
    replace [statement*]
        X [id] := E [expression] ;
        More [statement*]
    where
        More [hasMarkedUse X]
    by
        <mark> X := E; </mark>
        More
end rule

% Similar rule for read statements

rule backPropogateReads
    skipping [marked_statement]
    replace [statement*]
        read X [id] ;
        More [statement*]
    where
        More [hasMarkedUse X]
    by
        <mark> read X; </mark>
        More
end rule

function hasMarkedUse X [id]
    match * [marked_statement]
        M [marked_statement]
    deconstruct * [expression] M
        E [expression]
    deconstruct * [id] E
        X
end function

% Other propagation rules for loops
% and compound statements
  . . .
```

Fig. 28. TXL transformation to compute a backward slice

Since TIL does not have functions or classes, we are using structured statements (if, for, while) as a simple example. While this is clearly not a realistic case, detection of function or block clones (in any language) would be very similar.

Figure 29 shows a TXL solution to the detection of structured statement clones in TIL. The program begins with a set of grammar overrides to gather all of the structured statements into one type so that we don't need separate rules for each kind. As with the backward slicing example, we use XML-like markup to mark the results of our analysis. In this case, we want to mark up all instances of the same statement as members of the same clone class, so we allow for an XML attribute in the tags.

Paradigm. *Precise control of output spacing.* These overrides illustrate another output formatting cue that we have not seen before - the explicit control of output spacing. TXL normally uses a set of default output spacing rules that insert spaces around operators and other special symbols such as "<". Unfortunately, these spacing rules lead to strange output in the case of XML markup - for example, the XML tag "<clone class=4>" would be output as "< clone class = 4 >", which is not even legal XML.

The TXL built-in types [SPOFF], [SPON] and [SP] used here allow the programmer to take complete control of output spacing. Like [NL], [IN] and [EX], none of these has any effect on input parsing. [SPOFF] temporarily turns off TXL's output spacing rules, and [SPON] restores them. Between the two, items will be output with no spacing at all. The [SP] type allows programmers to insert spacing as they see fit, in this case forcing a space between the tag identifier and the attribute identifier in output tags.

As we have seen is often the case, the main TXL program works in two stages. In the first stage, a sequence containing one instance of each the cloned compound statements in the program is constructed using the function [findStructuredStatementClones]. In the second stage, all instances of each one of these are marked up in XML as instances of that clone class, assigning class numbers as we go (function [markCloneInstances]).

Paradigm. *Context-dependent rules.* The function [findStructuredStatementClones] works by using the subrule [addIfClone] to examine each of the set of all structured statements in the program (StructuredStatements) to see if it appears more than once, and if so adds it to its scope, which begins empty. While we have seen most of this paradigm before, we have not before seen a case where the transformation rule needs to look at both a local item and its entire global context at the same time to determine if it applies.

This kind of global contextual dependency is implemented in TXL using rule parameters. In this case an entire separate copy of StructuredStatements is passed to [addIfClone] so that it can use the global context in its transformation, in this case simply to check if each particular statement it is considering appears twice. This is an instance of the general TXL paradigm for context-dependent transformations, which allows for arbitrary contextual information, including if necessary a copy of the entire original program, to be passed in to a

File "TILclonesexact.txl"

```
% Clone detection for TIL programs

% Find exact clones of structured statements,
% and output the program with clones marked
% up to indicate their clone class.

% Begin with the TIL base grammar
include "TIL.grm"

% We are NOT interested in comments

% Overrides to conflate all structured
% statements into one nonterminal type.

redefine statement
        [structured_statement]
    |   ...
end redefine

define structured_statement
        [if_statement]
    |   [for_statement]
    |   [while_statement]
end define

% Allow XML markup of statements.

redefine statement
        ...
    |   [marked_statement]
end redefine

define marked_statement
    [xmltag]              [NL][IN]
        [statement]       [EX]
    [xmlend]              [NL]
end define

% [SPOFF] and [SPON] temporarily disable
% default TXL output spacing in tags

define xmltag
    < [SPOFF] [id] [SP] [id] = [number] > [SPON]
end define

define xmlend
    < [SPOFF] / [id] > [SPON]
end define

% Main program

function main
    replace [program]
        P [program]

    % First make a table of all repeated
    % structured statements
    construct StructuredClones
              [structured_statement*]
        _ [findStructuredStatementClones P]

    % Mark up all instances of each of them.
    % CloneNumber keeps track of the index of
    % each in the table as we step through it
    export CloneNumber [number] 0
    by
        P [markCloneInstances
              each StructuredClones]
end function

% We make a table of the cloned structured
% statements by first making a table
% of all structured statements in the program,
% then looking for repeats

function findStructuredStatementClones
         P [program]
    % Extract a list of all structured
    % statements in the program
    construct StructuredStatements
              [structured_statement*]
        _ [^ P]
    % Add each one that is repeated
    % to the table of clones
    replace [structured_statement*]
        % empty to begin with
    by
        _ [addIfClone StructuredStatements
                  each StructuredStatements]
end function

function addIfClone
        StructuredStatements [structured_statement*]
        Stmt [structured_statement]
    % A structured statement is cloned if it
    % appears twice in the list of all statements
    deconstruct * StructuredStatements
        Stmt
        Rest [structured_statement*]
    deconstruct * [structured_statement] Rest
        Stmt

    % If it does appear (at least) twice,
    % add it to the table of clones
    replace [structured_statement*]
        StructuredClones [structured_statement*]
    % Make sure it's not already in the table
    deconstruct not * [structured_statement]
            StructuredClones
        Stmt
    by
        StructuredClones [. Stmt]
end function

% Once we have the table of all clones,
% we mark up each instance of each of them
% in the program with its clone class,
% that is, the index of it in the clone table

rule markCloneInstances
        StructuredClone [structured_statement]
    % Keep track of the index of this clone
    % in the table
    import CloneNumber [number]
    export CloneNumber
        CloneNumber [+ 1]

    % Mark all instances of it in the program
    % 'skipping' avoids marking twice
    skipping [marked_statement]
    replace [statement]
        StructuredClone
    by
        <clone class=CloneNumber>
            StructuredClone
        </clone>
end rule
```

Fig. 29. TXL transformation to detect exact structured statement clones in TIL

local transformation. There it can be used both in conditions to guard the local transformation, as is the case this time, or as a source of additional parts to be used in the result of the local transformation.

Paradigm. *Accumulating multiple results.* [addIfClone] also demonstrates another common paradigm - the accumulation of results into a single sequence. Beginning with an empty sequence using the empty variable "_" in the replacement of [findStructuredStatementClones], [addIfClone] adds each result it finds to the end of its scope sequence. It makes sure that it does not put the same statement in twice using a guarding deconstructor, which checks to see if the cloned statement is already in the list:

```
deconstruct not * [structured_statement] StructuredClones
        Stmt
```

Once [findStructuredStatements] has constructed a unique list of all of the cloned structured statements in the program, [markCloneInstances] marks up all of the instances of each one in the program. Each is assigned a unique class number to identify it with its instances using the global variable CloneNumber, which begins at 0 and is incremented by [markCloneInstances] on each call.

Paradigm. *Updating global state.* While TXL is primarily a pure functional language, global state is sometimes required in complex transformations. For this purpose TXL allows global variables, which can be of any grammatical type (including forms that are not in the input language). In this case the global variable CloneNumber is a simple [number] that begins with the value 0. Inside a TXL rule, globals are simply normal local TXL variables. But they can be "exported" to the global scope where their value can be "imported" into another rule where they once again act as a local variable of the rule. Within a rule, the value bound to an imported global is set when it is imported, as if it were bound in a pattern match. The value bound to the variable can only be changed if the rule re-imports or exports the global with a new value.

In this case, on each invocation, [markCloneInstances] imports CloneNumber and immediately constructs and exports a new value for it, the previous value plus one. This new value is used in the replacement of the rule to mark up every instance of the current clone with that clone class number, making it clear which marked up statements are clones of one another in the result.

Of course, exact clone detection is the simplest case, and although interesting, not very realistic. Fortunately, we are using TXL, so modifying our clone detector to handle more aggressive techniques is not difficult. In particular, we can make the clones identifier-independent, like CCFinder [18], just by adding a normalization rule to make all identifiers the same when comparing:

```
rule normalizeIdentifiers
    replace $ [id]
        _ [id]
    by
        'X
end rule
```

If we want to be more precise, we can compare with consistent renaming - that is, where identifiers are normalized consistently with their original names.

```
% Rule to normalize structured statements
% by consistent renaming of identifiers
% to normal form (x1, x2, x3, ...)

rule renameStructuredStatement
    % For each outer structured statement
    % in the scope
    skipping [structured_statement]
    replace $ [structured_statement]
        Stmt [structured_statement]

    % Make a list of all of the unique
    % identifiers in the statement
    construct Ids [id*]
        _ [^ Stmt] [removeDuplicateIds]

    % Make normalized new names of the
    % form xN for each of them
    construct GenIds [id*]
        Ids [genIds 0]

    % Consistently replace each instance
    % of each one by its normalized form
    by
        Stmt [$ each Ids GenIds]
end rule
```

```
% Utility rule -
% remove duplicate ids from a list

rule removeDuplicateIds
    replace [id*]
        Id [id] Rest [id*]
    deconstruct * [id] Rest
        Id
    by
        Rest
end rule

% Utility rule -
% make a normalized id of the form xN
% for each unique id in a list

function genIds NM1 [number]
    % For each id in the list
    replace [id*]
        _ [id]
        Rest [id*]

    % Generate the next xN id
    construct N [number]
        NM1 [+ 1]
    construct GenId [id]
        _ [+ 'x] [+ N]

    % Replace the id with the generated one
    % and recursively do the next one
    by
        GenId
        Rest [genIds N]
end function
```

Fig. 30. TXL rule to consistently normalize identifiers in a TIL statement

Figure 30 shows a TXL rule to consistently rename the identifiers in a TIL structured statement. The rule works by extracting an ordered list of all of the identifiers used in the structured statement, and then generates a list of identifiers of the form x1, x2, x3 and so on of the same length by recursively replacing each identifier in a copy of the list with a generated one.

The result lists might look like this:

```
Ids          xyz  abc  n    wid  zoo
GenIds       x1   x2   x3   x4   x5
```

Paradigm. *Each corresponding pair.* The actual transformation of the original ids to the generated ones is done using the built-in rule [$], which is TXL shorthand for a fast global substitute. The rule application uses a paired *each* to pass the substitute rule each pair of corresponding identifiers in the lists, that is, [$'xyz$ 'x1], ['abc 'x2], and so on. This general paradigm can be used to match any two sequences of corresponding items, for example formals and actuals when analyzing function calls.

7.6 Unique Renaming

Unique renaming [15] gives scope-independent names to all declared items in a program. Unique naming flattens the name space so that every item declared in a program can be unambiguously referred to independent of its context. In

particular, unique naming is useful when creating a relationship database for the program in the form of facts, as in Rigi's RSF [21] or Holt's TA [16] format.

In this example transformation (Figure 31), we uniquely rename all declared variables, functions and modules in programs written in the module extension of TIL to reflect their complete scope. For example, a variable named X declared in function F of module M is renamed M.F.X .

This transformation demonstrates a number of new paradigms. Most obvious is that this process must be done from the innermost scopes to the outermost, so that when renaming things declared in a module M, all of the things declared in an embedded function F have already been renamed F.X. That way, we can simply rename everything transitively inside M with M. to reflect its scope, for example yielding M.F.X .

Paradigm. *Bottom-up traversal.* The paradigm for applying rules "inside out" (from the bottom up, from a parse tree point of view) is used in the main rule of this transformation, [uniqueRename] (Figure 31). [uniqueRename]'s real purpose is to find each declaration or statement that forms a scope, to get its declared name (ScopeName) and then use the [uniqueRenameScope] subrule to rename every declaration in the scope with the ScopeName. But in order for this to work correctly, it must handle the scopes from the inside out (bottom-up).

Bottom-up traversal is done by recursively applying the rule to each matched Scope more deeply before calling [uniqueRenameScope] for the current scope. The paradigm consists of two parts: "skipping [statement]" in [uniqueRename] assures that we go down only one level at a time, and the call to [uniqueRenameDeeper], which simply recursively applies [uniqueRename] to the inside of the current scope, ensures that we process deeper levels before we call [uniqueRenameScope] for the current level. This paradigm is generic and can be used whenever inner elements should be processed before outer.

The actual renaming is done by the rule [uniqueRenameScope], which finds every embedded declaration in a scope (no matter how deeply embedded), and renames both the declaration and all of its references in the scope to begin with the given ScopeName. For example, if ScopeName is M and some inner declaration is so far named F.G.X, then both the declaration and all references to it get renamed as M.F.G.X . Since we are processing inside out, there is no ambiguity with deeper declarations whose scopes have already been processed.

Paradigm. *Abstracted patterns.* [uniqueRenameScope] demonstrates another new paradigm: abstracted matching. Even though the real pattern it is looking for is a Declaration followed by its RestOfScope, the rule matches less precisely and uses a deconstructor to check for the pattern. This is because the replacement will have to consistently change both the Declaration and the RestOfScope in the same way (i.e. renaming occurrences of the declared name). By matching the part that requires change in one piece, the transformation requires only one use of the renaming substitution rule [$], making the rule simpler and clearer.

Once all declarations and embedded references have been renamed, there are two remaining tasks: renaming references to a module's public functions that

```
File "TILuniquerename.txl"

% Uniquely rename every Modular TIL variable
% and function with respect to its context.
% e.g., variable V declared in a while
% statement in function F of module M
% is renamed as M.F.whileN.V

% Begin with the MTIL grammar
include "TIL.grm"
include "TILarrays.grm"
include "TILfunctions.grm"
include "TILmodules.grm"

% Allow for unique names - in this case,
% TIL does not have a field selection operator,
% so we can use X.Y notation for scoped names.
redefine name
        [id]
    |   [id] . [name]
end redefine

% Main program
function main
    replace [program]
        P [program]
    by
        P [uniqueRenameDeeper]
          [uniqueRenameScope 'MAIN]
          [renameModulePublicReferences]
          [renameFunctionFormalParameters]
end function

rule uniqueRename
    % Do each statement on each level once
    skipping [statement]
    replace $ [statement]
        Scope [statement]
    % Only interested in statements with scopes
    deconstruct * [statement*] Scope
        _ [statement*]
    % Use the function, module or unique
    % structure name for it
    construct ScopeName [id]
        _ [makeKeyName Scope]
          [getDeclaredName Scope]
    % Visit inner scopes first, then this one
    by
        Scope [uniqueRenameDeeper]
              [uniqueRenameScope ScopeName]
end rule

% Recursively implement bottom-up renaming
function uniqueRenameDeeper
    replace * [statement*]
        EmbeddedStatements [statement*]
    by
        EmbeddedStatements [uniqueRename]
end function

% Make an identifier for the scope -
% if a declaration, use the declared id,
% otherwise synthesize a unique id from the
% statement keyword
function makeKeyName Scope [statement]
    deconstruct * [key] Scope
        Key [key]
    construct KeyId [id]
        _ [+ Key] [!]
    replace [id]
        _ [id]
    by
        KeyId
end function

function getDeclaredName Scope [statement]
    replace [id]
        _ [id]
    deconstruct Scope
        DeclaredScope [declaration]
    deconstruct * [id] DeclaredScope
        ScopeName [id]
    by
        ScopeName
end function

% Do the actual work - rename each declaration
% and its references with the scope id
rule uniqueRenameScope ScopeName [id]
    % Find a declaration in the scope
    replace $ [statement*]
        DeclScope [statement*]
    deconstruct DeclScope
        Declaration [declaration]
        RestOfScope [statement*]
    % Get its original id
    deconstruct * [name] Declaration
        Name [name]
    % Add the scope id to its name
    construct UniqueName [name]
        ScopeName '. Name
    % Rename the declaration and all
    % references in the scope.
    by
        DeclScope [$ Name UniqueName]
end rule

% This section handles the problem of
% references to a public function outside
% of the module it is declared in

rule renameModulePublicReferences
    % Find a module and its scope
    replace $ [statement*]
        'module ModuleName [name]
            ModuleStatements [statement*]
        'end ;
        RestOfScope [statement*]
    % Get all its public function names
    construct
            UniquePublicFunctionNames [name*]
        _ [extractPublicFunctionName
            each ModuleStatements]
    % Rename all references in the outer scope
    by
        'module ModuleName
            ModuleStatements
        'end ;
        RestOfScope
             [updatePublicFunctionCall
              each UniquePublicFunctionNames]
end rule
```

Fig. 31. TXL transformation to uniquely rename all declared items in TIL programs to reflect their scope

```
function extractPublicFunctionName
        Statement [statement]
    % We're interested only in functions
    deconstruct Statement
        Function [function_definition]
    % Which are public
    deconstruct * [opt 'public] Function
        'public
    % Get the function id
    deconstruct * [name] Function
        UniquePublicFunctionName [name]
    % Add it to the end of the list
    replace * [name*]
    by
        UniquePublicFunctionName
end function

rule updatePublicFunctionCall
        UniquePublicFunctionName [name]
    % Get the original name
    deconstruct * [name]
            UniquePublicFunctionName
        PublicFunctionName [id]
    % Replace all uses with unique name
    skipping [name]
    replace $ [name]
        PublicFunctionName
    by
        UniquePublicFunctionName
end rule

% Rules to rename function formal parameters
    ... (similar to rules above)
```

Fig. 32. TXL transformation to uniquely rename all declared items in TIL programs to reflect their scope (continued)

are outside its inner scope, and renaming formal parameters. Both of these pose a new kind of transformation problem: how to do a transformation on an outer level of the parse that depends on information from an inner level? Such a transformation is called a local-to-global transformation, and is a standard challenge for source transformation systems.

The TXL solution is demonstrated by the rule [renameModulePublicReferences] in Figure 32. We need to make a transformation of all references to the original name of any public function of the module that occur in the scope in which the module is declared, that is, in RestOfScope. But the public functions of the module cannot be in the pattern of the rule - what to do?

Paradigm. *Inner context-dependent transformation.* The answer is to contextualize the transformation by raising the information we need from the inner scope to the level we are at. In this case, that is done by the construct of UniquePublicFunctionNames, which uses the subrule [extractPublicFunctionName] to get a copy of the unique name of every public function declared in the module. Once we have brought the context up to the level we are at, we can do the transformation we need using [updatePublicFunctionCall] by passing it each public function unique name.

In general, the inner context to be raised could be much deeper or more complex than simply public functions declared one level down. Using a constructor and subrule to bring deeper context up, we can always get what is needed.

7.7 Design Recovery

Design recovery, or fact generation, is the extraction of basic program entities and relationships into an external graph or database that can be explored using graph and relationship analysis tools such as CrocoPat [4], Grok [17], or Prolog. In this problem, we show how TXL can be used to extract facts from programs using source transformation.

File "TILgeneratefacts.txl"

```
% Design recovery (fact extraction) for MTIL

% Given a uniquely renamed MTIL program,
% infer and generate architecture design facts
% contains(), calls(), reads(), writes()

% Begin with the MTIL grammar
include "TIL.grm"
include "TILarrays.grm"
include "TILfunctions.grm"
include "TILmodules.grm"

% Our input has been uniquely renamed by
% TILuniquerename.txl using X.Y notation
redefine name
        [id]
    |   [id] . [name]
end redefine

% Grammar for Prolog facts
include "Facts.grm"

% Override to allow facts on any statement
redefine statement
        ...
    |   ';     % null statement,
               % so we can add facts anywhere
end redefine

% Override to allow facts on any statement
redefine statement
        [fact*] ...
end redefine

% Override to allow facts on any expression
redefine primary
        [fact*] ...
end redefine

% Our output is the facts alone
redefine program
        ...
    |   [fact*]
end redefine

% Main program
function main
    replace [program]
        P [program]
    construct ProgramName [name]
        'MAIN
    construct AnnotatedP [program]
        P [addContainsFacts ProgramName]
          [inferContains]
          [addCallsFacts ProgramName]
          [inferCalls]
          [addReadsFacts ProgramName]
          [inferReads]
          [addWritesFacts ProgramName]
          [inferWrites]
    construct Facts [fact*]
        _ [^ AnnotatedP]
    by
        Facts
end function

% Infer contains() relationships
rule inferContains
    replace $ [declaration]
        ScopeDecl [declaration]
    deconstruct * [statement*] ScopeDecl
        Statements [statement*]
    deconstruct * [name] ScopeDecl
        ScopeName [name]
    by
        ScopeDecl
            [addContainsFacts ScopeName]
            [addContainsParameters ScopeName]
end rule

rule addContainsFacts ScopeName [name]
    skipping [statement]
    replace $ [statement]
        Facts [fact*] Declaration [declaration]
    deconstruct * [name] Declaration
        DeclName [name]
    construct NewFacts [fact*]
        'contains '( ScopeName, DeclName ')
        Facts
    by
        NewFacts Declaration
end rule

function addContainsParameters ScopeName [name]
    replace [declaration]
        Public [opt 'public]
        'function Fname [name]
            '( ParameterNames [name,] )
 OptResultParameter [opt colon_id]
            Statements [statement*]
        'end;
    construct OptResultParameterName [name*]
        _ [getResultParameterName
             OptResultParameter]
    construct ParameterContainsFacts [fact*]
        _ [makeFact 'contains ScopeName
                    each ParameterNames]
          [makeFact 'contains ScopeName
                    each OptResultParameterName]
    construct FactsStatement [statement]
        ParameterContainsFacts ';
    by
        Public
        'function Fname '( ParameterNames )
        OptResultParameter
            FactsStatement
            Statements
        'end;
end function

function getResultParameterName
         OptResultParameter [opt colon_id]
    deconstruct OptResultParameter
        ': ResultParameterName [name]
    replace [name*]
    by
        ResultParameterName
end function
```

Fig. 33. TXL transformation to generate basic facts for an MTIL program

```
% Infer calls() relationships
rule inferCalls
    replace $ [declaration]
        ScopeDecl [declaration]
    deconstruct * [statement*] ScopeDecl
        Statements [statement*]
    deconstruct * [name] ScopeDecl
        ScopeName [name]
    by
        ScopeDecl [addCallsFacts ScopeName]
end rule

rule addCallsFacts ScopeName [name]
    skipping [declaration]
    replace $ [statement]
        Facts [fact*]
        CallStatement [call_statement]
    skipping [id_assign]
    deconstruct * [name] CallStatement
        CalledName [name]
    by
        'calls '( ScopeName, CalledName ')
        Facts
        CallStatement
end rule

% Infer reads() relationships
rule inferReads
    replace $ [declaration]
        ScopeDecl [declaration]
    deconstruct * [statement*] ScopeDecl
        Statements [statement*]
    deconstruct * [name] ScopeDecl
        ScopeName [name]
    by
        ScopeDecl [addReadsFacts ScopeName]
end rule

rule addReadsFacts ScopeName [name]
    skipping [statement]
    replace $ [statement]
        Statement [statement]
    by
        Statement
            [addExpressionReadsFacts ScopeName]
end rule

rule addExpressionReadsFacts ScopeName [name]
    skipping [declaration]
    replace $ [primary]
        Primary [primary]
    deconstruct * [name] Primary
        FetchedName [name]
    construct ReadsFact [fact*]
        'reads ( ScopeName, FetchedName )
    by
        Primary [addFacts ReadsFact]
end rule

% Infer writes() relationships
. . .  ( similar to reads() )

% Utility functions
function makeFact FactId [id]
                 Name1 [name] Name2 [name]
    replace * [fact*]
    by
        FactId ( Name1, Name2 )
end function

function addFacts NewFacts [fact*]
    replace * [fact*]
        Facts [fact*]
    by
        Facts [. NewFacts]
end function
```

Fig. 34. TXL transformation to generate basic facts for an MTIL program (continued)

Figure 33 shows a program that extracts basic structural and usage facts for programs written in the module dialect of TIL. Facts extracted include *contains(), calls(), reads()* and *writes()* relationships for all modules and functions.

Paradigm. *Local fact annotation.* The basic strategy of the program is to annotate the program with facts directly in the local contexts where the fact can be inferred. For example, for the statement: x.y.z := a.b.c; appearing in function M.F, we will annotate the statement with the facts:

```
writes (M.F, x.y.z)
reads (M.F, a.b.c)
x.y.z := a.b.c;
```

In this way we can use local transformations to create the facts where the evidence for them occurs. The actual rules to infer each kind of fact are fairly simple: for each declaration in a scope, we annotate with a *contains()* fact. For each reference to a name in a scope, we annotate with a *reads()* fact. And so on. If more information is needed to infer a fact, we can leverage all of the previous techniques we have seen to assist us: context-dependent transformation, inner context-dependent transformation, bottom-up traversal, and any others we need.

Once the program is completely annotated with facts, the only remaining task is to gather them together, which is done using the usual type extraction paradigm to bring all the facts into one sequence, which we can then output as the result of our fact generation transformation. The final result of this program is a fact base in Prolog form, that looks like this:

```
contains (MAIN, MAIN.maxprimes)
contains (MAIN, MAIN.maxfactor)
writes (MAIN, MAIN.maxprimes)
writes (MAIN, MAIN.maxfactor)
contains (MAIN, MAIN.prime)
writes (MAIN, MAIN.prime)
contains (MAIN, MAIN.flags)
contains (MAIN.flags, MAIN.flags.flagvector)
contains (MAIN.flags, MAIN.flags.flagset)
contains (MAIN.flags.flagset, MAIN.flags.flagset.f)
writes (MAIN.flags.flagset, MAIN.flags.flagvector)
reads (MAIN.flags.flagset, MAIN.flags.flagset.f)
. . .
```

8 Conclusion and Future Work

The TXL Cookbook is very much a work in progress, and what we have seen is only part of what we hope will eventually be a comprehensive guide to using TXL in every kind of software analysis and transformation task. We have chosen this set of examples specifically to highlight some of the non-obvious ways in which TXL can be used to efficiently implement many tasks.

By using a range of real problems rather than small toy examples, we have been able to expose a number of paradigms of use that allow TXL to be effective. The real power of the language lies not in its own features, but rather in the way it is used - these solution paradigms. The purpose of the cookbook is to document and demonstrate these paradigms so that potential users can see how to solve their own problems using TXL and similar tools.

References

1. Barnard, D.T., Holt, R.C.: Hierarchic Syntax Error Repair for LR Grammars. Int. J. Computing and Info. Sci. 11(4), 231–258 (1982)
2. Baxter, I., Pidgeon, P., Mehlich, M.: DMS: Program Transformations for Practical Scalable Software Evolution. In: Proc. Int. Conf. on Software Engineering, pp. 625–634. ACM Press, New York (2004)
3. Bergstra, J.A., Heering, J., Klint, P.: Algebraic Specification. ACM Press, New York (1989)
4. Beyer, D.: Relational programming with CrocoPat. In: Proc. Int. Conf. on Software Engineering, pp. 807–810. ACM Press, New York (2006)
5. van den Brand, M., Klint, P., Vinju, J.J.: Term Rewriting with Traversal Functions. ACM Trans. on Software Eng. and Meth. 12(2), 152–190 (2003)
6. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A Language and Toolset for Program Transformation. Sci. Comput. Program. 72(1-2), 52–70 (2008)
7. Cordy, J.R., Dean, T.R., Malton, A.J., Schneider, K.A.: Source Transformation in Software Engineering using the TXL Transformation System. J. Info. and Software Tech. 44(13), 827–837 (2002)
8. Cordy, J.R.: The TXL Source Transformation Language. Sci. Comput. Program. 61(3), 190–210 (2006)

9. Cordy, J.R.: Source Transformation, Analysis and Generation in TXL. In: Proc. ACM SIGPLAN Works. on Partial Eval. and Program Manip., pp. 1–11. ACM Press, New York (2006)
10. Cordy, J.R.: The TXL Programming Language, Version 10.5. Queen's University at Kingston, Canada (2007), http://www.txl.ca/docs/TXL105ProgLang.pdf
11. Cordy, J.R., Visser, E.: Tiny Imperative Language, http://www.program-transformation.org/Sts/TinyImperativeLanguage
12. Cordy, J.R.: The TIL Chairmarks, http://www.program-transformation.org/Sts/TILChairmarks
13. Dean, T.R., Cordy, J.R., Malton, A.J., Schneider, K.A.: Agile Parsing in TXL. J. Automated Softw. Eng. 10(4), 311–336 (2003)
14. van Deursen, A., Kuipers, T.: Building Documentation Generators. In: Proc. 1999 Int. Conf. on Software Maint., pp. 40–49. IEEE Press, Los Alamitos (1999)
15. Guo, X., Cordy, J.R., Dean, T.R.: Unique Renaming of Java Using Source Transformation. In: Proc. IEEE Int. Works. on Source Code Analysis and Manip., pp. 151–160. IEEE Press, Los Alamitos (2003)
16. Holt, R.C.: An introduction to TA: The Tuple-Attribute Language. Technical report, University of Toronto (1997), http://plg.uwaterloo.ca/~holt/papers/ta-intro.htm
17. Holt, R.C.: Structural Manipulations of Software Architecture using Tarski Relational Algebra. In: Proc. Int. Working Conf. on Reverse Eng., pp. 210–219. IEEE Press, Los Alamitos (1998)
18. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Trans. Software Eng. 28(7), 654–670 (2002)
19. Moonen, L.: Generating Robust Parsers using Island Grammars. In: Proc. Int. Working Conf. on Reverse Eng., pp. 13–22. IEEE Press, Los Alamitos (2001)
20. Vinju, J., Klint, P., van der Storm, T.: Rascal: a Domain Specific Language for Source Code Analysis and Manipulation. In: Proc. Int. Working Conf. on Source Code Analysis and Manip., pp. 168–177. IEEE Press, Los Alamitos (2009)
21. Martin, J.: RSF file format. Technical report, University of Victoria (August 1999), http://strategoxt.org/Transform/RigiRSFSpecification
22. Weiser, M.D.: Program slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. University of Michigan, Ann Arbor (1979)