Excerpts from
The TXL Cookbook, Part III

Transformation & Analysis

James R. Cordy

School of Computing
Queen's University at Kingston, Canada

Transformation & Analysis

- The power 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 these next problems from the TXL Cookbook, we concentrate on *transformation and analysis* problems in three categories:
  - *Restructuring* problems
  - *Optimization* problems
  - *Static and dynamic analysis* problems
  - We only have time to look at a few *representative examples*
    - Chosen not because they are the most useful, but because they introduce new *recipes* and *paradigms*

tip: Doing the whole TXL Cookbook would take all week!

Reminder: Understanding TXL

- As we have seen, a **TXL** *"grammar"* is not really a grammar
    - It is a *functional program* for parsing the input
    - *Direct control* over parse - flexibility and generality
- Similarly, a **TXL** transformation *"rule set"* is not really a term rewriting system
    - The rules are a *functional program* for transforming the input
    - *Direct control* over traversal and strategy - flexibility and generality

tip: Think in terms of function application

Restructuring 1: Feature Reduction

- Reducing the number of features to process, known as *normalization* or *feature reduction*, is a common first step in slicing, dependency analysis, clone detection and other static analysis tasks
- In this example problem, we are to eliminate the *for loop* feature of *TIL* by replacing all *for* loops with equivalent *while* loops

- Paradigm: *Explicit patterns.*
    - Patterns in TXL are normally *fully explicated* with their parts rather than later taken apart piece by piece
    - This exploits the power of the *parser* to infer from source

```
rule main
   replace [statement*]
      for Id [id] := Expn1 [expression] to Expn2 [expression] do
         Statements [statement*]
      'end
      MoreStatements [statement*]
```

Restructuring 1: Feature Reduction (cont'd)

- Paradigm: *Raising the scope of application.*
  - Even though we plan to replace a single *for* statement, we need to replace it with *multiple statements*, so we have targeted the statement subsequence it is in
  - This is because **TXL** rules are strongly typed – the grammatical type of the *replacement* must be the same as the *pattern*
  - When targeting subsequences, we must remember to allow for the *tail* of the sequence – otherwise the pattern matches only the subsequence consisting solely of the statement at the *end*

```
replace [statement*]
    'for Id [id] := Expn1 [expression] 'to Expn2 [expression] 'do
        Statements [statement*]
    'end
    MoreStatements [statement*]
```

tip: Always remember to include the tail of a sequence!

tip: If you can't make the replacement you want, go up a level

Restructuring 1: Feature Reduction (cont'd)

- Paradigm: *Generating unique new identifiers.*
  - The while loop replacement needs to make a unique new identifier (*Upperid*) for each loop conversion
  - Use the TXL built-in uniquifier function [!]

```
construct UpperId [id]
   Id [_ 'upper] [!]   % unique new identifier, e.g., i_upper24
```

Restructuring 1: Feature Reduction (cont'd)

```
% Convert all TIL "for" statements to "while" form
% Jim Cordy, August 2005

include "TIL.grm"
#pragma -comment

rule main
   replace [statement*]
      'for Id [id] := Expn1 [expression] 'to Expn2 [expression] 'do
         Statements [statement*]
      'end MoreStatements [statement*]

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

   construct IterateStatement [statement]
      Id := Id + 1;

   by
      'var Id;
      Id := Expn1;
      'var UpperId;
      UpperId := (Expn2) + 1;
      'while Id <= UpperId 'do
         Statements [. IterateStatement]    % append to sequence
      'end
      MoreStatements
end rule
```

tip: TXL programmers often like to quote all target language keywords for readability

Restructuring 2: Local-to-Global

- All transformation tools need to be able to move things about, and a particular challenge is transformations at an *outer level* that depend on an *inner level* and vice-versa – we'll do an example of each
- In the first one, assuming that *TIL* is scopeless, we want to move all embedded declarations to the global level
- The basic strategy involves three steps: *extract* a copy of all embedded declarations, make a *copy* of the program with all declarations removed, then *concatenate* the two in the result

```
function main
    % This is a function transformer, so it applies only once
    replace [program]
        Program [statement*]
    construct Declarations [statement*]
        _ [^ Program] [removeNonDeclarations]
    construct ProgramSansDeclarations [statement*]
        Program [removeDeclarations]
    by
        Declarations [. ProgramSansDeclarations]
end function
```

Restructuring 2: Local-to-Global (cont'd)

- Extracting all declarations involves two paradigms in this case – *type extraction* and *type filtering*
- We first extract all the *statements* in the program, then filter out *non-declarations* (yes, we could have extracted declarations directly)
- Paradigm: *Extracting all instances of a type.*
  - The extract built-in function [^] is applied to a sequence [T*] of some type [T], and given any bound variable V as parameter returns a sequence of all the [T]s embedded in V
  - Normally begin with an empty sequence "_"
  - Extract is a powerful notion commonly used in **TXL**

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


Restructuring 2: Local-to-Global (cont'd)

- Paradigm: *Filtering all instances of a type.*
  - We need to *filter* embedded declarations out of the result
  - Uses the **TXL** filtering paradigm – look for each subsequence beginning with something we *don't want*, and replace it with the *tail* only

```
rule removeDeclarations
    % Remove every declaration at every level from statements
    replace [statement*]
        Declaration [declaration]
        FollowingStatements [statement*]
    by
        FollowingStatements
end rule
```

Restructuring 2: Local-to-Global (cont'd)

- Paradigm: *Negative patterns.*
  - For the sequence of statements we extracted, we need to filter out those statements that are *not* declarations
  - We use a *negated deconstructor* as a guard
  - A normal deconstructor matches a bound variable to a pattern and binds the parts to variables

```
deconstruct Statement
    Declaration [declaration]
```

  - We are interested statements that are not a declaration, so we negate the condition

```
deconstruct not Statement
    Declaration [declaration]
```

tip: A negated deconstructor succeeds if it does not match – so nothing is bound!

Restructuring 2: Local-to-Global (cont'd)

```
% Make TIL global variable scope explicit by promoting declarations
% Jim Cordy, October 2005
. . .
function main
    replace [program]
        Program [statement*]
    construct Declarations [statement*]
        _ [^ Program] [removeNonDeclarations]
    construct ProgramSansDeclarations [statement*]
        Program [removeDeclarations]
    by
        Declarations [. ProgramSansDeclarations]
end function
. . .
rule removeNonDeclarations
    replace [statement*]
        NonDeclaration [statement]
        FollowingStatements [statement*]
    % Check that the statement isn't a declaration
    deconstruct not NonDeclaration
        _ [declaration]
    by
        FollowingStatements
end rule
```

Restructuring 3: Global-to-Local

- The other half of the movement challenge is transformations on an *inner* level that depend on things from an *outer* level
- In this next problem, we assume that *TIL* is a *scoped* language, and move all declarations of variables that are artificially global and *localize* them to the deepest inner scope in which they are used
- Our solution will involve two steps:
  - *immediatize* declarations by moving them to just before the first statement in their scope that uses them, and
  - *localize* declarations by moving those just before a block inside if they are not used after it

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

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

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

Restructuring 3: Global-to-Local (cont'd)

- Paradigm: *Transforming to a fixed point.*
  - Because declarations may be more than one level too global, and each step may create *new opportunities* for the other, the process must be continued to a *fixed point*
  - Use a deconstructor as an explicit *fixed-point test* in the main rule

```
rule main
    % Pattern matches result, so has no natural termination point
    replace [program]
        Program [program]

    % Add an explicit fixed-point guard – after each set of
    % transformations, check to see if anything changed
    construct NewProgram [program]
        Program [immediatizeDeclarations]
                [localizeDeclarations]

    deconstruct not NewProgram
        Program
    by
        NewProgram
end rule
```

tip: If your transformation doesn't stop, check to see if anything changed

Restructuring 3: Global-to-Local (cont'd)

- Paradigm: *Dependency sorting.*
  - The *immediatize* step works by iteratively moving declarations over statements that do not depend on them – a *dependency sort*
  - This is a common paradigm in **TXL** we will see again

```
rule immediatizeDeclarations
    % Move declarations past statements that don't depend on them
    % Use a one-pass search ($) since two declarations in a row
    % has no natural fixed point
    replace $ [statement*]
        'var V [id];
        Statement [statement]
        MoreStatements [statement*]

    % Can move the declaration if the statement doesn't refer to it
    deconstruct not * [id] Statement
        V
    by
        Statement
        'var V;
        MoreStatements
end rule
```

Restructuring 3: Global-to-Local (cont'd)

- Paradigm: *Deep pattern match.*
  - The constraint under which a declaration can be moved is that the statement does not *contain* any references to it
  - Use TXL's *deep deconstruct*

```
rule immediatizeDeclarations
    % Move declarations past statements that don't depend on them
    % Use a one-pass search ($) since two declarations in a row
    % has no natural fixed point
    replace $ [statement*]
       'var V [id];
       Statement [statement]
       MoreStatements [statement*]
    % Can move the declaration if the statement doesn't refer to it
    deconstruct not * [id] Statement
        V
    by
        Statement
        'var V;
        MoreStatements
end rule
```

Restructuring 3: Global-to-Local (cont'd)

- Paradigm: *One pass rules.*
  - The case of two declarations in a row has no fixed point
  - Use **TXL**'s *one pass search*

```
rule immediatizeDeclarations
    % Move declarations past statements that don't depend on them
    % Use a one-pass search ($) since two declarations in a row
    % has no natural fixed point
    replace $ [statement*]
       'var V [id];
       Statement [statement]
       MoreStatements [statement*]
    % Can move the declaration if the statement doesn't refer to it
    deconstruct not * [id] Statement
        V
    by
        Statement
        'var V;
        MoreStatements
end rule
```

Restructuring 3: Global-to-Local (cont'd)

- Paradigm: *Multiple transformation cases.*
  - The other half of the solution is *localize*, which moves a declaration preceding a block statement such as *if, while, for* inside if following statements don't depend on it
  - Involves *multiple cases* in the result
  - In **TXL**, use one *subrule for each case*, all applied to the scope
  - **TXL** rules are *total functions* - if they don't match they return a copy of their original scope as result

tip: Composed multiple rules are TXL's if-then-else

Restructuring 3: Global-to-Local (cont'd)

```
rule localizeDeclarations
    % Move declarations outside a structured statement inside if
    % statements following it do not depend on the declared variable
    replace $ [statement*]
        Declaration [declaration]
        CompoundStatement [statement]
        MoreStatements [statement*]
    % Check it is a compound statement (with statement list inside)
    deconstruct * [statement*] CompoundStatement
        _ [statement*]
    % Check the following statements don't depend on the declaration
    deconstruct * [id] Declaration
        V [id]
    deconstruct not * [id] MoreStatements
        V
    % This solution does each kind of compound statement separately -
    % another solution might use agile parsing to abstract them
    by
        CompoundStatement [injectDeclarationWhile Declaration]
                          [injectDeclarationFor Declaration]
                          [injectDeclarationIfThen Declaration]
                          [injectDeclarationIfElse Declaration]
        MoreStatements
end rule
```

Restructuring 3: Global-to-Local (cont'd)

- Paradigm: *Context-dependent transformation rules.*
  - The *Declaration* to be inserted into the *CompoundStatement* is passed into the transformation function as a *parameter*
  - TXL rule parameters allow us to carry context from *outer* scopes into rules that transform *inner* scopes – the paradigm for *context-dependent* transformation rules

```
function injectDeclarationWhile Declaration [declaration]
    % The while Expn can't depend on the declaration, since
    % there are no assignments between the declaration and the while
    replace [statement]
        'while Expn [expression] 'do
            Statements [statement*]
        'end
    by
        'while Expn 'do
            Declaration
            Statements
        'end
end function
```

Restructuring 3: Global-to-Local (cont'd)

```
% Localize TIL declarations
% Jim Cordy, October 2005
 . . .
rule main
    % apply to fixed point
    replace [program]
        Program [program]
    construct NewProgram [program]
        Program
            [immediatizeDeclarations]
            [localizeDeclarations]
    deconstruct not NewProgram
        Program
    by
        NewProgram
end rule

rule immediatizeDeclarations
    % iterative dependency sort
     . . .
end rule

rule localizeDeclarations
    % insert if only used inside
     . . .
end rule
```

```
function injectDeclarationWhile
            Declaration [declaration]
    replace [statement]
        'while Expn [expression] 'do
            Statements [statement*]
        'end
    by
        'while Expn 'do
            Declaration
            Statements
        'end
end function

function injectDeclarationFor
            Declaration [declaration]
    . . .
end function

function injectDeclarationIfThen
            Declaration [declaration]
    . . .
end function

function injectDeclarationIfElse
            Declaration [declaration]
    . . .
end function
```

Restructuring 4: Goto Elimination

- The flagship of all restructuring problems – infer structured code from spaghetti-coded *goto* statements (*Cobol*, etc.)
- Next example: in a dialect of *TIL* with *goto* statements, recognize and convert equivalent *while* statements

```
var n; var f;
  write "Input n please"; read n;
  write "The factors of n are";
  f := 2;
factors:
  if n = 1 then
    goto endfactors;
  end;
multiples:
  if (n / f) * f != n then
    goto endmultiples;
  end;
  write f;
  n := n / f;
  goto multiples;
endmultiples:
  f := f + 1;
  goto factors;
endfactors:
```

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

Restructuring 4: Goto Elimination (cont'd)

- The basic solution we will use is to *catalogue* patterns of use we observe, encode them as *patterns*, and use one rule per pattern to replace them by their equivalent structures
- In practice we would first run a goto *normalization* (*feature reduction*) transformation to reduce the number of cases

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


Restructuring 4: Goto Elimination (cont'd)

- Paradigm: *Matching a gapped subsequence.*
  - By now the idea of making a subsequence pattern in **TXL** should be familiar – you might write the pattern below, remembering the trailing *Rest [statement*]* as usual
  - Unfortunately, this pattern cannot be parsed, because **TXL** is strongly typed, and it is not an instance of *[statement*],* defined as *[statement] [statement*] | [empty]*

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

tip: Don't give up, some patterns come in parts!

Restructuring 4: Goto Elimination (cont’d)

- Paradigm: *Matching a gapped subsequence (cont’d).*
  - What we need to do is to match the *first part* we want, and then match the *last part* we want separately
  - The trick here is not to look *inside* the intervening statements when looking for the last part
  - In TXL, we express this constraint using *skipping deep deconstruct,* which limits the deep match to the top level
  - For example, this deconstructor only matches if we can find the *goto back* and the *ending label* at the same level (i.e., without looking inside any of the *statements* in the sequence)

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

Restructuring 4: Goto Elimination (cont'd)

- Paradigm: *Truncating the tail of a sequence.*
  - The other paradigm this solution needs is a way to truncate the tail of a sequence, so we can use the *subsequence* of statements before a *goto* as the body of the structured statement
  - The **TXL** paradigm for truncation is used in the function below, which given the labels involved, searches a statement sequence for the *end of loop pattern* and *deletes it and what follows*

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

tip: Notice we need skipping stay at the same level

Restructuring 4: Goto Elimination (cont'd)

```
% Goto elimination in TIL programs
% Jim Cordy, January 2008
 . . .
rule transformForwardWhile
   replace [statement*]
     L0 [label] ':
      'if C [expression] 'then
         'goto L1 [label];
      'end;
      Rest [statement*]
   skipping [statement] deconstruct * Rest
      'goto L0;
     L1 ':
      Follow [statement]
      FinalRest [statement*]
   construct LoopBody [statement*]
      Rest [truncateGoto L0 L1]
   by
      'while !(C) 'do
         LoopBody
      'end;
      Follow
      FinalRest
end rule
 . . .
```

Optimization 1: Code Motion

- Source transformation tools are often used in *code optimization* tasks, and in this part we look at **TXL** solutions in that domain
- In the next set of problems we attack some classical source level optimization tasks, and observe some new **TXL** *paradigms*
- Our first task is *code motion*, the movement of statements independent of a loop outside it

```
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;
```

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

Optimization 1: Code Motion (cont'd)

- Paradigm: *Guarding a transformation with a complex condition.*
  - The most difficult part, since we have already seen *restructuring,* is how to be sure an assignment is *independent* of the loop
  - In TXL we encode such complex guards as *where* statements, which use other *condition rules* to evaluate stronger patterns

```
% We can only lift an 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 the effect of it does not wrap around the loop
construct PreContext [statement*]
    Body [deleteAssignmentAndRest X]
where not
    PreContext [refers X]
```

Optimization 1: Code Motion (cont'd)

- Paradigm: *Each element of a sequence.*
  - The first where condition uses another TXL paradigm – the *each* modifier
  - Each takes a sequence *[X*]* of any type *[X]*, and calls the subrule once with *each element* of the sequence as parameter
  - So if *IdsInExpression* is bound to the identifiers *a b c,* then *[assigns each IdsInExpression]* means three calls to the subrule, *[assigns 'a] [assigns 'b] [assigns 'c]*

```
where not
    Loop [assigns each IdsInExpression]

% Given a scope, does the scope assign to the identifier?
function assigns Id [id]
    match * [assignment_statement]
        Id := Exppn [expression];
end function
```

Optimization 1: Code Motion (cont'd)

- Paradigm: *Using a transformation rule as a condition.*
  - The main rule of this transformation simply finds every *while* loop, extracts all the *assignment statements* in it by making a copy of the loop body and *filtering out* statements that are not assignments, then uses the subrule *[loopLift]* to try to move each one outside the loop
  - Instead of a fixed-point rule, the main rule simply asks *[loopLift]* itself in advance if it is going to work, using *[?loopLift]* as a guard

```
% Can loopLift succeed?
where
    LiftedLoop [?loopLift Body each AllAssignments]
```

tip: You can ask any set of transformations if any of them will succeed, before committing to applying them

Optimization 1: Code Motion (cont'd)

```
% Lift independent assignments outside of TIL while loops
% J.R. Cordy, November 2005
 . . .
rule main
    replace [statement*]
        while Expn [expression] do
            Body [statement*]
        'end
        Rest [statement*]

    % Construct a list of all the top-level assignments in it
    construct AllAssignments [statement*]
        Body [deleteNonAssignments]

    % Construct 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
    where
        LiftedLoop [?loopLift Body each AllAssignments]

    % If the guard succeeds, some can be moved out, so go ahead
    by
        LiftedLoop [loopLift Body each AllAssignments]
        [. Rest]
end rule
```

Optimization 1: Code Motion (cont'd)

```
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*]

    % Can only lift if ids in the expn 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 the effect of it does not wrap around the loop
    construct PreContext [statement*]
        Body [deleteAssignmentAndRest X]
    where not
        PreContext [refers X]

    by
        Assignment
        Loop [deleteAssignment Assignment]
end function
```

Optimization 2: Common Subexpressions

- *Common subexpression elimination* is a well known traditional optimization technique that avoids recalculating expressions
- In this example problem, we are to *recognize* and *optimize* repeated expressions that do not change in *TIL* programs

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

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

Optimization 2: Common Subexpressions (cont'd)

- The problem has much in common with *code motion*, because again we must check that variables used in the expression *do not change*

```
% What we're looking for is an expression ...
deconstruct * [expression] S1
    E [expression]
% ... that is nontrivial ...
deconstruct * [op]  E
    _ [op]
% ... and repeated
deconstruct * [expression] SS
    E

% See if we can abstract it (checks if variables assigned)
where
    SS [?replaceExpnCopies S1 E 'T]

% If so, generate a new temporary variable name ...
construct T [id]
    _ [+ "t"] [!]

% ... declare it, assign the common expression, and replace instances
by
    'var T;     'NEW
    T := E;     'NEW
    S1 [replaceExpn E T]
    SS [replaceExpnCopies S1 E T]
```

Optimization 2: Common Subexpressions (cont'd)

- Paradigm: *Marking using attributes.*
  - A common problem in source transformation rules is avoiding reconsidering what's already done, in this case *generated code*
  - In TXL we often do this using *marking attributes*, marks that attribute the *result* but do not appear in the *output*

```
% Allow statements marked NEW
redefine statement
    ...
  | [statement] [attr 'NEW]
end redefine
```

tip: Attributes can be any nonterminal type and carry any information

```
rule main
   replace [statement*]
      S1 [statement]
      SS [statement*]
   % Don't process generated statements
   deconstruct not * [attr 'NEW] S1
      'NEW
    . . .
   by
      'var T;     'NEW
      T := E;     'NEW
      S1 [replaceExpn E T]
      SS [replaceExpnCopies S1 E T]
end rule
```

Optimization 2: Common Subexpressions (cont'd)

- Paradigm: *Grammatical form abstraction.*
  - When we want to treat *different* forms in the *same* way in a particular task, we can use *agile parsing* tp abstract the equivalent cases into one

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

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

 . . .

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

tip: Type abstraction can simplify and clarify rules without affecting the original grammar

Optimization 2: Common Subexpressions (cont'd)

- Paradigm: *Tail-recursive continuation.*
  - If we want to continue a transformation in a *sequence*, but only as long as some *condition* holds, use *tail recursion*

```
function replaceExpnCopies PrevS [statement] E [expression] T [id]
   construct Eids [id*]
      _ [^ E]
   % If the previous statement did not assign any variable in E
   where not
      PrevS [assigns each Eids]
   % Then we can continue to substitute in the next statement ...
   replace [statement*]
      S [statement]
      SS [statement*]
   % ... as long as it isn't a compound statement that assigns
   % 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
```

Optimization 2: Common Subexpressions (cont'd)

- Paradigm: *Guarding with multiple conditions.*
    - As we've seen in previous paradigms, *where* clauses normally check whether *any* of the condition rules matches
    - *where all* can be used to check whether *all* of the conditions apply and *where not all* whether *not all* apply

```
% ... as long as it isn't a compound statement that assigns
% one of the variables in the expression
where not all
    S [assignsOne Eids]
      [isCompoundStatement]
```

Optimization 2: Common Subexpressions (cont'd)

```
% TXL transformation to recognize and optimize common subexpressions
% Jim Cordy, March 2007

 . . .

rule main
   replace [statement*]
      S1 [statement]
      SS [statement*]
   deconstruct not * [attr 'NEW] S1
      'NEW
   deconstruct * [expression] S1
      E [expression]
   deconstruct * [op]  E
      _ [op]
   deconstruct * [expression] SS
      E
   where
      SS [?replaceExpnCopies S1 E 'T]
   construct T [id]
      _ [+ "temp"] [!] \vspace{0.2cm}
   by
      'var T;    'NEW
      T := E;    'NEW
      S1 [replaceExpn E T]
      SS [replaceExpnCopies S1 E T]
end rule
```

Optimization 2: Common Subexpressions (cont'd)

```
function replaceExpnCopies PrevS [statement] E [expression] T [id]
   construct Eids [id*]
      _ [^ E]
   where not
      PrevS [assigns each Eids]
   replace [statement*]
      S [statement]
      SS [statement*]
   where not all
      S [assignsOne Eids]
        [isCompoundStatement]
   by
      S [replaceExpn E T]
      SS [replaceExpnCopies S E T]
end function

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

 . . .
```

Optimization 3: Constant Folding

- Constant *propagation* (replacing assign-once variables by their values) and *folding* (evaluating expressions with constant operands) is a common optimization transformation
- At this point we require no new paradigms to solve it using **TXL** – we simply use one rule for each, and combine the two using the *fixed-point* paradigm

```
% Constant propagation and folding for TIL
% Jim Cordy, January 2008
 . . .
rule main
   replace [program]
      P [program]
   construct NewP [program]
      P [propagateConstants]
        [foldConstantExpressions]
   deconstruct not NewP
      P
   by
      NewP
end rule
```


Optimization 3: Constant Folding (cont'd)

```
% Constant propagation - find each constant assignment,
% and if not assigned again then replace references
rule propagateConstants
   replace [repeat statement]
      Var [id] := Const [literal] ;
      Rest [repeat 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
```

tip: Remember the paradigm: to replace one type with another, target their common ancestor

Optimization 3: Constant Folding (cont'd)

```
% Constant folding - find and evaluate constant expressions
rule foldConstantExpressions
   replace [expression]
      E [expression]
   construct NewE [expression]
      E [resolveAddition]
        [resolveSubtraction]
        [resolveMultiplication]
        [resolveDivision]
        % Other special cases involving constants
         . . .
   % Continue to fixed point
   deconstruct not NewE
      E
   by
      NewE
end rule

rule resolveAddition
   replace [expression]
      N1 [integernumber] + N2 [integernumber]
   by
      N1 [+ N2]
end rule
```

Optimization 4: Statement Folding

- Statement folding is the elimination of *statements* that cannot be reached because their guarding condition is constant or known
- Goes hand-in-hand with *constant folding*
- Similar transforms used in *conditional compilation*, *partial evaluation*
- In this problem we implement statement folding for *TIL* using **TXL**

```
 . . .
x := 5;
 . . .
if x != 5 then
    y := x;
    write "x case";
else
    y := Z;
    write "z case";
end;
 . . .
```

```
 . . .
x := 5;
 . . .
y := Z;
write "z case";
. . .
```

Optimization 4: Statement Folding (cont'd)

- Paradigm: *Concatenating sequences.*
  - As usual, we the *[statement*]* sequence of the statement to fold
  - But we cannot simply replace with *two sequences* (why?)
  - Need TXL sequence concatenate function *[.]* to join

```
rule foldTrueIfStatements
   % For each if statement
   replace [statement*]
      'if Cond [expression] 'then
         TrueStatements [statement*]
      ElseClause [else_statement?]
      'end
      Rest [statement*]
   % That has a constant true condition
   where
      Cond [isTrueEqual]
           [isTrueNotEqual]
   % By the true part
   by
      '// Folded true if
      TrueStatements [. Rest]
end rule
```

tip: Always use [.] to add a tail to a sequence

Optimization 4: Statement Folding (cont'd)

- Paradigm: *Handling optional parts.*
  - When a rule involves an *optional* part, we must be careful to handle *both* the case where it is present and the case not
  - In TXL this is done using a *construct* with a *subrule*

```
rule foldFalseIfStatements
   replace [statement*]
      'if Cond [expression] 'then
         TrueStatements [statement*]
      ElseClause [else_statement?]
      'end
      Rest [statement*]
   where not
      Cond [isTrueEqual]
           [isTrueNotEqual]
   construct FalseStatements [statement*]
      _ [getElseStatements ElseClause]
   by
      '// Folded true if
      TrueStatements [. Rest]
end rule
```

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

Optimization 4: Statement Folding (cont'd)

```
% Statement folding for TIL
% Jim Cordy, January 2008
 . . .
function main
   replace [program]
      P [program]
   by
      P [foldTrueIfStatements]
        [foldFalseIfStatements]
end function

rule foldTrueIfStatements
   replace [statement*]
      'if Cond [expression] 'then
         TrueStatements [statement*]
      ElseClause [else_statement*]
      'end
      Rest [statement*]
   where
      Cond [isTrueEqual]
           [isTrueNotEqual]
   by
      '// Folded true if
      TrueStatements [. Rest]
end rule
```

```
 . . .
function getElseStatements
        ElseClause [else_statement?]
   deconstruct ElseClause
      'else
         FalseStatements [statement?]
   replace [statement*]
      % none
   by
      FalseStatements
end function

 . . .
function isTrueEqual
   match [expression]
      X [number] = Y [number]
   where
      X [= Y]
end function
 . . .
```

tip: When preserving comments, easy to create new ones in the result

Analysis 1: Program Statistics

- Source transformation tools are often used in *static and dynamic analysis*, and in this last part we look at **TXL** solutions in that domain
- In the next set of problems we attack some classical *source analysis* tasks, and observe the **TXL** *recipes*
- Our first task is a simple example of a *code metrics* analysis, *statement statistics* for *TIL* programs – a trivial case of a more general problem, step 1 of a software *portfolio assessment*

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

Analysis 1: Program Statistics (cont'd)

- Paradigm: *Counting feature instances.*
  - A general TXL paradigm that combines *agile parsing* (to abstract the classes of items we are interested in), *type extraction*, and *type filtering* to yield all instances of a feature
  - Could then use normalizing transformations to abstract into common *patterns of use*, but in this case just *count*

```
% Get all one-way ifs in the program
construct SimpleIfs [if_statement*]
    _ [^ Program] [filterIfElses]
% Count them
construct SimpleIfCount [number]
    _ [length SimpleIfs] [putp "One-way ifs: %"]
```

tip: Use [length X] to count elements of a sequence

Analysis 1: Program Statistics (cont'd)

- Paradigm: *Dynamic output.*
    - **TXL** allows for *error stream* output while transforming e.g., for *tracing*, *error messages*, etc.
    - Can be used to dynamically print out auxiliary output – of *any type,* to either the standard error stream or a *named file*

```
% Count them
construct SimpleIfCount [number]
    _ [length SimpleIfs] [putp "One-way ifs: %"]
```

tip: [putp] is kind of like printf() in C, but can print anything

Analysis 1: Program Statistics (cont'd)

```
% Statement statistics for Tiny Imperative Language programs
% Jim Cordy, October 2005

include "TIL.grm"

function main
   replace [program]
      Program [program]

   % Count each kind of statement we're interested in
   % (could use polymorphics to do this generically)

   construct Statements [statement*]
       _ [^ Program]
   contruct 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: %"]

    . . .

   by
      % nothing
end function
```

Analysis 2: Dynamic Tracing

- Addition of auxiliary monitoring code to a program for *dynamic analysis* or *run-time monitoring* is a very common transformation task
- In this example, the problem is to use **TXL** to transform a *TIL* program to a *self-tracing* version of itself, one that prints out the text of each statement just before it is executed

```
 write "Trace: var n;";
var n;
 write "Trace: write \"Input n please\";";
write "Input n please";
 write "Trace: read n;";
read n;
 write "Trace: write \"The factors of n are:\";";
write "The factors of n are:";
 write "Trace: var f;";
var f;
 write "Trace: f := 2;";
f := 2;
 write "Trace: while n != 1 do ... end;";
while n != 1 do
  . . .
```

Analysis 2: Dynamic Tracing

- Addition of auxiliary monitoring code to a program for *dynamic analysis* or *run-time monitoring* is a very common transformation task
- In this example, the problem is to use **TXL** to transform a *TIL* program to a *self-tracing* version of itself, one that prints out the text of each statement just before it is executed

```
 Trace: var n;
 Trace: write "Input n please";
Input n please
 Trace: read n;
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;
   . . .
```

Analysis 2: Dynamic Tracing (cont'd)

- Paradigm: *Eliding detail.*
  - Sometimes we want to be able to *summarize* a form when reporting or tracing
  - This is easy in TXL using *agile parsing* and a *filtering* rule

```
% Allow for concise elided structured statements
redefine statement
        ...
    |   '...
end redefine

% Replace any embedded statement sequence with an elision symbol
% for conciseness in the trace
function deleteBody
   replace * [statement*]
      _ [statement*]
   by
      '...
end function
```

Analysis 2: Dynamic Tracing (cont'd)

- Paradigm: *Converting program fragments to text strings.*
  - Construction of the program text fragments as strings uses the **TXL** text manipulation built-in functions *[+]* (text concatenation) and *[quote]* (convert parsed item to output text)
  - Constructing lexical text of any terminal type can use the same paradigm, beginning with an *empty* text and *concatenating* pieces

```
% Make a concise version of structured statements
construct ConciseS [statement]
    S [deleteBody]
% Make trace output string for the statement
construct QuotedS [stringlit]
    _ [+ "Trace: "] [quote ConciseS]
```

tip: TXL built-in text functions can be used to directly manipulate text of any token type

Analysis 2: Dynamic Tracing (cont'd)

```
% Make a TIL program self-tracing
% Jim Cordy, August 2005

include "TIL.grm"
#pragma -esc "\"

redefine statement
     ...
  | '...
end redefine

redefine statement
    ...
  | [traced_statement]
end redefine

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

```
rule main
    replace [statement*]
        S [statement]
        Rest [statement*]
    deconstruct not S
        _ [statement]  'TRACED
    construct ConciseS [statement]
        S [deleteBody]
    construct QuotedS [stringlit]
        _ [+ "Trace: "] [quote ConciseS]
    by
        'write QuotedS;    'TRACED
        S                  'TRACED
        Rest
end rule

function deleteBody
    replace * [statement*]
        _ [statement*]
    by
        '...
end function
```

Analysis 3: Type Inference

- Type inference is an example of a classical *inductive inference* problem – other examples are *dead code* detection, *call graph* analysis, *aspect* analysis, *business concept* tracing (*Y2K*), many others
- In this example problem, we are to infer the *static types* of all variables in a *TIL* program, and detect those that are inconsistent – for example, as part of a migration from a dynamically to statically typed language (e.g., *Python* to *Java*)
- Basically, this is an attribution problem, and the solution is an inductive implementation of *derived attributes* in **TXL**

Analysis 3: Type Inference (cont'd)

- The TXL solution will work in *five steps*:
  - *Normalize* to attributed primary expressions
  - Use *local induction* to attribute expressions to a fixed point
  - *Extract* inferred type attributes to declarations
  - *Report* conflicted inferences
  - *Denormalize* to original expressions

Analysis 3: Type Inference (cont'd)

- The purpose will be to allow all variables to be explicitly typed, so we begin by extending *TIL* to have types on variable declarations

```
% Grammar overrides to allow for type specs on variable declarations
redefine declaration
    'var [primary] [colon_type_spec?] ';    [NL]
end redefine

define colon_type_spec
    ': [type_spec]
end define

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

Analysis 3: Type Inference (cont'd)

- The method will involve *inductively inferring types* for literals, references and expressions, so we allow for type attributes on *primary expressions*

```
% Grammar overrides to allow type attributes on primary expressions
redefine primary
    [subprimary] [attr type_attr]
end redefine

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

define type_attr
    '{ [type_spec?] '}
end define
```

Analysis 3: Type Inference (cont'd)

- Paradigm: *Program normalization.*
  - Of course, we want to infer a type for every subexpression at *every level*, not just primary expressions
  - We could add and attribute inference rules for every level, but simpler – we can *normalize* expressions to full parenthesization, reducing *all cases* to the primary expression case

```
% Complete parenthesization to allow full type attribution
rule bracketExpressions
   skipping [expression]
   replace [expression]
      E1 [expression] Op [op] E2 [expression]
   by
      '( E1 Op E2 ')
end rule

rule unbracketExpressions
   skipping [expression]
   replace [expression]
      '( E1 [expression] Op [op] E2 [expression] ') '{ _ [type_spec] '}
   by
      E1 Op E2
end rule
```

Analysis 3: Type Inference (cont'd)

- Paradigm: *Grammatical form generalization.*
  - We *generalize* all other variable references to be a *primary expression*, effectively moving entirely into the transformation's own *typed primary expression* concept
  - This is **TXL** *agile parsing* – generalizing to a parse that better matches the transformation's own concepts

```
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
```

note: automatically allows for type attributes in all these contexts

Analysis 3: Type Inference (cont'd)

- Paradigm: *Inductive transformation.*
  - The actual inference then works using an *inductive transformation*, where the *base case* is attribution of literal values, and the *induction step* is a set of simple local inference rules
  - Under control of the usual **TXL** *fixed-point paradigm*

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

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

```
rule attributeOperations
  replace [primary]
    ( P1 [subprimary] {Type [type_spec] } Op [op] P2 [subprimary] {Type} )
    { }
  by
    ( P1 {Type} Op P2 {Type} ) {Type}
end rule
```

Analysis 3: Type Inference (cont'd)

```
% Infer types for TIL vars and expns
% Jim Cordy, March 2007
 . . .
function main
   replace [program]
      P [program]
   by
      P [bracketExpressions]
        [enterDefaultAttributes]
        [attributeStringConstants]
        [attributeIntConstants]
        [attributeProgramToFixedPoint]
        [completeWithUnknown]
        [typeDeclarations]
        [reportErrors]
        [unbracketExpressions]
end function

rule bracketExpressions
   skipping [expression]
   replace [expression]
     E1 [expression]
       Op [op] E2 [expression]
   by
      '( E1 Op E2 ')
end rule
 . . .
```

```
rule attributeProgramToFixedPoint
    replace [program]
        P [program]
    construct NP [program]
        P [attributeAssignments]
          [attributeOperations]
          [attributeForIds]
          [attributeDeclarations]
    deconstruct not NP
         P
    by
         NP
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
```

Analysis 3: Type Inference (cont'd)

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

```
rule completeWithUnknown
   replace [attr type_attr]
      { }
   by
      { UNKNOWN }
end rule
 . . .
rule reportErrors
    replace $ [statement]
       S [statement]
    skipping [statement*]
    deconstruct * [type_spec] S
        'UNKNOWN
    construct Message [statement]
        S [pragma "-attr"]
          [message "*** ERROR: Unable
                to resolve types in:"]
          [stripBody]
          [putp "%"]
          [pragma "-noattr"]
    by
         S
end rule
```

Analysis 4: Backward Slicing

- Dependency analysis is an important and common static analysis technique, and *static slicing* is a common example
- In this problem, we are to use **TXL** to compute the *backward slice* of a *TIL* program given a particular statement (and its variables) as criterion
- *Reminder*: a *backward slice* is an executable subset of the statements in a program that preserves the observable values of the original program at a particular statement

Analysis 4: Backward Slicing (cont'd)

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

```
var lines;
lines := 0;



var eof_flag;
read eof_flag;

while !eof_flag do
    lines := lines + 1;

    read eof_flag;

end;
write (lines);
```

Analysis 4: Backward Slicing (cont'd)

- Paradigm: *Cascaded markup.*
  - The basic strategy is simple: an assignment or change to a variable is in the slice if any *subsequent use* already in the slice
  - Use agile parsing to allow *markup* of statements, *generalize* all loop statement forms to one form

```
% Rule to back-propagate markup of assignments – similar for reads
rule backPropagateAssignments
    skipping [marked_statement]
    replace [statement*]
        X [id] := E [expression] ;
        More [statement*]
    where
        More [hasMarkedUse X]
    by
        <mark> X := E; </mark>
        More
end rule
```

tip: Continue markup to fixed point

Analysis 4: Backward Slicing (cont'd)

```
% Backward slicing of TIL programs
% Jim Cordy, February 2007
 . . .
function main
   replace [program] P [program]
   by P [propagateMarkupToFixedPoint]
        [removeUnmarkedStatements]
        [removeRedundantDeclarations]
        [stripMarkup]
end function

Rule propagateMarkupToFixedPoint
   replace [program]
      P [program]
   construct NP [program]
      P [backPropagateAssignments]
        [backPropagateReads]

[whilePropagateControlVariables]
        [loopPropagateMarkup]
        [loopPropagateMarkupIn]
        [ifPropagateMarkupIn]
        [compoundPropagateMarkupOut]
   deconstruct not NP
      P
   by
      NP
end rule
```

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

rule loopPropagateMarkup
   replace $ [statement]
      Head [loop_head]
         S [statement*]
      'end;
   construct MarkedS [marked_statement*]
      _ [^ S]
   construct MarkedE [expression*]
      _ [^ MarkedS]
   by
      Head
         S [markAssignmentsTo each
MarkedE]
           [markReadsOf each MarkedE]
      'end;
end rule
```

Analysis 5: Clone Detection

- Clone detection is a popular and interesting source analysis task with a wide range of applications, including *code reduction* and *refactoring*
- Can vary in *granularity*, from statements to functions to classes
- In this example problem, we demonstrate the basic **TXL** recipe for *clone detection* using *TIL* structured statements as the clone units

- As usual, the **TXL** solution is multi-step:
  - *Extract* a sequence of all instances of structured statements in the program (the *"potential clones"*)
  - Optionally *normalize for comparison* *(e.g.,* by anonymizing identifiers)
  - *Filter* those that appear only once from the sequence, to yield one copy of each actual clone
  - For each clone in the sequence, *mark up* all instances in the program with its *clone class* (position in the sequence)

Analysis 5: Clone Detection (cont'd)

- Paradigm: *Precise control of output format.*
  - Using *[SPOFF], [SP], [TAB], [SPON],* can take complete control over output spacing

```
% Generalize granule concept
redefine statement
        [structured_statement]
    |   ...
end redefine

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

```
% Allow for clone class markup
redefine statement
        ...
    |   [marked_statement]
end redefine

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

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

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

Analysis 5: Clone Detection (cont'd)

- Paradigm: *Context-dependent rules.*
  - We have seen context-dependent rules before, but in this case we depend on *both* a *part* of the context *and* the *entire* context
  - As usual, context uses TXL rule parameters

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

tip: Don't assume a copy costs anything!

Analysis 5: Clone Detection (cont'd)

- Paradigm: *Accumulating multiple results.*
  - As well as using global context, the *addIfClone* rule also demonstrates the general way to accumulate multiple results, using *each* and concatenation to a *sequence*

```
function addIfClone StructuredStatements [structured_statement*]
                    Statement [structured_statement]

   % A fragment is a clone if it appears twice in all fragments
   deconstruct * StructuredStatements
      Statement
      Rest [structured_statement*]
   deconstruct * [structured_statement] Rest
      Statement

   % If it appears (at least) twice, add it to the clones
   replace [repeat structured_statement]
      StructuredClones [structured_statement*]

   % Make sure it's not already in the table
   deconstruct not * [structured_statement] StructuredClones
      Statement
   by
      StructuredClones [. Statement]
end function
```


Analysis 5: Clone Detection (cont'd)

```
% Exact clone detection for TIL
% Jim Cordy, October 2007
 . . .
function main
   replace [program]
      P [program]
   construct StructuredClones [structured_statement*]
      _ [findStructuredStatementClones P]
   export CloneNumber [number] 0
   by
      P [markCloneInstances each StructuredClones]
end function
 . . .
rule markCloneInstances StructuredClone [structured_statement]
   % Keep track of the index of this clone in the table
   import CloneNumber [number]
   export CloneNumber CloneNumber [+ 1]

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

- Paradigm: *Global state.*
  - Global *blackboard* for *shared state*

Analysis 6: Unique Renaming

- *Unique renaming* introduces *scope-independent* names for every declaration and reference in the program, as a first step in *fact generation*
  - Represents a common problem in design recovery and analysis (e.g., *Cobol* implicit qualification, *Java* scope rule resolution)
- In this example problem, we are to uniquely rename every *variable*, *function* and *module* in an *MTIL* (module dialect of *TIL*) program, using **TXL**
- Our strategy will be to find every *scope*, and add to the name of every declaration embedded in it the name of the scope (e.g., variable *x* of function *f* becomes *f.x* in its declaration and every reference)
- By proceeding inside out, from the deepest scopes to the shallowest, everything ends up fully qualified (e.g., *m.f.x* for variable *x* of function *f* in module *m*) – but *how?*

Analysis 6: Unique Renaming (cont'd)

- Paradigm: *Bottom-up traversal.*
  - The TXL paradigm for bottom-up traversal is simple: *pre-recursion*

```
rule uniqueRename
   % Do each scope on each level once
   skipping [statement] replace $ [statement]
      Scope [statement]

   % Only interested in statements that form scopes
   deconstruct * [statement*] Scope
      _ [statement*]

   % Get function, module or structure name
   construct ScopeName [id]
      _ [makeKeyName Scope] [getDeclaredName Scope]

   % Visit inner scopes first, then rename things in this one
   by
      Scope [uniqueRenameDeeper]
            [uniqueRenameScope ScopeName]
end rule

function uniqueRenameDeeper
   replace * [statement*]
      EmbeddedStatements [statement*]
   by
      EmbeddedStatements [uniqueRename]
end function
```

Analysis 6: Unique Renaming (cont'd)

- Within each scope, we find each *declaration*, add the *scope name* to its name, and rename it and all references
- Since we are working inside out, this successively adds all the *outer* scope names, giving the unique fully qualified name *p.m.f.x*

```
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]
   % Construct the new id for it from the scope id
   construct UniqueName [name]
      ScopeName '. Name
   % Rename the declaration and all its references
   % The [$ X Y] TXL built-in rule replaces every X by Y
   by
      DeclScope [$ Name UniqueName]
end rule
```

```
redefine name
    [id]
  | [id] . [name]
end redefine
```

Analysis 6: Unique Renaming (cont'd)

```
% Unique renaming for MTIL
% Jim Cordy, March 2009
 . . .
function main
 replace [program]
   P [program]
 by
   P [uniqueRenameDeeper]
     [uniqueRenameScope 'MAIN]
     [renameModulePublicReferences]

[renameFunctionFormalParameters]
end function
rule uniqueRename
   . . .
end rule
function uniqueRenameDeeper
   . . .
end function
rule uniqueRenameScope ScopeName
[id]
   . . .
end rule
```

```
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 outer scope
  by
     'module ModuleName
        ModuleStatements
     'end ;
     RestOfScope [updatePublicFunctionCall
             each
UniquePublicFunctionNames]
end rule
 . . .
rule renameFunctionFormalParameters
   . . .
end function
```

Analysis 7: Design Recovery

- Design recovery, or *fact generation*, is the extraction of facts about basic *entities* and *relationships* to an external graph or database that can be explored using tools like *Grok*, *CrocoPat*, *Rscript* or *Prolog*
- In this example problem, we are to extract basic structural and usage facts for programs written in *MTIL*, in particular *contains(), calls(), reads()*, *writes()* relationships for all modules, functions and variables

```
contains (MAIN, MAIN.maxprimes)
contains (MAIN, MAIN.maxfactor)
writes (MAIN, MAIN.maxprimes)
writes (MAIN, MAIN.maxfactor)
    . . .
contains (MAIN.flags, MAIN.flags.flagvector)
contains (MAIN.flags, MAIN.flags.flagset)
contains (MAIN.flags.flagset, MAIN.flags.flagset.f)
contains (MAIN.flags.flagset, MAIN.flags.flagset.tf)
writes (MAIN.flags.flagset, MAIN.flags.flagvector)
reads (MAIN.flags.flagset, MAIN.flags.flagset.f)
reads (MAIN.flags.flagset, MAIN.flags.flagset.tf)
    . . .
calls (MAIN, MAIN.flags.flagget)
calls (MAIN, MAIN.flags.flagset)
    . . .
```

Analysis 7: Design Recovery (cont'd)

- By now, *you* can tell *me* how to implement such a **TXL** program – begin with *unique renaming* transformation

- Paradigm: *Local fact annotation.*
  - The strategy will then be to *locally annotate* references and statements in the program with *facts* in *Prolog* syntax using contextual inference rules, then *extract* all the facts

```
module m
  . . .
  function m.f
     . . .
     writes (m.f, x.y.z)
     reads (m.f, a.b.c)
     x.y.z := a.b.c;
     . . .
   end;
   . . .
end;
```

Analysis 7: Design Recovery (cont'd)

```
% Fact extraction for MTIL
% Jim Cordy, May 2009
 . . .
include "Facts.grm"

% Allow facts on any statement
redefine statement
    [fact*] ...
end redefine

% Allow facts on any expression
redefine primary
    [fact*] ...
end redefine

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

```
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
```


Analysis 7: Design Recovery (cont'd)

```
rule addContainsFacts ScopeName [name]
   skipping [statement]
   replace $ [statement]
      Facts [fact*]
      Declaration [declaration]

   deconstruct * [name] Declaration
      DeclName [name]

   by
      'contains '( ScopeName, DeclName ')
      Facts
      Declaration
end rule

rule inferContains
   replace $ [declaration]
      ScopeDeclaration [declaration]

   deconstruct * [name] ScopeDeclaration
      ScopeName [name]

   by
      ScopeDeclaration [addContainsFacts ScopeName]
                       [addContainsParameters ScopeName]
end rule

 . . .
```

More to Come, Another Time …

- That's unfortunately all we have time for here
  - Lots more to come – in the **TXL** Cookbook

- References:

  TXL Website http://www.txl.ca

  TIL examples http://www.program-transformation.org/Sts/TILChairmarks