11. Program Transformation

Oscar Nierstrasz
Roadmap

- Program Transformation
- Refactoring
- Aspect-Oriented Programming
Links

> **Program Transformation:**
  — http://swel.tudelft.nl/bin/view/Pt
  — http://www.program-transformation.org/

> **Stratego:**
  — http://strategoxt.org/

> **TXL:**
  — http://www.txl.ca/

> **Refactoring:**
  — http://recoder.sourceforge.net/wiki/
  — http://www.refactory.com/RefactoringBrowser/

> **AOP:**
  — http://www.eclipse.org/aspectj/
Roadmap

> Program Transformation
  – Introduction
  – Stratego/XT
  – TXL

> Refactoring

> Aspect-Oriented Programming

Thanks to Eelco Visser and Martin Bravenboer for their kind permission to reuse and adapt selected material from their Program Transformation course. http://swerl.tudelft.nl/bin/view/Pt
What is “program transformation”? 

> **Program Transformation** is the process of transforming one program to another.

> Near synonyms:
  — Metaprogramming
  — Generative programming
  — Program synthesis
  — Program refinement
  — Program calculation
Applications of program transformation

> **Translation**
  - *Migration*
  - *Synthesis*
    - Refinement
    - Compilation
  - *Reverse Engineering*
    - Decompilation
    - Architecture Extraction
    - Visualization
  - *Program Analysis*
    - Control flow
    - Data flow
Translation — compilation

```
function fact(n : int) : int =
  if n < 1 then 1
  else (n * fact(n - 1))

Tiger

```

```
fact:
  subu $sp, $sp, 20
  sw $fp, 8($sp)
  addiu $fp, $sp, 20
  sw $s2, -8($fp)
  sw $ra, -4($fp)
  sw $a0, 0($fp)
  move $s2, $a1
  li $t0, 1
  bge $s2, $t0, c_0
  li $v0, 1
  b d_0

MIPS

```

```
c_0:
  lw $a0, ($fp)
  li $t0, 1
  subu $a1, $s2, $t0
  jal fact_a_0
  mul $v0, $s2, $v0

d_0:
  lw $s2, -8($fp)
  lw $ra, -4($fp)
  lw $fp, 8($sp)
  addiu $sp, $sp, 20
  jr $ra
```

[Link to lecture slides](http://www.cs.uu.nl/docs/vakken/pt/slides/PT05-ProgramTransformation.pdf)
Translation — migration from procedural to OO

Tiger

```
type tree = {key: int, children: treelist}
type treelist = {hd: tree, tl: treelist}
function treeSize(t : tree) : int =
    if t = nil then 0 else 1 + listSize(t.children)
function listSize(ts : treelist) =
    if ts = nil then 0 else listSize(t.tl)
```

Java

```
class Tree {
    Int key;
    TreeList children;
    public Int size() {
        return 1 + children.size
    }
}
class TreeList { ... }
```

Rephrasing — desugaring regular expressions

EBNF

```
Exp := Id
  | Id "(" {Exp","})* ")"
  | Exp "+" Exp
  | ...
```

BNF

```
Exp := Id
  | Id "(" Exps ")"
  | Exp "+" Exp
  | ...

Exps :=
  | Expp

Expp := Exp
  | Expp "," Exp
```
Rephrasing — partial evaluation

```plaintext
function power(x : int, n : int) : int =
  if n = 0 then 1
  else if even(n) then square(power(x, n/2))
  else (x * power(x, n - 1))
```

\[ \downarrow \quad n = 5 \]

```plaintext
function power5(x : int) : int =
  x * square(square(square(x)))
```

Transformation pipeline

http://losser.st-lab.cs.uu.nl/~mbravenb/PT05-Infrastructure.pdf
Roadmap

> Program Transformation
  — Introduction
  — Stratego/XT
  — TXL

> Refactoring

> Aspect-Oriented Programming
Stratego/XT

> **Stratego**
> — A language for specifying program transformations
>   - term rewriting rules
>   - programmable rewriting strategies
>   - pattern-matching against syntax of object language
>   - context-sensitive transformations

> **XT**
> — A collection of transformation tools
>   - parser and pretty printer generators
>   - grammar engineering tools

Stratego/XT

Syntax definition

Parser generator
  Parse table
  Parse
    Program

Tree grammar generator
  Tree grammar
  Transform
    Tree
    Pretty-print
      Program

Pretty-print generator
  Pretty-print table

http://losser.st-lab.cs.uu.nl/~mbravenb/PT05-Infrastructure.pdf
Parsing

Rules translate terms to terms

Stratego parses any context-free language using Scannerless Generalized LR Parsing

module Exp
exports
  context-free start-symbols Exp
  sorts Id IntConst Exp

lexical syntax
  [\t\n] \rightarrow LAYOUT
  [a-zA-Z]+ \rightarrow Id
  [0-9]+ \rightarrow IntConst

context-free syntax
  Id \rightarrow Exp \{cons("Var")\}
  IntConst \rightarrow Exp \{cons("Int")\}

  "( " Exp " )" \rightarrow Exp \{bracket\}

  Exp "*" Exp \rightarrow Exp \{left, cons("Mul")\}
  Exp "/" Exp \rightarrow Exp \{left, cons("Div")\}
  Exp "%" Exp \rightarrow Exp \{left, cons("Mod")\}

  Exp "+" Exp \rightarrow Exp \{left, cons("Plus")\}
  Exp "-" Exp \rightarrow Exp \{left, cons("Minus")\}

context-free priorities
  \{left:
    Exp "*" Exp \rightarrow Exp
    Exp "/" Exp \rightarrow Exp
    Exp "%" Exp \rightarrow Exp
  \}

> \{left:
    Exp "+" Exp \rightarrow Exp
    Exp "-" Exp \rightarrow Exp
  \}
testsuite Exp
topsort Exp

test egl parse
   "1 + 2 * (3 + 4) * 3 - 1"
->
   Minus(
      Plus(
         Int("1")
         , Mul(
            Mul(Int("2"), Plus(Int("3"), Int("4")))
         , Int("3")
      )
      , Int("1")
   )
Running tests

Pack the definitions

pack-sdf -i Exp.sdf -o Exp.def
  including ./Exp.sdf

Generate the parse table

sdf2table -i Exp.def -o Exp.tbl -m Exp
SdfChecker: error: Main module not defined
--- Main

Run the tests

parse-unit -i Exp.testsuite -p Exp.tbl
executing testsuite Exp with 1 tests

* OK : test 1 (egl parse)

results testsuite Exp
successes : 1
failures  : 0
Interpretation example

```plaintext
module ExpEval

imports libstratego-lib
imports Exp

rules
  convert : Int(x) -> <string-to-int>(x)
  eval : Plus(m,n) -> <add>(m,n)
  eval : Minus(m,n) -> <subt>(m,n)
  eval : Mul(m,n) -> <mul>(m,n)
  eval : Div(m,n) -> <div>(m,n)
  eval : Mod(m,n) -> <mod>(m,n)

strategies
  main = io-wrap(innermost(convert <+ eval))
```

Stratego separates the specification of **rules** (transformations) from **strategies** (traversals). In principle, both are reusable.

File: ExpEval.str

File: ultimate-question.txt

1 + 2 * (3 + 4) * 3 - 1

18
A strategy determines how a set of rewrite rules will be used to traverse and transform a term.

- innermost
- top down
- bottom up
- repeat
- …
Running the transformation

sdf2rtg -i Exp.def -o Exp.rtg -m Exp
SdfChecker: error: Main module not defined
Main

rtg2sig -i Exp.rtg -o Exp.str

strc -i ExpEval.str -la stratego-lib
[ strc | info ] Compiling 'ExpEval.str'
[ strc | info ] Front-end succeeded : [user/system] = [0.56s/0.05s]
[ strc | info ] Optimization succeeded -O 2 : [user/system] = [0.00s/0.00s]
[ strc | info ] Back-end succeeded : [user/system] = [0.16s/0.01s]

gcc -I /usr/local/strategoxt/include -I /usr/local/strategoxt/include -I /usr/local/strategoxt/include -Wall -Wno-unused-label -Wno-unused-variable -Wno-unused-function -Wno-unused-parameter -DSIZEOF_VOID_P=4 -DSIZEOF_LONG=4 -DSIZEOF_INT=4 -c ExpEval.c -fno-common -DPIC -o .libs/ExpEval.o
gcc -I /usr/local/strategoxt/include -I /usr/local/strategoxt/include -I /usr/local/strategoxt/include -Wall -Wno-unused-label -Wno-unused-variable -Wno-unused-function -Wno-unused-parameter -DSIZEOF_VOID_P=4 -DSIZEOF_LONG=4 -DSIZEOF_INT=4 -c ExpEval.c -o ExpEval.o >/dev/null 2>&1
[ strc | info ] C compilation succeeded : [user/system] = [0.31s/0.36s]
[ strc | info ] Compilation succeeded : [user/system] = [1.03s/0.42s]

gslri -p Exp.tbl -i ultimate-question.txt | ./ExpEval
42
Roadmap

> Program Transformation
  — Introduction
  — Stratego/XT
  — TXL

> Refactoring

> Aspect-Oriented Programming
The TXL paradigm: *parse, transform, unparses*

http://www.txl.ca/docs/TXLintro.pdf
TXL programs

- **Base grammar**: defines tokens and non-terminals
- **Grammar overrides**: extend and modify types from grammar
- **Transformation rules**: rooted set of rules and functions
% Part I. Syntax specification

define program
    [expression]
end define

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

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

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

% Part 2. Transformation rules

rule main
    replace [expression]
        E [expression]
    construct NewE [expression]
        E [resolveAddition]
        [resolveSubtraction]
        [resolveMultiplication]
        [resolveDivision]
        [resolveBracketedExpressions]
    where not
        NewE [= E]
    by
        NewE
end rule

rule resolveAddition
    replace [expression]
        N1 [number] + N2 [number]
    by
        N1 [+ N2]
end rule

rule resolveBracketedExpressions
    replace [primary]
        ( N [number] )
    by
        N
end rule
Running the example

File: Ultimate.Question

1 + 2 * (3 + 4) * 3 - 1

```plaintext
taxl Ultimate.Question
TXL v10.5d (1.7.08) (c)1988-2008 Queen's University at Kingston
Compiling Question.Txl ...
Parsing Ultimate.Question ...
Transforming ...
42
```
// Find all factors of a given input 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: factors.til

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

```plaintext
define program
  [statement*]
end define

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

define declaration
  'var [id] ;                   [NL]
end define

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

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

All TXL parsers are also pretty-printers if the grammar includes formatting cues
Pretty-printing TIL

```til
include "TIL.Grm"
function main
  match [program]
    _ [program]
  end function

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: TILparser.Txl
Generating statistics

include "TIL.Grm"

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: %"]

        ...
    by
        % nothing

end function
include "TIL.Grm"

redefine statement
  ...
  | [traced_statement]
end redefine

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

rule main
replace [repeat statement]
  S [statement]
  Rest [repeat statement]
  ...
  by
  'write QuotedS;    'TRACED
  S                      'TRACED
  Rest
end rule

write "Trace: var n;";
var n;
write "Trace: write "Input n please";";
write "Input n please";
write "Trace: read n;";
read n;
...
## TXL vs Stratego

<table>
<thead>
<tr>
<th></th>
<th>Stratego</th>
<th>TXL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scannerless GLR parsing</td>
<td>Agile parsing (top-down + bottom-up)</td>
<td></td>
</tr>
<tr>
<td>Reusable, generic traversal strategies</td>
<td>Fixed traversals</td>
<td></td>
</tr>
<tr>
<td>Separates rewrite rules from traversal strategies</td>
<td>Traversals part of rewrite rules</td>
<td></td>
</tr>
</tbody>
</table>
Commercial systems

“The DMS Software Reengineering Toolkit is a set of tools for automating customized source program analysis, modification or translation or generation of software systems, containing arbitrary mixtures of languages.”

http://www.semdesigns.com/Products/DMS/DMSToolkit.html
Roadmap

> Program Transformation

> Refactoring
  — Refactoring Engine and Code Critics
  — Eclipse refactoring plugins

> Aspect-Oriented Programming
What is Refactoring?

> The process of *changing a software system* in such a way that it *does not alter the external behaviour* of the code, yet *improves its internal structure*.

Rename Method — manual steps

> Do it yourself approach:
  — Check that no method with the new name already exists in any subclass or superclass.
  — Browse all the implementers (method definitions)
  — Browse all the senders (method invocations)
  — Edit and rename all implementers
  — Edit and rename all senders
  — Remove all implementers
  — Test

> Automated refactoring is better!
Rename Method

> Rename Method (method, new name)

> Preconditions
  — No method with the new name already exists in any subclass or superclass.
  — No methods with same signature as method outside the inheritance hierarchy of method

> PostConditions
  — method has new name
  — relevant methods in the inheritance hierarchy have new name
  — invocations of changed method are updated to new name

> Other Considerations
  — Typed/Dynamically Typed Languages => Scope of the renaming
The Refactoring Browser
## Typical Refactorings

<table>
<thead>
<tr>
<th>Class Refactorings</th>
<th>Method Refactorings</th>
<th>Attribute Refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (sub)class to hierarchy</td>
<td>add method to class</td>
<td>add variable to class</td>
</tr>
<tr>
<td>rename class</td>
<td>rename method</td>
<td>rename variable</td>
</tr>
<tr>
<td>remove class</td>
<td>remove method</td>
<td>remove variable</td>
</tr>
<tr>
<td>push method down</td>
<td>push method down</td>
<td>push variable down</td>
</tr>
<tr>
<td>push method up</td>
<td>pull variable up</td>
<td></td>
</tr>
<tr>
<td>add parameter to method</td>
<td>create accessors</td>
<td>abstract variable</td>
</tr>
<tr>
<td>move method to component</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extract code in new method</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Don Roberts, “Practical Analysis for Refactoring,”

Bill Opdyke, “Refactoring Object-Oriented Frameworks,”
Code Critic — search for common errors
Refactoring Engine — matching trees

NB: All metavariables start with `\`

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\</code></td>
<td>recurse</td>
</tr>
<tr>
<td><code>@</code></td>
<td>list</td>
</tr>
<tr>
<td><code>.</code></td>
<td>statement</td>
</tr>
<tr>
<td><code>#</code></td>
<td>literal</td>
</tr>
</tbody>
</table>

```
`\@object halt` recursively match send of halt
`\@.Statements` match list of statements
Class `\@message: `\@args` match all sends to Class
```
Rewrite rules

```ruby
initialize

super initialize.

self rewriteRule
  replace: `'@object = nil'` with: `'@object isNil';
  replace: `'@object == nil'` with: `'@object isNil';
  replace: `'@object =~ nil'` with: `'@object notNil';
  replace: `'@object ~~ nil'` with: `'@object notNil'
```
Roadmap

> Program Transformation
> Refactoring
  — Refactoring Engine and Code Critics
  — Eclipse refactoring plugins
> Aspect-Oriented Programming
A workbench action delegate

When the workbench action proxy is triggered by the user, it delegates to an instance of this class.

```java
package astexampleplugin.actions;
...
import org.eclipse.ui.IWorkbenchWindowActionDelegate;

public class ChangeAction implements IWorkbenchWindowActionDelegate {
    ...
    public void run( IAction action ) {
        for ( ICompilationUnit cu : this.classes ) {
            try {
                ...
                parser.setSource( cu );
                ...
                CompilationUnit ast = (CompilationUnit)parser.createAST( null );
                ...
                StackVisitor visitor = new StackVisitor( ast.getAST() );
                ast.accept( visitor );
                ...
            } catch ...
        }
    }
    ...
}
```

A field renaming visitor

```java
package astexampleplugin.ast;
...
import org.eclipse.jdt.core.dom.ASTVisitor;

public class StackVisitor extends ASTVisitor {
    private static final String PREFIX = "_";
    ...
    public boolean visit(FieldDeclaration field){
        ...
    }

    public boolean visit(FieldAccess fieldAccess){
        String oldName = fieldAccess.getName().toString();
        String newName = this.fields.get( oldName );
        if(newName == null){
            newName = PREFIX + oldName;
            this.fields.put( oldName , newName );
        }
        fieldAccess.setName( this.ast.newSimpleName( newName ) );
        return true;
    }
}
```

The visitor simply implements the visit method for field declarations and accesses, and prepends an underscore.
Renaming fields

```java
public class Machine {
    private Hashtable<String, Integer> _store; // current this.values of variables
    private StringBuffer _output; // print stream so far
    private int _value;
    private Vector<Integer> _vlist; // result of current expression
    private Vector<Integer> _vlist; // list of expressions computed

    public Machine() {
        this._store = new Hashtable<String, Integer>();
        this._output = new StringBuffer();
        this.setValue(0);
        this._vlist = new Vector<Integer>();
    }
}
```
Roadmap

- Program Transformation
- Refactoring
- Aspect-Oriented Programming
Problem: cross-cutting concerns

Certain features (like logging, persistence and security), cannot usually be encapsulated as classes. They *cross-cut* code of the system.

“Identifying Cross-Cutting Concerns in Embedded C Code”, Bruntink, van Deursen, Tourwé
Aspect-Oriented Programming

AOP improves modularity by supporting the separation of cross-cutting concerns.

An **aspect** packages cross-cutting concerns

A **pointcut** specifies a set of **join points** in the target system to be affected

**Weaving** is the process of applying the aspect to the target system
package tjp;

public class Demo {
  static Demo d;
  public static void main(String[] args) {
    new Demo().go();
  }
  void go() {
    d = new Demo();
    d.foo(1, d);
    System.out.println(d.bar(new Integer(3)));
  }
  void foo(int i, Object o) {
    System.out.println("Demo.foo(\" + i + ", \" + o + \")\n");
  }
  String bar (Integer j) {
    System.out.println("Demo.bar(\" + j + \")\n");
    return "Demo.bar(\" + j + \")";
  }
}


Demo.foo(1, tjp.Demo@939b78e)
Demo.bar(3)
Demo.bar(3)
A logging aspect

aspect GetInfo {
    pointcut goCut(): cflow(this(Demo) && execution(void go()));
    pointcut demoExecs(): within(Demo) && execution(* *(..));

    Object around(): demoExecs() && !execution(* go()) && goCut() {
        ...
    }
    ...
}

Intercept execution within control flow of Demo.go()
Identify all methods within Demo
Wrap all methods except Demo.go()
aspect GetInfo {
    ...
    Object around(): demoExecs() && !execution(* go()) && goCut() {
        println("Intercepted message: " +
            thisJoinPointStaticPart.getSignature().getName());
        println("in class: " +
            thisJoinPointStaticPart.getSignature().getDeclaringType().getName());
        printParameters(thisJoinPoint);
        println("Running original method: 
            Object result = proceed();
        println("  result: " + result );
        return result;
    }
    ...
}
Making classes visitable with aspects

```java
public class SumVisitor implements Visitor {
    int sum = 0;
    public void visit(Nil l) {}

    public void visit(Cons l) {
        sum = sum + l.head;
        l.tail.accept(this);
    }

    public static void main(String[] args) {
        List l = new Cons(5, new Cons(4, new Cons(3, new Nil())));
        SumVisitor sv = new SumVisitor();
        l.accept(sv);
        System.out.println("Sum = " + sv.sum);
    }
}

public interface Visitor {
    void visit(Nil l);
    void visit(Cons l);
}

public interface List {}
public class Nil implements List {}
public class Cons implements List {
    int head;
    List tail;
    Cons(int head, List tail) {
        this.head = head;
        this.tail = tail;
    }
}
```

We want to write this

```java
public interface List {}
public class Nil implements List {}
public class Cons implements List {
    int head;
    List tail;
    Cons(int head, List tail) {
        this.head = head;
        this.tail = tail;
    }
}
```

But we are stuck with this …
package ajvisit;

public aspect Visitable {
    public void List.accept(Visitor v) { }

    public void Nil.accept(Visitor v) {
        v.visit(this);
    }

    public void Cons.accept(Visitor v) {
        v.visit(this);
    }
}
With aspects, who needs visitors?

```java
public class SumList {
    public static void main(String[] args) {
        List l = new Cons(5, new Cons(4, new Cons(3, new Nil())));
        System.out.println("Sum = " + l.sum());
    }
}

This would be even cleaner

```java
public aspect Summable {
    public int List.sum() {
        return 0;
    }
    public int Nil.sum() {
        return 0;
    }
    public int Cons.sum() {
        return head + tail.sum();
    }
}
```

The missing method is just an aspect
What you should know!

- What are typical program transformations?
- What is the typical architecture of a PT system?
- What is the role of term rewriting in PT systems?
- How does TXL differ from Stratego/XT?
- How does the Refactoring Engine use metavariables to encode rewrite rules?
- Why can’t aspects be encapsulated as classes?
- What is the difference between a pointcut and a join point?
Can you answer these questions?

- How does program transformation differ from metaprogramming?
- In what way is optimization a form of PT?
- What special care should be taken when pretty-printing a transformed program?
- How would you encode typical refactorings like “push method up” using a PT system like TXL?
- How could you use a PT system to implement AOP?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

[http://creativecommons.org/licenses/by-sa/4.0/](http://creativecommons.org/licenses/by-sa/4.0/)