11. Program Transformation

Oscar Nierstrasz
Roadmap

- Program Transformation
- Refactoring
- Aspect-Oriented Programming
<table>
<thead>
<tr>
<th>Links</th>
<th></th>
</tr>
</thead>
</table>
| **Program Transformation:** | — http://swerl.tudelft.nl/bin/view/Pt  
|                  | — http://www.program-transformation.org/ |
| **Stratego:**  | — http://strategoxt.org/ |
| **TXL:**       | — http://www.txl.ca/ |
|                  | — 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

Refinement — transform high-level spec down to an implementation that fulfils requirements
Renovation — reengineering
Translation — compilation

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

MIPS

fact: subu $sp, $sp, 20
sw $fp, 0($sp)
addiu $fp, $sp, 20
sw $s2, -0($fp)
sw $ra, -4($fp)
sw $s0, 0($fp)
mv $s2, $a1
li $t0, 1
bge $s2, $t0, c_0
li $v0, 1
b d_0
c_0: lw $s0, 0($fp)
li $t0, 1
subu $s1, $s2, $t0
jal fact, a_0
slw $v0, $s2, $v0
d_0: lw $s2, -8($fp)
lw $ra, -4($fp)
lw $fp, 8($sp)
addiu $sp, $sp, 20
jr $ra
```

Translation — migration from procedural to OO

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

```tiger
type tree = {key: int, children: treelis}
type treelis = {hd: tree, tl: treelis}
function treesize(t : tree) : int =
    if t = nil then 0 else 1 + listsize(t.children)
function listsize(ts : treelis) =
    if ts = nil then 0 else listsize(t.tl)
```
Rephrasing — desugaring regular expressions

\[
\text{Exp} := \text{Id} \\
| \text{Id} \ "(" \text{Exps} \ "\)"} \\
| \text{Exp} \ "+" \text{Exp} \\
| \ldots
\]

\[
\text{Exps} := \\
| \text{Expp}
\]

\[
\text{Expp} := \text{Exp} \\
| \text{Expp} \ "\," \text{Exp}
\]
Rephrasing — partial evaluation

```c
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))
```

Tiger

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

This general scheme applies to Stratego, TXL and various other systems. Transformation systems and languages may support or automate different parts of this pipeline.

If the source language is fixed, then a fixed parser and pretty-printer may be used.

If the source and target languages are arbitrary, then there should be support to specify grammars and automatically generate parsers and pretty-printers.
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

http://strategoxt.org/
Parser and basic pretty-printer 100% generated.
Language specific support for transformations generated.
Parsing

Rules translate terms to terms

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

See the Makefile for the steps needed to run this.

GLR parsing essentially does a parallel, breadth-first LR parse to handle ambiguity.

http://en.wikipedia.org/wiki/GLR_parser
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-sdf -i Exp.sdf -o Exp.def
  including ./Exp.sdf

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

parse-unit -i Exp.testsuite -p Exp.tbl
executing testsuite Exp with 1 tests
* OK : test 1 (eg1 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: ultimate-question.txt

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

18
Strategies

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

```
running the transformation

ddf2rtg -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
    libstratego-lib.dylib /usr/local/strategoxt/lib/libstratego-lib-native.dylib /usr/local/strategoxt/lib/
    libstratego-runtime.dylib -lm /usr/local/strategoxt/lib/libATerm.dylib
    [ strc | info ] C compilation succeeded : [user/system] = [0.31s/0.36s]
    [ strc | info ] Compilation succeeded : [user/system] = [1.03s/0.42s]
sgli -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, unparse

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*
NB: TXL reverses the usual BNF convention and puts non-terminals in square brackets while interpreting everything else (except special chars) as terminals. The default lexical scanner can be modified, but is usually fine for first experiments.
Running the example

File: Ultimate.Question

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

```
txl 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

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

% Keywords of TIL
keys
  var if then else while
do for read write
end keys

% Compound tokens
compounds
  := !=
end compounds

% Commenting convention
comments
  //
end comments

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

File: TIL.Grm

27
Pretty-printing 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
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"

... redefined 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
...
## TXL vs Stratego

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

See also http://www.semdesigns.com/Products/DMS/DMSComparison.html for a comparison to other approaches
Roadmap

> Program Transformation
> **Refactoring**
>   — Refactoring Engine and Code Critics
>   — Eclipse refactoring plugins
> > Aspect-Oriented Programming
> 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 (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></td>
<td>push method down</td>
<td>push variable down</td>
</tr>
<tr>
<td></td>
<td>push method up</td>
<td>pull variable up</td>
</tr>
<tr>
<td>add parameter to method</td>
<td>create accessors</td>
<td></td>
</tr>
<tr>
<td>move method to component</td>
<td>abstract variable</td>
<td></td>
</tr>
<tr>
<td>extract code in new method</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Code Critic — search for common errors
Refactoring Engine — matching trees

**Syntax**

| `~`  | recurse           |
| `@`  | list              |
| `.`  | statement         |
| `#`  | literal           |

NB: All metavariables start with `~`

| `~`@object halt          | recursively match send of halt |
| `~`@.Statements          | match list of statements      |
| Class `~`@message: `~@args` | match all sends to Class      |

The first `~` is for all meta-variables.
Rewrite rules
Roadmap

- Program Transformation
- Refactoring
  - Refactoring Engine and Code Critics
  - Eclipse refactoring plugins
- Aspect-Oriented Programming
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 ...
      }
    }
  }
  ...
}

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

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;
    }
}
Renaming fields
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, Tourné
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 + ")";
    }
}

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;
  }

...}

Intercepted message: foo
in class: tjp.Demo
Arguments:
  0. i : int = 1
  1. o : java.lang.Object = tjp.Demo@c0b76fa
Running original method:
Demo.foo(1, tjp.Demo@c0b76fa)
  result: null
Intercepted message: bar
in class: tjp.Demo
Arguments:
  0. j : java.lang.Integer = 3
Running original method:
Demo.bar(3)
  result: Demo.bar(3)
Demo.bar(3)
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 class Cons implements List {
    int head;
    List tail;
    Cons(int head, List tail) {
        this.head = head;
        this.tail = tail;
    }
}

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 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);
}
```

But we are stuck with this ...

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

Dunno why List.sum() needs a body – it should just be an interface signature.
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?
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/