

A Taxonomy of Program Transformation

Eelco Visser

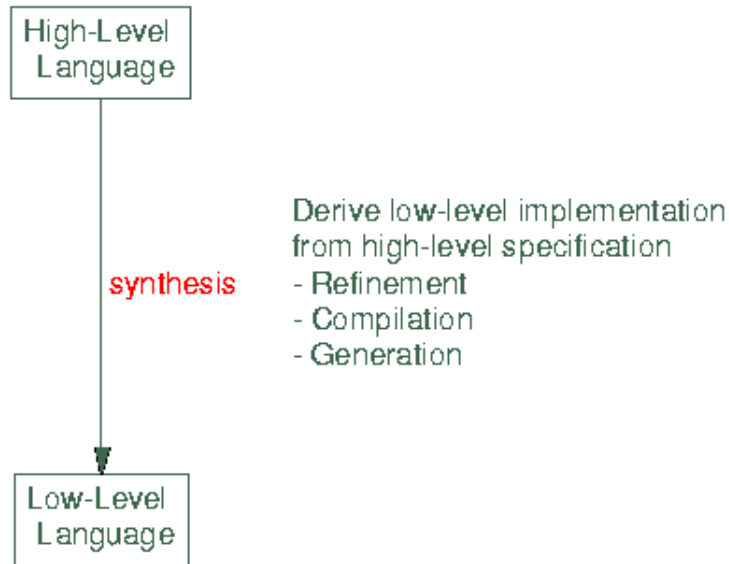
Center for Software Technology, Utrecht University, Utrecht, The Netherlands
visser@cs.uu.nl, <http://www.cs.uu.nl/~visser>

Program Transformation 2003

Program Transformation

- *A program transformation is a modification or sequence of modifications to a program* (or an algorithm for making such modifications)
- Taxonomy: classification of program transformations
 - What is the purpose of a transformation?
 - How does it affect a program?
 - What is the relation with other transformations?
- Implementation of program transformation
 - Program transformation by term rewriting
 - and beyond

Program Synthesis



Example: 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
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
```

MIPS

Example: Parser Generation

```

expr   : lam;
lam    : apps | '\\\ ' var '.' lam ;
apps   : simple | apps simple ;
simple  : var | num | '(' expr ')' ;
var    : ID;
num    : DIGITS;

```

YACC



C

```

static const short yygoto[] = {-32768, 9,-2,-32768,7,12,-32768};
static const short yytable[] = { 1, 2, 3, 1, 4, 1, 2, 14, 18, 4, 1 ...};
int yyparse(YYPARSE_PARAM_ARG) { ...
yyreduce: yylen = yyr2[yyin];
  if (yylen > 0) yyval = yyvsp[1-yylen]; ...
  yystate = yygoto[yyin - YYNTBASE] + *yyssp;
  if (yystate >= 0 && yystate <= YYLAST && yycheck[yystate] == *yyssp)
    yystate = yytable[yystate];
  else
    yystate = yydefgoto[yyin - YYNTBASE];
  goto yystate; ...
}

```

Application Generation

```
Object(
  Name("Reservation"),
  Properties([ Property(Name("Remark"),          Type(Prim(String()))
                , Property(Name("StartDateTime"), Type(CustomPrim("DateTime"))), ...]),
  Relations( [ Reference(Name("Product"),        Type(Object("Product"))
                , Reference(Name("Customer"),    Type(Object("Customer"))), ...]))
```

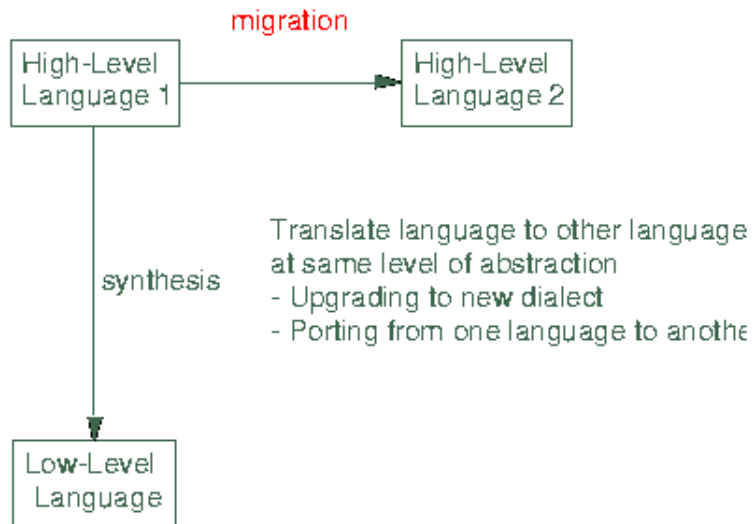
DSL



Java

```
public class ReservationRelation extends AbstractDatabase<Reservation, StoredReser...
{
  public List<StoredReservation> read(StoredProduct product, ...) throws ReadException {
    return readList("SELECT " + ALL + " FROM Reservation WHERE productID = " + prod...
  public List<StoredReservation> readWithStartTime (StoredProduct product , Calenda...
    return readList("SELECT " + ALL + " FROM Reservation WHERE productID = " + pro...
  public ReservationRelation (final JReserveDatabaseDomainObjectLoader loader , fin...
    super(database , new StoredReservationFactory( ) , "Reservation");
    _loader = loader;
    _dispatcher = new StoreDispatcher( ); }
  public void store (Reservation reservation) throws StoreException { ... }
  protected void write (final PreparedStatement stm , final StoredReservation reser...
  protected StoredReservation read(final ResultSet resultSet) throws SQLException {...
}
```

Migration



Example: Migration from Procedural to OO

```
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)
```

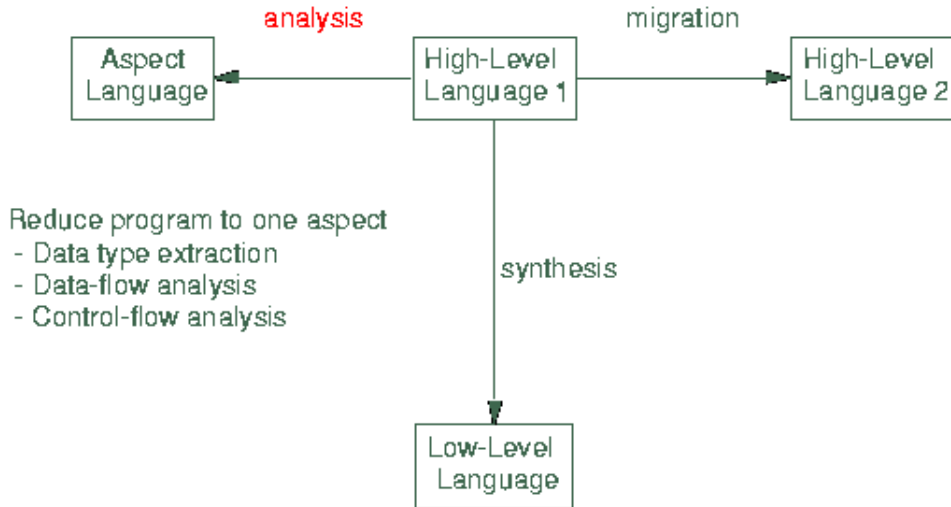
Tiger



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

Java

Program Analysis



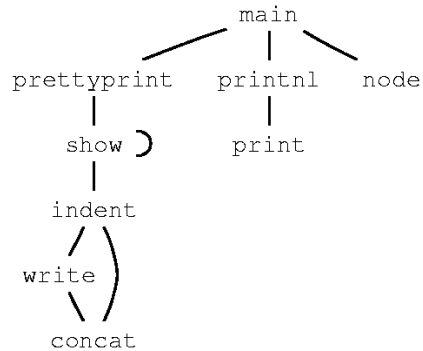
Example: Call Graph

```

type tree = ...
function prettyprint(...) : string =
let function write(s: string) =
  output := concat(...)
  function show(...) =
  let function indent(...) =
    (write(..);
     for i := 1 to n do write(" ");
     output := concat(output, s))
  in if t = nil then indent(".")
     else (...; show(...); ...)
  end
  in show(0, tree); output end
function node(...) : tree =
  tree{key = x, left = l, right = r}
function println(...) =
  (print(x); print("\n"))
in let var t := node(...)
   in println(prettyprint(t)) end
end

```

⇒



Tiger

Package ssl / Module simple-traversal (code view) - Mozilla

File Edit View Go Bookmarks Tools Window Help

http://catamaran.labs.cs.uu.nl/docs/ssl/current/simple-traversal/simple-traversal.html

Home Bookmarks Red Hat, I... Build Log ... Red Hat N... Support Shop Products Training VerzekeV...

Overview | Browse code | All definitions | Module index | Constructor index | Overlay index | Strategy index | Rule index | Dynamic Rule index

Packages

ssl

Modules

ssl
 annotation-props
 annotations
 apply
 bin-tree
 bin-tree-set
 char
 char-io
 collect
 conditional
 config
 cpl
 date-format
 dir

Definitions

Strategy
 alldownup2 [pp]
 alld [pp]
 alld-fold [pp]
 bottomup [pp]
 bottomup-para [pp]
 bottomupS [pp]
 breadthfirst [pp]
 don't-stop [pp]
 downup [pp]
 downup [pp]
 downup2 [pp]

```

\literate[simple-traversal]

  \begin{abstract}

    The primitive term traversal operators of Stratego (all, some, one)
    can be combined with the other control operators in a wide
    variety of ways to define full term traversals.
    This module defines
    a collection of the most common generic one-pass traversals
    over terms.

  \end{abstract}

\begin{code}
\module simple-traversal
\imports conditional term-properties
\strategies
\end{code}

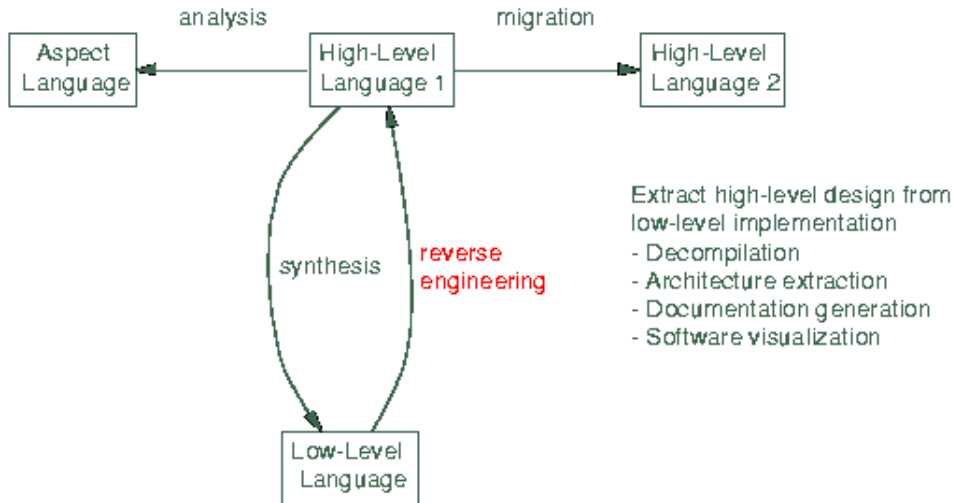
Term traversals can be categorized into classes according to
how much of the term they traverse and to which parts
of the term they modify.

\paragraph{Everywhere}

The most general class of traversals visits every node
of a term and applies a transformation to it. The following

```

Reverse Engineering



Example: Goto Elimination

```
f <- 1
a_0 : if x > n goto b_3
      x <- x + 1
      f <- f * x
      goto a_0
b_3 : print(f)
```

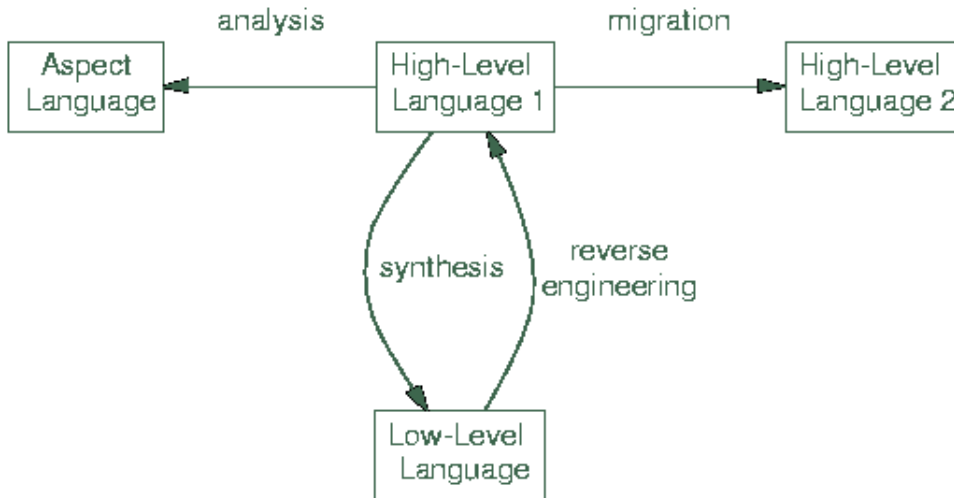
Tiger

⇒

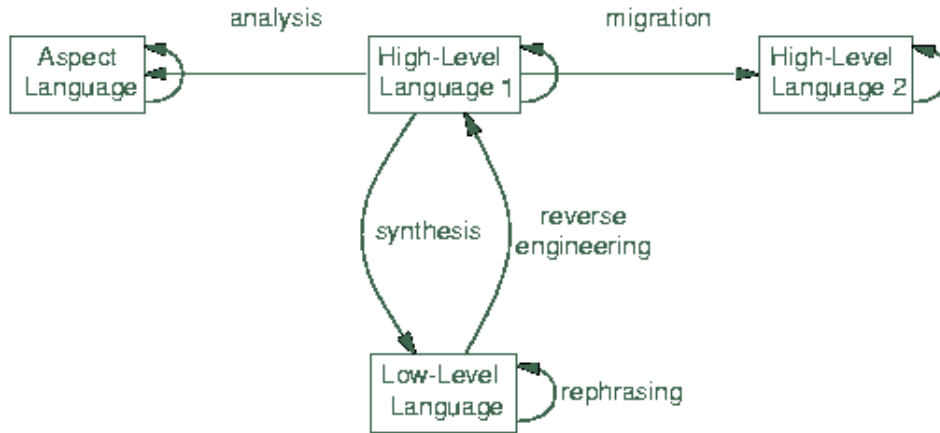
```
f := 1;
while x <= n do (
  x := x + 1;
  f := f * x
);
print(f)
```

Tiger

Program Transformation: Translation



Program Transformation: Rephrasing



Rephrasing: Normalization

- A normalization reduces a program to a program in a sub-language to decrease its syntactic complexity
- Simplification
 - Algebraic simplifications in compiler
- Desugaring
 - Haskell to Core Haskell
 - EBNF to BNF
- Aspect weaving
 - Tracing
 - Synchronisation

Example: Desugaring Regular Expressions

```

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

```

EBNF

⇒

```

Exp := Id
     | Id "(" Exps ")"
     | Exp "+" Exp
     | ...
Exps :=
     | Expp
Expp := Exp
      | Expp "," Exp

```

BNF

Example: Desugaring List Comprehensions

```
[ square x | x <- xs; odd x ]
```

Haskell



```
let h = \ us -> case us of
    [] -> []
    (x : us') -> if odd x
                    then (square x) : (h us')
                    else (h us')
in h xs
```

Haskell

Rephrasing: Optimization

- An optimization improves the run-time and/or space performance of a program
- Examples
 - Common subexpression elimination
 - Constant folding and propagation
 - Dead code elimination
 - Fusion: Loop fusion, Deforestation
 - Inlining, Specialization
 - Instruction scheduling
 - Strength reduction
 - Tail-recursion elimination
 - ...

Example: Constant Folding and Propagation

```
var N := 8
var solutions := 0
type intArray = array of int
var row := intArray [ N ] of 0
var col := intArray [ N ] of 0
var diag1 := intArray [N+N-1] of 0
var diag2 := intArray [N+N-1] of 0
```

Tiger



```
var N := 8
var solutions := 0
type intArray = array of int
var row := intArray[8] of 0
var col := intArray[8] of 0
var diag1 := intArray[15] of 0
var diag2 := intArray[15] of 0
```

Tiger

Example: Tail Recursion Elimination

```
function fact(n : int) : int =  
  let function f(n : int, acc : int) : int =  
    if n < 1 then acc else f(n - 1, n * acc)  
  in f(n, 1)  
end
```

Tiger



```
function fact(n : int) : int =  
  let function f(n : int, acc : int) : int =  
    (while n >= 1 do  
      (acc := n * acc;  
       n := n - 1);  
     acc)  
  in f(n, 1)  
end
```

Tiger

Example: Inlining

```
function fact(n : int) : int =  
  let function f(n : int, acc : int) : int =  
    (while n >= 1 do  
      (acc := n * acc;  
       n := n - 1);  
      acc)  
  in f(n, 1)  
end
```

Tiger

```
function fact(n : int) : int =  
  let var acc : int := 1  
  in while n >= 1 do  
    (acc := n * acc;  
     n := n - 1);  
    acc  
  end
```

Tiger

Example: Partial Evaluation

```
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

⇓ n = 5

Tiger

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

Example: Strength Reduction

```
x := 0;
while x < n do (
  x := x + 1;
  s := i * x;
  f(s)
)
```

Tiger

\Rightarrow

```
x := 0;
s := 0;
while x < n do (
  x := x + 1;
  s := s + i;
  f(s)
)
```

Tiger

Example: Vectorization

```
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

FORTRAN90



```
DO I = 1, N
  DO J = 1, N, 64
    C(J:J+63,I) = 0.0
    DO K = 1, N
      C(J:J+63,I) = C(J:J+63,I) + A(J:J+63,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

FORTRAN90

Rephrasing: Refactoring

- A refactoring changes the structure of the program to make it easier (or harder) to understand
- Preserves observable behavior
- Design improvement
 - Extract method
 - Move method
 - Inline method
- Obfuscation
 - Hide business rules

Example: Extract Method

```
void printOwing() {  
    printBanner();  
    //print details  
    System.out.println ("name: " + _name);  
    System.out.println ("amount " + getOutstanding());  
}
```

Java*Java*

```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
void printDetails (double outstanding) {  
    System.out.println ("name: " + _name);  
    System.out.println ("amount " + outstanding);  
}
```

Rephrasing: Renovation

- Software renovation is used to
 - Repair an error
 - Bring a program up to date with respect to changed requirements
- Does not preserve semantics
- Error Repair
 - Year 2000
- Changed requirements
 - Euro
 - Changed product

Example: Picture Widening

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. EXPANDPICTURE.  
DATA DIVISION.
```

```
WORKING-STORAGE SECTION.  
01  PRODKODE PIC 99.  
01  X PIC 9(2).  
01  Y PIC 9(2).
```

```
PROCEDURE DIVISION.  
FOO SECTION.  
PAR1.  
    MOVE PRODKODE TO X.
```

COBOL

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. EXPANDCONSTANT.  
DATA DIVISION.
```

```
WORKING-STORAGE SECTION.  
01  PRODKODE PIC 999.  
01  X PIC 9(3).  
01  Y PIC 9(2).
```

```
PROCEDURE DIVISION.  
FOO SECTION.  
PAR1.  
    MOVE PRODKODE TO X.
```

COBOL

Program Transformation Scenarios

- **Translation:** Translate program in source language A to program in target language B
 - Migration
 - Synthesis
 - Reverse engineering
 - Analysis
- **Rephrasing:** Transform program in source language A to program in the same language
 - Normalization
 - Optimization
 - Refactoring
 - Renovation

Summary

A program transformation is a modification or sequence of modifications affecting the

- level of abstraction,
- implementation language,
- performance,
- understandability,
- correctness, or
- scope

of a program

Summary

with the purpose to

- understand a program
- support a different platform
- produce efficient implementation
- improve/hide the design of a program
- update a program to new requirements
- derive a high-level specification from an implementation
- generate an implementation from a high-level specification

Synonyms for 'Program Transformation'

- Meta-programming
 - synonym for program transformation (programming about programming)
- Generative programming
 - programming methodology based on program generation
- Program synthesis
 - (automatically) 'synthesizing' a program
- Program refinement
 - refining a specification to an implementation
- Program calculation
 - (manually) applying semantic laws to 'compute' a program

Assignment Classifying Transformations

- Online transformation demonstration
- Transformations labeled with letters
- Deduce which letter corresponds to which transformation
 - Bound Variable Renaming
 - Canonical (side-effect) free expressions
 - Compilation
 - Desugaring
 - Escaping Variables Analysis
 - Function Inlining
 - Interpreting
 - ...

Sources

Examples on these slides are derived from various sources, including:

- Andrew Appel. *Modern Compiler Implementation in ML*
- Neil Jones, Carsten K. Gomar and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*
- Martin Fowler. *Refactoring. Improving the Design of Existing Code*
- Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*