In Control of Rewriting

Eelco Visser

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

Program Transformation 2003
Outline

•
Term Rewriting for Program Transformation

Advantages

- First-order terms describe abstract syntax
- Rewrite rules express basic transformation rules (operationalizations of the algebraic laws of the language.)
- Rules specified separately from strategy

Limitations

- Rewrite systems for programming languages often non-terminating and/or non-confluent
- In general: do not apply all rules at the same time or apply all rules under all circumstances
A Non-terminating Rewrite System

signature
  sorts Prop
  constructors
    False : Prop
    True : Prop
    Atom : String -> Prop
    Not : Prop -> Prop
    And : Prop * Prop -> Prop
    Or : Prop * Prop -> Prop
rules
  DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
  DN : Not(Not(x)) -> x
  DMA : Not(And(x, y)) -> Or(Not(x), Not(y))
  DMO : Not(Or(x, y)) -> And(Not(x), Not(y))
Encoding Control with Functions

Common solution

- Introduce additional constructors that achieve normalization under a restricted set of rules
- Replace a ‘pure’ rewrite rule
  \[ p_1 \rightarrow p_2 \]
  with a functionalized rewrite rule:
  \[ f(p_1) \rightarrow p'_2 \]
  where \( f \) is a new constructor symbol (a function)
- Normalize terms \( f(t) \) with respect to these rules
- The function now controls where rules are applied
signature
  constructors
    dnf : Prop -> Prop
    and : Prop * Prop -> Prop
    not : Prop -> Prop
rules
  DNF1 : dnf(True) -> True
  DNF2 : dnf(False) -> False
  DNF3 : dnf(Atom(x)) -> Atom(x)
  DNF4 : dnf(Not(x)) -> not(dnf(x))
  DNF5 : dnf(And(x,y)) -> and(dnf(x),dnf(y))
  DNF6 : dnf(Or(x,y)) -> Or(dnf(x),dnf(y))

  AND1 : and(Or(x,y),z) -> Or(and(x,z),and(y,z))
  AND2 : and(z,Or(x,y)) -> Or(and(z,x),and(z,y))
  AND3 : and(x,y) -> And(x,y) (default)

  NOT1 : not(Not(x)) -> x
  NOT2 : not(And(x,y)) -> Or(not(x),not(y))
  NOT3 : not(Or(x,y)) -> and(not(x),not(y))
  NOT4 : not(x) -> Not(x) (default)
Analysis of Functionalization

Functional encoding has two main problems:

- **Overhead** due to explicit specification of *traversal*
  
  - A traversal rule needs to be defined for each constructor in the signature and for each transformation.

- **Separation of rules and strategy is lost**
  
  - Rules and strategy are completely *intertwined*
  
  - Intertwining makes it more difficult to *understand* the transformation
  
  - Intertwining makes it impossible to *reuse* the rules in a different transformation.
Language Complexity

Traversal overhead and reuse of rules is important, considering the complexity of real programming languages:

<table>
<thead>
<tr>
<th>language</th>
<th># constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiger</td>
<td>65</td>
</tr>
<tr>
<td>C</td>
<td>140</td>
</tr>
<tr>
<td>Java</td>
<td>140</td>
</tr>
<tr>
<td>COBOL</td>
<td>300–1200</td>
</tr>
</tbody>
</table>
Solutions

Requirements

- Control over application of rules
- No traversal overhead
- Separation of rules and strategies

Approaches

- Strategy annotations
- Sequences of normal forms
- Traversal functions
- Programmable rewriting strategies
Strategy Annotations

- Systems: Just-in-time, CafeOBJ
- Adapt the standard rewriting strategy with annotations
- Aim: solve the problem of non-termination in rewrite systems with infinite reduction paths that cannot be resolved by removing unnecessary rules
imports integers
signature
  sorts Int
  constructors
    Fac : Int -> Int
    If : Bool * Int * Int -> Int
rules
  Fac : Fac(x) -> If(Eq(x,0), 1, Mul(x,Fac(Sub(x,1))))
  IfT : If(True, x, y) -> x
  IfF : If(False, x, y) -> y
  IfE : If(p, x, x) -> x
annotations
  strat(If) = [1,IfT,IfF,2,3,IfE]

Strategy annotation defines the order in which constructor arguments are normalized and rules are applied
E-Strategy

signature
sorts Nat List(*)

constructors
  Z : Nat
  S : Nat -> Nat
  Cons : a * List(a) -> List(a) {strat: (1 0)}
  Inf : Nat -> List(Nat)
  Nth : List(a) -> a

rules
  Inf(x) -> Cons(x, Inf(S(x)))
  Nth(Z, Cons(x, l)) -> x
  Nth(S(x), Cons(y, l)) -> Nth(x, l)

The annotation declares the order of evaluation of the arguments.
Laziness Annotations

signature
  constructors
  Cons : a * List(a) -> List(a) \{Lazy(Cons,2)\}
  Inf : Nat -> List(Nat)
  Nth : List(a) -> a
rules
  Inf(x) -> Cons(x, Inf(S(x)))
  Nth(Z, Cons(x, l)) -> x
  Nth(S(x), Cons(y, l)) -> Nth(x, l)

implemented by transformation to terminating TRS:

rules
  Inf(x) -> Cons(x, Thunk(L, Vec1(x)))
  Nth(Z, Cons(x, l)) -> x
  Nth(S(x), Cons(y, l)) -> Nth(x, Inst(l))
  Inst(Thunk(L, Vec1(x))) -> Inf(S(X))
  Inst(x) -> x
Analysis of Strategy Annotations

- Help to make some rewrite systems terminating
- Do not help in other respects for program transformation
- No selection of a restricted set of rewrite rules.
- **Traversals** over abstract syntax trees still need to be defined explicitly.
Sequences of Normal Forms

- System: TAMPR
- Idea: control the selection of rules used in a normalization
- Sequences of normal forms
- Organize a large transformation into a sequence of consecutive reductions to canonical forms under different sets of rewrite rules
- Example: from polynomial in $y$ to polynomial in $x$ in four stages:

  sum-of-monomonials;
  x-commuted-to-right;
  like-powers-collected;
  x-factored-out
Sequences of Normal Forms: Example

\[(x^2 + 2x + 1)y^2 + (x^2 - 9)y - (20x^2 + 18x - 18)\]

**sum-of-monomonials:**

\[x^2y^2 + 2xy^2 + y^2 + x^2y - 9y - 20x^2 - 18x + 18\]

**x-commuted-to-right:**

\[y^2x^2 + 2y^2x + y^2 + yx^2 - 9y - 20x^2 - 18x + 18\]

**like-powers-collected:**

\[y^2x^2 + yx^2 - 20x^2 + 2y^2x - 18x + y^2 - 9y + 18\]

**factoring out the powers of** \(x\):

\[(y^2 + y - 20)x^2 + (2y^2 - 18)x + (y^2 - 9y + 18)\]
Analysis

• Close to the original term rewriting paradigm
• Traversal is implicit and handled by the rewriting strategy
• Problem of conflicting rules is solved by staging the application of such rules
• Does not allow more fine-grained control over the application of rules
Traversals Functions

- System: ASF+SDF
- Use functional encoding, but
- Reduce the overhead of traversal by **traversal functions**
- Traversal function declaration

  \[
  \text{dnf} : \text{Prop} \to \text{Prop} \ {\text{traversal(trafo,bottom-up)}}
  \]

  ‘generates’ traversal rules for this function

- Parameters
  - traversal strategy (bottom-up or top-down)
  - kind of transformation (transformation, accumulator)
# DNF with Traversal Function (Version 1)

**signature**
- **constructors**
  - `dnf : Prop -> Prop {traversal(trafo,bottom-up)}`
  - `and : Prop * Prop -> Prop`
  - `not : Prop -> Prop`

**rules**
- `DNF4 : dnf(Not(x)) -> not(x)`
- `DNF5 : dnf(And(x,y)) -> and(x,y)`

- `AND1 : and(Or(x,y),z) -> Or(and(x,z),and(y,z))`
- `AND2 : and(z,Or(x,y)) -> Or(and(z,x),and(z,y))`
- `AND3 : and(x,y) -> And(x,y) (default)`

- `NOT1 : not(Not(x)) -> x`
- `NOT2 : not(And(x,y)) -> Or(not(x),not(y))`
- `NOT3 : not(Or(x,y)) -> and(not(x),not(y))`
- `NOT4 : not(x) -> Not(x) (default)`
signature
constructors
dnf : Prop -> Prop \{traversal(trafo,bottom-up)\}

rules
AND1 : dnf(And(Or(x,y),z)) -> dnf(Or(And(x,z)),And(y,z))
AND2 : dnf(And(z,Or(x,y))) -> dnf(Or(And(z,x)),And(z,y))
NOT1 : dnf(Not(Not(x))) -> x
NOT2 : dnf(Not(And(x,y))) -> dnf(Or(Not(x),Not(y)))
NOT3 : dnf(Not(Or(x,y))) -> dnf(And(Not(x),Not(y)))
Analysis of Traversal Functions

Advantage

- Default traversal behaviour does not need to be implemented manually

Disadvantage

- No separation of rules from strategies
- No reuse of rules in different traversals
- Rules are intertwined with strategies
- Limited set of traversals
- Not first-class: not possible to make generic traversal schema (other than the ones provided)
Programmable Rewriting Strategies

• System: Stratego
• Rules can be combined using user-definable strategies into full transformations
• Define traversals with minimal overhead
• Maintain separation of rules and strategies
• Rules are first-class citizens
• Select rules to be applied in a specific transformation.
• Rules are not intertwined with strategies
• Rules can be reused in different transformations
• Many different idioms can be realized in one language
Cascading Transformations

• Apply small, independent transformations in combination
• Accumulative effect of small rewrites
• Example: compilation-by-transformation in the Glasgow Haskell Compiler
• Realization in Stratego

\[
\text{simplify} = \text{innermost}(R_1 + \ldots + R_n)
\]

• Example: disjunctive normal form

\[
\text{dnf} = \text{innermost}(DAOL + DAOR + DN + DMA + DMO)
\]

and conjunctive normal form

\[
\text{cnf} = \text{innermost}(DOAL + DOAR + DN + DMA + DMO)
\]
One-pass Traversals

- Apply rules in a single traversal over a program tree
- Realization in Stratego

\[
\text{simplify} = \text{downup}(\text{repeat}(R_1 + \ldots + R_n)) \\
\text{simplify} = \text{bottomup}(\text{repeat}(R_1 + \ldots + R_n))
\]

- Example

\[
\text{eval} = \\
\text{bottomup(} \\
\text{repeat(} \\
T_1 + T_2 + T_3 + T_4 + T_5 + T_6 + T_7 + T_8 + T_9 + T_{10} + T_{11} + T_{12} + T_{13} + T_{14} + T_{15} + T_{16} + T_{17} \\
\text{)} \\
\text{)}
\]
One-pass Traversals (More Examples)

rules

DefN : Not(x) -> Impl(x, False)
DefI : Impl(x, y) -> Or(Not(x), y)
DefE : Eq(x, y) -> And(Impl(x, y), Impl(y, x))
DefO1 : Or(x, y) -> Impl(Not(x), y)
DefO2 : Or(x, y) -> Not(And(Not(x), Not(y)))
DefA1 : And(x, y) -> Not(Or(Not(x), Not(y)))
DefA2 : And(x, y) -> Not(Impl(x, Not(y)))
IDefI : Or(Not(x), y) -> Impl(x, y)
IDefE : And(Impl(x, y), Impl(y, x)) -> Eq(x, y)

strategies

desugar = topdown(try(DefI + DefE))
impl-nf = topdown(repeat(DefN + DefA2 + DefO1 + DefE))
Staged Transformations

- Transformations are not applied to a subject term all at once, but rather in stages.
- In each stage, only rules from some particular subset of the entire set of available rules are applied.
- Realization in Stratego:

  ```stratego
simplify =
  innermost(A1 + ... + Ak)
  ; innermost(B1 + ... + Bl)
  ; ...
  ; innermost(C1 + ... + Cm)
  ``

- Implements ‘sequences of normal forms’ of TAMPR.
‘Local’ Transformations

• Apply rules only to selected parts of the subject program
• Realization in Stratego

transformation =
   alltd(
      trigger-transformation
   ; innermost(A1 + ... + An)
   )
Conclusion

• Term rewriting is a nice paradigm for program transformation
  – separation of rules and strategies
  – implicit traversal

• Lack of control can be problematic

• A solution should provide
  – control over application of rules
  – maintain separation of rules and strategies
  – avoid overhead for traversal specification

• This is the goal of the programmable rewriting strategies approach of Stratego