Traversals Strategies
Program Transformation 2004–2005

Eelco Visser

Institute of Information & Computing Sciences
Utrecht University,
The Netherlands

February 17, 2005
Part I

In Control of Rewriting
Term Rewriting for Program Transformation

- apply set of rewrite rules exhaustively

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
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))
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:
  $f : p_1 \rightarrow p'_2$
  applying $f$ recursively in the right-hand side.
- Normalize terms $f(t)$ with respect to these rules.
- The function now controls where rules are applied.
Recursive Rewrite Rules: Map

map(s) : [] -> []
map(s) : [x | xs] -> [<s> | <map(s)> xs]
Recursive Rewrite Rules: Constant Folding

Constant folding rules

Eval : Plus(Int(i), Int(j)) → Int(<addS>(i,j))
Eval : Times(Int(i), Int(j)) → Int(<mulS>(i,j))

Constant folding entire tree

fold : Int(i) → Int(i)
fold : Var(x) → Var(x)
fold : Plus(e1, e2) → <try(Eval)>Plus(<fold>e1, <fold>e2)
fold : Times(e1, e2) → <try(Eval)>Times(<fold>e1, <fold>e2)

Traversing and application of rules are tangled

http://www.strategoxt.org

Traversal Strategies
Recursive Rewrite Rules: Disjunctive Normal Form

\[
\text{dnf} : \text{True} \rightarrow \text{True} \\
\text{dnf} : \text{False} \rightarrow \text{False} \\
\text{dnf} : \text{Atom}(x) \rightarrow \text{Atom}(x) \\
\text{dnf} : \text{Not}(x) \rightarrow \text{not}(<\text{dnf}>x) \\
\text{dnf} : \text{And}(x,y) \rightarrow <\text{and}>(<\text{dnf}>x,<\text{dnf}>y) \\
\text{dnf} : \text{Or}(x,y) \rightarrow \text{Or}(<\text{dnf}>x,<\text{dnf}>y)
\]

\[
\text{and1} : (\text{Or}(x,y),z) \rightarrow \text{Or}(<\text{and}>(x,z),<\text{and}>(y,z)) \\
\text{and2} : (z,\text{Or}(x,y)) \rightarrow \text{Or}(<\text{and}>(z,x),<\text{and}>(z,y)) \\
\text{and3} : (x,y) \rightarrow \text{And}(x,y) \\
\text{and} = \text{and1} <+ \text{and2} <+ \text{and3}
\]

\[
\text{not1} : \text{Not}(x) \rightarrow x \\
\text{not2} : \text{And}(x,y) \rightarrow \text{Or}(<\text{not}>(x),<\text{not}>(y)) \\
\text{not3} : \text{Or}(x,y) \rightarrow <\text{and}>(<\text{not}>(x),<\text{not}>(y)) \\
\text{not4} : x \rightarrow \text{Not}(x) \\
\text{not} = \text{not1} <+ \text{not2} <+ \text{not3} <+ \text{not4}
\]
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.
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>
Requirements

- Control over application of rules
- No traversal overhead
- Separation of rules and strategies
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
  
  \[
  \text{simplify} = \text{innermost}(R_1 \leftarrow \ldots \leftarrow R_n)
  \]

- Example: disjunctive normal form
  
  \[
  \text{dnf} = \text{innermost}(\text{DAOL} \leftarrow \text{DAOR} \leftarrow \text{DN} \leftarrow \text{DMA} \leftarrow \text{DMO})
  \]

  and conjunctive normal form
  
  \[
  \text{cnf} = \text{innermost}(\text{DOAL} \leftarrow \text{DOAR} \leftarrow \text{DN} \leftarrow \text{DMA} \leftarrow \text{DMO})
  \]
One-pass Traversals

- Apply rules in a single traversal over a program tree
- Realization in Stratego
  
  \[
  \text{simplify} = \text{downup}(\text{repeat}(R1 <+ ... <+ Rn))
  \]
  
  \[
  \text{simplify} = \text{bottomup}(\text{repeat}(R1 <+ ... <+ Rn))
  \]

- Example

  \[
  \text{eval} = \\
  \text{bottomup}(\\
  \text{repeat}(\\
  \text{T1 <+ T2 <+ T3 <+ T4 <+ T5 <+ T6 <+ T7 <+ T8 <+ T9 <+ T10 <+ T11 <+ T12 <+ T13 <+ T14 <+ T15 <+ T16 <+ T17})
  )
  )
  \]
One-pass Traversals (More Examples)

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))
Def01 : Or(x, y) → Impl(Not(x), y)
Def02 : 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)

desugar = topdown(try(DefI <+ DefE))

impl-nf = topdown(repeat(DefN <+ DefA2 <+ Def01 <+ DefE))
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

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

http://www.strategoxt.org

Traversal Strategies
‘Local’ Transformations

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

```plaintext
transformation =
   alltd(
      trigger-transformation
   ; innermost(A1 <+ ... <+ An)
   )
```

http://www.strategoxt.org

Traversal Strategies
Part II

Realizing Term Traversal
Realizing Term Traversal

Requirements

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

Many ways to traverse a tree

- Bottom-up
- Top-down
- Innermost
- ...

What are the primitives of traversal?
<table>
<thead>
<tr>
<th>Traversal Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One-Level Traversal Operators</strong></td>
</tr>
<tr>
<td>Apply a strategy to one or more direct subterms</td>
</tr>
<tr>
<td><strong>Congruence: Data-Type Specific Traversal</strong></td>
</tr>
<tr>
<td>Apply a different strategy to each argument of a specific constructor</td>
</tr>
<tr>
<td><strong>Generic Traversal</strong></td>
</tr>
<tr>
<td>All</td>
</tr>
<tr>
<td>- Apply to all direct subterms</td>
</tr>
<tr>
<td>One</td>
</tr>
<tr>
<td>- Apply to one direct subterm</td>
</tr>
<tr>
<td>Some</td>
</tr>
<tr>
<td>- Apply to as many direct subterms as possible, and at least one</td>
</tr>
</tbody>
</table>
Congruence Operator: Data-type Specific Traversal

- Syntax: $c(s_1, \ldots, s_n)$
  for each $n$-ary constructor $c$
- Apply strategies to direct sub-terms of a $c$ term

```
Plus(Int("14"),Int("3"))
stratego> Plus(!Var("a"), id)
Plus(Var("a"),Int("3"))
```
**Congruence Operator: Data-type Specific Traversal**

- Syntax: $c(s_1, \ldots, s_n)$
  - for each $n$-ary constructor $c$
- Apply strategies to direct sub-terms of a $c$ term

```plaintext
Plus(Int("14"), Int("3"))
```

```plaintext
stratego> Plus(!Var("a"), id)
Plus(Var("a"), Int("3"))
```

**Example: List Operations**

```plaintext
map(s) = Nil + Cons(s, map(s))
```

```plaintext
fetch(s) = Cons(s, id) <+ Cons(id, fetch(s))
```

```plaintext
filter(s) =
  Nil + (Cons(s, filter(s)) <+ ?Cons(_, <id>); filter(s))
```
Congruence Operator: Data-type Specific Traversal

- Syntax: \( c(s_1, \ldots, s_n) \)
  - for each \( n \)-ary constructor \( c \)
- Apply strategies to direct sub-terms of a \( c \) term

\[
\text{Plus}(\text{Int}("14"), \text{Int}("3"))
\]

```
stratego> \text{Plus}(!\text{Var}("a"), \text{id})
\text{Plus}(\text{Var}("a"), \text{Int}("3"))
```

Example: List Operations

\[
\text{map}(s) = [] + [s | \text{map}(s)]
\]

\[
\text{fetch}(s) = [s | \text{id}] <+ [\text{id} | \text{fetch}(s)]
\]

\[
\text{filter}(s) =
\quad [] + ([s | \text{filter}(s)] <+ ?[\_|<\text{id}>]; \text{filter}(s))
\]

http://www.strategoxt.org

Traversal Strategies
Example: Constant Folding

\[
\text{BinOp}(\text{PLUS}(), \text{Int}("14"), \text{Int}("3"))
\]

\text{stratego}\>
\text{EvalBinOp}
\text{Int}("17")
**Example: Constant Folding**

\[
\text{BinOp(PLUS(), Int("14"), Int("3"))}
\]

\texttt{stratego> EvalBinOp}

\texttt{Int("17")}

\[
\text{const-fold =}
\]

\texttt{BinOp(id, const-fold, const-fold); try(EvalBinOp)}

\texttt{<+ Call(id, map(const-fold)); try(EvalCall)}

\texttt{<+ If(const-fold, const-fold, const-fold); try(EvalIf)}
Example: Constant Folding

BinOp(PLUS(), Int("14"), Int("3"))

stratego> EvalBinOp
Int("17")

const-fold =
    BinOp(id, const-fold, const-fold); try(EvalBinOp)
    <+ Call(id, map(const-fold)); try(EvalCall)
    <+ If(const-fold, const-fold, const-fold, const-fold); try(EvalIf)

BinOp(TIMES(),
    If(BinOp(PLUS(), Int("14"), Int("3")), Int("2"), Int("23")), Int("4"))
)

stratego> const-fold
Int("8")
Format Checking

Example

Format checking

- Describe a subset of a term language using a recursive pattern

Applications

- Verify output (testing)
- Verify input (pre-condition)
- Documentation

http://www.strategoxt.org
- Describe format of terms in normal form
  - \( \text{conj}(s) = \text{rec } x(\text{And}(x, x) <+ s) \)
    - Conjunct
  - \( \text{disj}(s) = \text{rec } x(\text{Or}(x, x) <+ s) \)
    - Disjunct
  - \( \text{conj}-\text{nf} = \text{conj}(\text{disj}(\text{Not}(\text{Var}(x)) + \text{Var}(x))) \)
    - Conjunctive normal form
  - \( \text{disj}-\text{nf} = \text{disj}(\text{conj}(\text{Not}(\text{Var}(x)) + \text{Var}(x))) \)
    - Disjunctive normal form
Generic Traversal

Data-type specific traversal requires tedious enumeration of cases

Even if traversal behaviour is uniform

Generic traversal allows concise specification of default traversals
Visiting All Subterms

- Syntax: `all(s)`
- Apply strategy `s` to all direct sub-terms

```plaintext
Plus(Int("14"), Int("3"))
stratego> all(!Var("a"))
Plus(Var("a"), Var("a"))
```

http://www.strategoxt.org
Visiting All Subterms

- Syntax: `all(s)`
- Apply strategy `s` to all direct sub-terms

Plus(Int("14"), Int("3"))

```stratego>
all(!Var("a"))
Plus(Var("a"), Var("a"))
```

```plaintext
bottomup(s) = all(bottomup(s)); s
topdown(s)  = s; all(topdown(s))
downup(s)  = s; all(downup(s)); s
alltd(s)   = s <+ all(alltd(s))
```
Visiting All Subterms

- Syntax: all(s)
- Apply strategy s to all direct sub-terms

Plus(Int("14"), Int("3"))

stratego> all(!Var("a"))
Plus(Var("a"), Var("a"))

bottomup(s) = all(bottomup(s)); s
topdown(s) = s; all(topdown(s))
downup(s) = s; all(downup(s)); s
alltd(s) = s <-> all(alltd(s))

const-fold =
    bottomup(try(EvalBinOp <-> EvalCall <-> EvalIf))
Example: Desugaring Expressions

DefAnd : \text{And}(e_1, e_2) \rightarrow \text{If}(e_1, e_2, \text{Int}("0"))

DefPlus : \text{Plus}(e_1, e_2) \rightarrow \text{BinOp}(\text{PLUS}(), e_1, e_2)

DesugarExp = \text{DefAnd} \leftarrow \text{DefPlus} \leftarrow \ldots

desugar = \text{topdown}(\text{try}(\text{DesugarExp}))

\begin{verbatim}
IfThen(
    \text{And}(\text{Var}("a"),\text{Var}("b")),
    \text{Plus}(\text{Var}("c"),\text{Int}("3")))
\end{verbatim}

\texttt{stratego}\> desugar
\begin{verbatim}
IfThen(
    \text{If}(\text{Var}("a"),\text{Var}("b"),\text{Int}("0")),
    \text{BinOp}(\text{PLUS},\text{Var}("c"),\text{Int}("3")))
\end{verbatim}
Fixed-point Traversal

innermost(s) = bottomup(try(s; innermost(s)))

Normalization

dnf = innermost(DAOL <+ DAOR <+ DN <+ DMA <+ DMO)
cnf = innermost(DOAL <+ DOAR <+ DN <+ DMA <+ DMO)
Visiting One Subterms

- Syntax: `one(s)`
- Apply strategy `s` to exactly one direct sub-terms

```plaintext
Plus(Int("14"),Int("3"))
stratego> one(!Var("a"))
Plus(Var("a"),Var("a"))
```
Visiting One Subterms

- Syntax: one(s)
- Apply strategy s to exactly one direct sub-terms

```
Plus(Int("14"), Int("3"))
stratego> one(!Var("a"))
Plus(Var("a"), Var("a"))
```

```
onctd(s) = s <+ one(onctd(s))
oncebu(s) = one(oncebu(s)) <+ s
spinetd(s) = s; try(one(spinetd(s)))
spinebu(s) = try(one(spinebu(s))); s
```
Visiting One Subterms

- Syntax: `one(s)`
- Apply strategy `s` to exactly one direct sub-terms

Plus(Int("14"),Int("3"))

```plaintext
stratego> one(!Var("a"))
Plus(Var("a"),Var("a"))
```

```plaintext
oncetd(s) = s <+ one(oncetd(s))
oncebu(s) = one(oncebu(s)) <+ s
spinetd(s) = s; try(one(spinetd(s)))
spinebu(s) = try(one(spinebu(s))); s
```

contains(|t) = oncetd(?t)
Visiting One Subterms

- Syntax: one(s)
- Apply strategy s to exactly one direct sub-terms

Plus(\text{Int}("14"), \text{Int}("3"))

\text{stratego}> \text{one}(\text{!Var("a")})

Plus(\text{Var("a")}, \text{Var("a")})

\text{oncetd}(s) = s <\leftrightarrow one(oncetd(s))
\text{oncebu}(s) = one(oncebu(s)) <\leftrightarrow s
\text{spinetd}(s) = s; \text{try}(\text{one}(\text{spinetd}(s)))
\text{spinebu}(s) = \text{try}(\text{one}(\text{spinebu}(s))); s

\text{contains}(|t|) = \text{oncetd}(?t)

\text{reduce}(s) = \text{repeat}(\text{one}(\text{reduce}(s)) + s)
\text{outermost}(s) = \text{repeat}(\text{oncetd}(\text{outermost}(s)))
\text{innermostI}(s) = \text{repeat}(\text{oncebu}(\text{innermostI}(s)))
One-pass traversals

\[ \text{sometd}(s) = s \oplus \text{some}(\text{sometd}(s)) \]
\[ \text{somebu}(s) = \text{some}(\text{somebu}(s)) \oplus s \]

Fixpoint traversals

\[ \text{reduce-par}(s) = \text{repeat}(\text{some}(\text{reduce-par}(s)) + s) \]
Summary

- Tangling of rules and strategy (traversal) considered harmful
- Separate traversal from rules
- One-level traversal primitives allow wide range of traversals