Strategic Tree Rewriting in Attribute Grammars

Lucas Kramer and Eric Van Wyk

Department of Computer Science & Engineering
University of Minnesota

SLE ’20
November 15, 2020
Virtual Conference

Slides available at z.umn.edu/strag
Problem

- (Strategic) term rewriting
  - ✓ Transformations (e.g. optimizing $x + 0 \rightarrow x$)
  - ✗ Analyses (e.g. free variables, type checking)

- Attribute grammars
  - ✓ Analyses
  - ✗ Transformations - requires boilerplate for all productions!

- Both approaches
  - ✗ Contextual transformations (e.g. inlining `let x = 7 in x + y \rightarrow 7 + y`)

- Most language engineering tasks involve both analyses & transformations
Solution

- Rewriting on attribute-decorated *trees*, rather than undecorated *terms*

- Attributes carry contextual information and perform ancillary computations

- Rewrite rules can access attributes

- Strategies (à la *Stratego*) control the application of rules

- Generate attribute equations from rules and strategies
Example

Consider performing optimizations in a simple functional language

```haskell
let a = 1 + 2; b = -a in a - b
```

This can be represented in abstract syntax as

```plaintext
letE(seq(decl("a", add(const(1), const(2))),
        decl("b", neg(var("a")))),
        sub(var("a"), var("b")))
```
We can define attributes on this language

```plaintext
synthesized attribute freeVars::[String];
inherited attribute usedVars::[String];
synthesized attribute defs::[Pair<String Maybe<Expr>>];
inherited attribute env::[Pair<String Maybe<Expr>>];

nonterminal Expr with env, freeVars;
production var e::Expr ::= id::String
{ e.freeVars = [id]; }

production letE e::Expr ::= ds::Decls e1::Expr
{ e.freeVars = ds.freeVars ++
    removeAll(map(fst, ds.defs), e1.freeVars);
    ds.usedVars = e.freeVars;
    ds.env = top.env;
    e1.env = ds.defs ++ top.env; }
```
Example: Rewrite rules

- Optimizations can be concisely expressed as rewrite rules

\[
\begin{align*}
\text{add}(e, \text{const}(0)) & \rightarrow e \\
\text{add}(\text{const}(0), e) & \rightarrow e \\
\text{add}(\text{const}(a), \text{const}(b)) & \rightarrow \text{const}(a + b) \\
\text{sub}(e_1, e_2) & \rightarrow \text{add}(e_1, \text{neg}(e_2)) \\
\text{neg}(\text{neg}(e)) & \rightarrow e \\
\text{neg}(\text{const}(a)) & \rightarrow \text{const}(\neg a) \\
\text{var}(id) | (id, \text{just}(e)) & \in \text{env} \rightarrow e
\end{align*}
\]

- Rule 7 relies on an environment context
Strategy Attributes: Non-Contextual Rules

```
partial strategy attribute optimizeStep =
  rule on Expr of
  | add(e, const(0)) -> e
  | add(const(0), e) -> e
  | add(const(a), const(b)) -> const(a + b)
  | sub(e1, e2) -> add(e1, neg(e2))
  | neg(neg(e)) -> e
  | neg(const(a)) -> const(-a)
  end
  occurs on Expr;

strategy attribute optimize = -- innermost(optimizeStep)
  all(optimize) <* ((optimizeStep <* optimize) <+ id)
  occurs on Expr, Decls;

propagate optimizeStep on Expr;
propagate optimize on Expr, Decls;
```
**Strategy Attributes: Using Contextual Information**

```plaintext
partial strategy attribute inlineStep =
    rule on top::Expr of
    | var(n) when lookup(n, top.env) matches just(just(e)) -> e
    | letE(empty(), e) -> e
end
<+
    rule on top::Decls of
    | decl(id, e) when !contains(id, top.usedVars) -> empty()
    | seq(d, empty()) -> d
    | seq(empty(), d) -> d
end
occurs on Expr, Decls;

propagate inlineStep on Expr, Decls;
```
Strategy Attributes: Traversal Order with Context

- Misses optimizations (e.g. `let x = 7 in x → let x = 7 in 7` instead of 7):

  ```plaintext
  strategy attribute optimizeInline =
  innermost(optimizeStep <+ inlineStep));
  ```

- Correct, but inefficient:

  ```plaintext
  strategy attribute optimizeInline =
  repeat(onceBottomUp(optimizeStep <+ inlineStep));
  ```

- Better:

  ```plaintext
  strategy attribute optimizeInline =
  ((seq(optimizeInline, id) <* 
  seq(id, optimizeInline) <* 
  seq(optimizeInline, id)) <+ 
  (letE(optimizeInline, id) <* 
  letE(id, optimizeInline) <* 
  letE(optimizeInline, id)) <+ all(optimizeInline)) <* 
  (((optimizeStep <+ inlineStep) <* optimizeInline) <+ id));
  ```
Implementation

- Strategy attributes ⇒ higher-order attributes
- propagate declarations ⇒ aspect productions with generated equations
Implementation: Rules

```
partial strategy attribute optimizeStep = rule on Expr of ... end;
propagate optimizeStep on Expr;

synthesized attribute optimizeStep<α>::Maybe<α>;
attribute optimizeStep<Expr> occurs on Expr;

aspect production add top::Expr ::= e1::Expr e2::Expr
{ top.optimizeStep =
  case top of
    | add(e, const(0)) -> just(e)
    | add(const(0), e) -> just(e)
    | add(const(a), const(b)) -> just(const(a + b))
    | _ -> nothing() end;
}
aspect production const top::Expr ::= i::Integer
{ top.optimizeStep = nothing(); }
```
Implementation: Lifting Sequence

```plaintext
strategy attribute optimize =
   all(optimize) <* ((optimizeStep <* optimize) <+ id)
occurs on Expr, Decls;

propagate optimize on Expr, Decls;

↓

strategy attribute optimize = all(optimize) <* optimize_snd
occurs on Expr, Decls;

strategy attribute optimize_snd = (optimizeStep <* optimize) <+ id
occurs on Expr, Decls;

propagate optimize, optimize_snd on Expr, Decls;
```
Implementation: Total Sequence, all

```
strategy attribute optimize = all(optimize) <> optimize_snd
  occurs on Expr;
propagate optimize on Expr;

⇓

synthesized attribute optimize<a>::a;
attribute optimize<Expr> occurs on Expr;

aspect production add  top::Expr ::= e1::Expr e2::Expr
{ top.optimize =
  decorate add(e1.optimize, e2.optimize)
  with {env = top.env;}.optimize_snd;
}
aspect production const top::Expr ::= i::Integer
{ top.optimize = top.optimize_snd; }
```
Implementation: Partial Sequence, Choice

strategy attribute optimize_snd = (optimizeStep <* optimize) <+ id occurs on Expr;
propagate optimize_snd on Expr;

⇓
synthesized attribute optimize_snd<a>::a;
attribute optimize_snd<Expr> occurs on Expr;

aspect production add top::Expr ::= e1::Expr e2::Expr
{ top.optimize_snd =
  case top.optimizeStep of
  | just(a) -> decorate a with {env = top.env;}.optimize
  | nothing() -> just(top)
  end;
}
aspect production const top::Expr ::= i::Integer
{ top.optimize_snd = top; }

Applications

- $\lambda$-calculus
  - Inspired by Stratego and Kiama examples

- Regex matching with Brzozowski derivatives
  - Use strategy attributes to simplify regexes

- Normalizing for-loops

- Optimizing strategy expressions before translation
Applications: for-Loop Normalization

- Strategy attributes are useful in building language extensions, e.g. normalizing for-loops

- Can use C concrete syntax in rules

```
partial strategy attribute preprocessLoop =
  rule on Stmt of
  | ableC_Stmt{
    for ($Decl{init}; $Name{i} <= $Expr{limit}; $Expr{iter})
      $Stmt{b}
  } ->
  ableC_Stmt{
    for ($Decl{init}; $Name{i} < $Expr{limit} + 1; $Expr{iter})
      $Stmt{b}
  }
  | ...
  end;
```
Applications: Optimizing Strategy Expressions

partial strategy attribute genericStep =
  rule on StrategyExpr of
  | sequence(fail(), _) -> fail()
  | sequence(id(), s) -> s
  | choice(s, _) when s.isTotal -> s
  | allTraversal(id()) -> id()
  | ...
end;

partial strategy attribute prodStep =
  rule on StrategyExpr of
  | rewriteRule(_, _, r) when !r.matchesFrame -> fail()
  | ...
end;
Strategy attributes provide a compelling, seamless integration between strategic term rewriting and attribute grammars.

Proper interaction with other attribute features (e.g. forwarding) makes them appealing for use in implementing modular language extensions.

Future work
- Spotting performance issues due to repeated decoration
- Other patterns of propagated attributes (e.g. monoid, chained, equality, etc.)
Please Stay for Question Time

Lucas Kramer  
krame505@umn.edu

Eric Van Wyk  
evw@umn.edu

MELT Research Group  
melt.cs.umn.edu

Slides available at z.umn.edu/strag
<table>
<thead>
<tr>
<th>Totality analysis</th>
<th>λ-Calculus</th>
<th>Regex Matching with Derivatives</th>
<th>Optimizing Strategy Expressions</th>
</tr>
</thead>
</table>

Backup
## Totality Analysis

<table>
<thead>
<tr>
<th>Context</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash id ; total$</td>
<td><strong>Id</strong></td>
<td>$\Gamma \vdash s_1 ; total$</td>
</tr>
<tr>
<td>$\Gamma \vdash s_2 ; total$</td>
<td></td>
<td>$\Gamma \vdash id ; total$</td>
</tr>
<tr>
<td>$\Gamma \vdash s_1 ;&lt;/strong&gt; s_2 ; total$</td>
<td><strong>SEQ</strong></td>
<td>$\Gamma \vdash s_1 ; total$</td>
</tr>
<tr>
<td>$\Gamma \vdash s_2 ; total$</td>
<td></td>
<td>$\Gamma \vdash s_1 ;&lt;/strong&gt; s_2 ; total$</td>
</tr>
<tr>
<td>$\Gamma \vdash s_1 ;&lt;/strong&gt; s_2 ; total$</td>
<td><strong>CHOICE L</strong></td>
<td>$\Gamma \vdash s_1 ; total$</td>
</tr>
<tr>
<td>$\Gamma \vdash s_2 ; total$</td>
<td></td>
<td>$\Gamma \vdash s_1 ;&lt;/strong&gt; s_2 ; total$</td>
</tr>
<tr>
<td>$\Gamma \vdash s_1 ;&lt;/strong&gt; s_2 ; total$</td>
<td><strong>CHOICE R</strong></td>
<td>$\Gamma \vdash s_1 ; total$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Gamma \vdash s_2 ; total$</td>
</tr>
<tr>
<td>$\Gamma \vdash s ; total$</td>
<td><strong>ALL</strong></td>
<td>$\Gamma \vdash s ; total$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Gamma \vdash all(s) ; total$</td>
</tr>
<tr>
<td>$n \in \Gamma$</td>
<td><strong>REF</strong></td>
<td>$\Gamma \vdash n ; total$</td>
</tr>
<tr>
<td>$\Gamma \vdash n ; total$</td>
<td></td>
<td>$\Gamma \vdash n ; total$</td>
</tr>
<tr>
<td>$\Gamma \cup {n} \vdash s ; total$</td>
<td><strong>REC</strong></td>
<td>$\Gamma \vdash rec ; n \rightarrow s ; total$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Gamma \vdash rec ; n \rightarrow s ; total$</td>
</tr>
</tbody>
</table>
Applications: $\lambda$-Calculus

```latex
synthesized attribute freeVars::[String];
nonterminal Term with freeVars;
abstract production var
top::Term ::= id::String
{ top.freeVars = [id]; }

abstract production abs
top::Term ::= id::String body::Term
{ top.freeVars = remove(id, body.freeVars); }

abstract production app
top::Term ::= t1::Term t2::Term
{ top.freeVars = t1.freeVars ++ t2.freeVars; }

abstract production letT
top::Term ::= id::String t::Term body::Term
{ top.freeVars = t.freeVars ++ remove(id, body.freeVars); }
```
Applications: \(\lambda\)-Calculus

```plaintext
partial strategy attribute beta =
  rule on Term of
  | app(abs(x, e1), e2) -> letT(x, e2, e1)
end;

partial strategy attribute letDist =
  rule on Term of
  | letT(x, e, var(y)) when x == y -> e
  | letT(x, e, var(y)) -> var(y)
  | letT(x, e0, app(e1, e2)) ->
    app(letT(x, e0, e1), letT(x, e0, e2))
  | letT(x, e1, abs(y, e2)) ->
    let z::String = freshVar() in
    abs(z, letT(x, e1, letT(y, var(z), e2))) end
  | letT(x, _, e) when !contains(x, e.freeVars) -> e
end;
```
Applications: $\lambda$-Calculus

strategy attribute evalInnermost = innermost(beta <+ letDist);

strategy attribute evalWHNF =
  try(app(evalWHNF, evalWHNF) <+
    letT(id, evalWHNF, evalWHNF)) <*
  try((beta <+ letDist) <*> evalWHNF);
Applications: Regex Matching with Derivatives

```plaintext
synthesized attribute nullable::Boolean;
nonterminal Regex with nullable;
abstract production epsilon  top::Regex ::= 
{ top.nullable = true; }
abstract production empty   top::Regex ::= 
{ top.nullable = false; }
abstract production char   top::Regex ::= c::Integer -- UTF-16 char
{ top.nullable = false; }
abstract production seq    top::Regex ::= r1::Regex r2::Regex
{ top.nullable = r1.nullable && r2.nullable; }

abstract production alt    top::Regex ::= r1::Regex r2::Regex
{ top.nullable = r1.nullable || r2.nullable; }

abstract production star   top::Regex ::= r::Regex
{ top.nullable = true; }
```
Applications: Regex Matching with Derivatives

\textbf{synthesized attribute} deriv::Regex occurs on Regex;
\textbf{autocopy attribute} wrt::Integer occurs on Regex;
\textbf{aspect production} epsilon top::Regex ::= 
{ top.deriv = empty(); }
\textbf{aspect production} empty top::Regex ::= 
{ top.deriv = empty(); }
\textbf{aspect production} char top::Regex ::= c::Integer 
{ top.deriv = if c == top.wrt then epsilon() else empty(); }
\textbf{aspect production} seq top::Regex ::= r1::Regex r2::Regex 
{ top.deriv = alt(seq(r1.deriv, r2), 
  if r1.nullable then r2.deriv else empty()); }
\textbf{aspect production} alt top::Regex ::= r1::Regex r2::Regex 
{ top.deriv = alt(r1.deriv, r2.deriv); }
\textbf{aspect production} star top::Regex ::= r::Regex 
{ top.deriv = seq(r.deriv, top); }
Applications: Regex Matching with Derivatives

```latex
strategy attribute simpl = innermost(
  rule on Regex of
  | seq(empty(), r) -> empty()
  | seq(epsilon(), r) -> r
  | alt(empty(), r) -> r
  | alt(epsilon(), r) when r.nullable -> r
  | ... -- Symmetric equivalents of the above
  | star(empty()) -> epsilon()
  | star(epsilon()) -> epsilon()
end);

strategy attribute simplDeriv = deriv <*> simpl;
propagate simpl, simplDeriv on Regex;

function matchStep Regex ::= r::Regex c::Integer
{ r.wrt = c; return c.simplDeriv; }
function matchesRegex Boolean ::= r::Regex s::String
{ return foldl(matchStep, stringToChars(s)).nullable; }
```
Applications: Optimizing Strategy Expressions

```plaintext
strategy attribute simplify = innermost(genericStep);

strategy attribute optimize =
  (sequence(optimize, simplify) <+ choice(optimize, optimize) <+ allTraversal(simplify) <+ someTraversal(simplify) <+ oneTraversal(simplify) <+ recComb(id, optimize) <+ id) <* try((genericStep <+ prodStep) <* optimize);
```