Stack-based Strategic Control

Victor Winter
Department of Computer Science
University of Nebraska at Omaha

Seventh International Workshop on Reduction Strategies in
Rewriting and Programming
Paris, France
Monday June 25, 2007
Outline

Background and Introduction
  Rewriting and Strategic Programming
  Observing the Application of Strategies to Terms
  Control Stacks

A Stack-based Semantics for a Set of Strategic Combinators

Example: Let-block Optimization

Related Work and Conclusion
Background and Introduction
A Classical Rewriting Framework

- Is motivated by the desire to mechanize equational reasoning.
- Rewrite rules represent directed equalities.
- The application of rules to terms is \textit{implicit}, exhaustive, and universal.
- And, the user can control rewriting at the term-level by inhibiting matching/unification.

In this framework:
- the existence of unique normal forms is critical (\textit{confluence}), and
- the ability to always reach normal forms (when they exist) is also critical (\textit{termination}).
In a Classical Strategic Framework

- The application of rules (a.k.a. strategies) to terms is explicit.
- The application of strategies to a single (subject) term can be controlled by:
  - matching/unification - at the term-level
  - conditions - at the rule-level
  - combinators such as: <+ and <; - at the strategy-level

- The application of strategies to term sequences is controlled by iterators.
  - Traversals: top-down left-to-right, bottom-up left-to-right
  - Indefinite Iterators: FIX, Repeat
What is a Strategy?

**Basis:** A conditional rewrite rule is a strategy.

**Induction:** An expression composed of strategies, combinators, and iterators is a strategy.

1.1 **Remark:** One could also consider a term to be a strategy much in the same way that a constant is represented, in term languages, as a nullary function.
## Examples of Strategy Application

<table>
<thead>
<tr>
<th>Rule Label</th>
<th>Rewrite Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_0$:</td>
<td>$a \rightarrow a$</td>
</tr>
<tr>
<td>$r_1$:</td>
<td>$a \rightarrow b$</td>
</tr>
<tr>
<td>$r_2$:</td>
<td>$b \rightarrow c$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Term</th>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_0$</td>
<td>$a$</td>
<td>$\Rightarrow$</td>
<td>$a$</td>
</tr>
<tr>
<td>$r_1 \leftarrow r_2$</td>
<td>$a$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2 \leftarrow r_1$</td>
<td>$a$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_1 &lt;; r_2$</td>
<td>$a$</td>
<td>$\Rightarrow$</td>
<td>$c$</td>
</tr>
</tbody>
</table>
Successful/Unsuccessful Application

- In a strategic framework, standard combinators such as left-biased choice \( <+ \) exercise control over rewriting based on an *abstract view* of strategy application. In particular, the application of a strategy to a term is either successful or unsuccessful.

- This approach assumes the ability to *observe* the successful/unsuccessful nature of strategy application.

- A fundamental question concerns itself with how this observation is made.
Examples of Successful and Unsuccessful Application

\[ r_0: \ a \rightarrow a \]
\[ r_1: \ a \rightarrow b \]
\[ r_2: \ b \rightarrow c \]

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Term</th>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 \leftrightarrow r_2 )</td>
<td>( a )</td>
<td>( \Rightarrow )</td>
<td>( b )</td>
</tr>
<tr>
<td>( r_2 \leftrightarrow r_1 )</td>
<td>( a )</td>
<td>( \Rightarrow )</td>
<td>( b )</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>( a )</td>
<td>( \Rightarrow )</td>
<td>( ? )</td>
</tr>
<tr>
<td>( r_2 \prec; r_1 )</td>
<td>( a )</td>
<td>( \Rightarrow )</td>
<td>( ? )</td>
</tr>
</tbody>
</table>
The unsuccessful application of a strategy \( s \) to a term \( t \) causes the term \( t \) to be rewritten to the term \( \text{FAIL} \) – which is a constant denoting unsuccessful application. In this case, unsuccessful application can be observed at the term level.

\[
\begin{align*}
  r_0 & : a \rightarrow a \\
r_1 & : a \rightarrow b \\
r_2 & : b \rightarrow c
\end{align*}
\]

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Term</th>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 \leftarrow r_2 )</td>
<td>( a )</td>
<td>( \Rightarrow )</td>
<td>( b )</td>
</tr>
<tr>
<td>( r_2 \leftarrow r_1 )</td>
<td>( a )</td>
<td>( \Rightarrow )</td>
<td>( b )</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>( a )</td>
<td>( \Rightarrow )</td>
<td>( \text{FAIL} )</td>
</tr>
<tr>
<td>( r_2 \leftarrow; r_1 )</td>
<td>( a )</td>
<td>( \Rightarrow )</td>
<td>( \text{FAIL} )</td>
</tr>
</tbody>
</table>
Identity-based Solution: $t \rightarrow t$

The unsuccessful application of a strategy $s$ to a term $t$ yields the term $t$. In this case, unsuccessful application cannot be observed at the term level.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Term</th>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1 \leftarrow r_2$</td>
<td>$a$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2 \leftarrow r_1$</td>
<td>$a$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2$</td>
<td>$a$</td>
<td>$\Rightarrow$</td>
<td>$a$</td>
</tr>
<tr>
<td>$r_2 &lt;; r_1$</td>
<td>$a$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
</tbody>
</table>
Control Stacks

In this talk:

- A framework is presented where the operation of strategy application, when viewed abstractly from the perspective of being either successful or unsuccessful, is implicitly stored in an internal structure called a control stack.

- More specifically, I will show how control stacks can be used to formally define the semantics of a variety of standard as well as non-standard combinators that belong to the identity-based strategic programming language TL.
About Stacks

- Stacks, as they are used here, are infinite structures of Boolean values.
- \( \bot \) - denotes an infinite stack of Boolean values, all of which are \textit{false}.
- Stacks are constructed using an infix “dot-notation” whose signature is: \( \text{bool} \times \text{stack} \rightarrow \text{stack} \).
- Stacks are deconstructed using pattern matching.
- Boolean operations are generalized to stacks.

\[ A_1 \lor A_2 \text{ where } A_1 \text{ and } A_2 \text{ are stacks} \]
A Stack-based Semantics for a Set of Strategic Combinators
Core Combinators

\{<+, <; , hide, lift}\cup \{transient, opaque, raise\}

- \{<+, <; , hide, lift\} - these combinators control strategy application. By this I mean that these combinators control which strategies get applied to a given subject term.

- \{transient, opaque, raise\} - these combinators control strategic reduction. In particular, in TL strategies themselves can change during the act of strategy application.
The Control Stacks $\mathcal{A}$ and $\mathcal{R}$

- Two (infinite) stacks of Boolean values will be used to define the semantics of these combinators.

- In this talk:
  - The symbols $\mathcal{A}, \mathcal{A}_1, \mathcal{A}_2, \ldots$ are associated with the stack used to control strategy application.
  - The symbols $\mathcal{R}, \mathcal{R}_1, \mathcal{R}_2, \ldots$ are associated with the stack used to control strategic reduction.
The semantics of combinators will be given in a big-step style with respect to an abstract notation where:

- The application of a strategy $s$ to a term $t$ is denoted $s \cdot \langle t \rangle$.

- The result of a big-step evaluation is a tuple of the form: $\langle A, R, s', t' \rangle$.

where $A$ and $R$ denote control stacks, and $s'$ and $t'$ respectively denote the strategy and term that result from the application $s \cdot \langle t \rangle$. 
Remark: The first two definitions assume that it is possible to determine whether or not the application of a rule $r$ to a term is successful.

\[
\begin{align*}
\text{applies}(r, t) & \quad t \xrightarrow{r} t' \\
\implies r \cdot \langle t \rangle \Downarrow \langle \text{true.} \bot_A, \text{true.} \bot_R, r, t' \rangle & \quad \text{E-successful}
\end{align*}
\]

\[
\begin{align*}
\neg \text{applies}(r, t) & \\
\implies r \cdot \langle t \rangle \Downarrow \langle \bot_A, \bot_R, r, t \rangle & \quad \text{E-unsuccessful}
\end{align*}
\]

\[
\begin{align*}
\text{SKIP} \cdot \langle t \rangle \Downarrow \langle \bot_A, \bot_R, \text{SKIP}, t \rangle & \quad \text{E-skip}
\end{align*}
\]
Two standard combinators: $\leftrightarrow$ and $<$;

\[
\begin{align*}
\text{E-choice1:} & \quad s_1 \cdot \langle t \rangle \Downarrow \langle \text{true}.A, R, s'_1, t' \rangle \\
& \quad (s_1 \leftrightarrow s_2) \cdot \langle t \rangle \Downarrow \langle \text{true}.A, R, s'_1 \leftrightarrow s_2, t' \rangle
\end{align*}
\]

\[
\begin{align*}
\text{E-choice2:} & \quad s_1 \cdot \langle t \rangle \Downarrow \langle \bot.A, R_1, s'_1, t' \rangle \\
& \quad s_2 \cdot \langle t' \rangle \Downarrow \langle A_2, R_2, s'_2, t'' \rangle \\
& \quad (s_1 \leftrightarrow s_2) \cdot \langle t \rangle \Downarrow \langle \bot.A \lor A_2, R_1 \lor R_2, s'_1 \leftrightarrow s'_2, t'' \rangle
\end{align*}
\]

\[
\begin{align*}
\text{E-seq:} & \quad s_1 \cdot \langle t \rangle \Downarrow \langle A_1, R_1, s'_1, t' \rangle \\
& \quad s_2 \cdot \langle t' \rangle \Downarrow \langle A_2, R_2, s'_2, t'' \rangle \\
& \quad (s_1 <; s_2) \cdot \langle t \rangle \Downarrow \langle A_1 \lor A_2, R_1 \lor R_2, s'_1 <; s'_2, t'' \rangle
\end{align*}
\]

2.1 Lemma: \textit{false}.A $\Rightarrow$ A $\equiv$ $\bot$
The “visibility” combinators: \textit{hide} and \textit{lift}

\[
\frac{s \cdot \langle t \rangle \Downarrow \langle x.A, R, s', t' \rangle}{\text{E-hide}}
\]

\[
\frac{\text{hide}(s) \cdot \langle t \rangle \Downarrow \langle A, R, \text{hide}(s'), t' \rangle}{\text{E-hide}}
\]

\[
\frac{s \cdot \langle t \rangle \Downarrow \langle x.A, R, s', t' \rangle}{\text{E-lift}}
\]

\[
\frac{\text{lift}(s) \cdot \langle t \rangle \Downarrow \langle x.x.A, R, \text{lift}(s'), t' \rangle}{\text{E-lift}}
\]

\[
\Gamma \vdash \text{hide}(s_1) \leftrightarrow s_2 \equiv s_1 <; s_2
\]

\[
\Gamma \vdash \text{hide}(\text{lift}(s_1)) \leftrightarrow s_2 \equiv s_1
\]
The combinators: \textit{transient}, \textit{opaque}, and \textit{raise}

\[
\begin{align*}
\frac{s \cdot \langle t \rangle \downarrow \langle A, \text{true}.R, s', t' \rangle}{\text{E-transient1}}
\end{align*}
\]

\[
\begin{align*}
\frac{s \cdot \langle t \rangle \downarrow \langle A, \bot_R, s', t' \rangle}{\text{E-transient2}}
\end{align*}
\]

\[
\begin{align*}
\frac{s \cdot \langle t \rangle \downarrow \langle A, y.R, s', t' \rangle}{\text{E-opaque}}
\end{align*}
\]

\[
\begin{align*}
\frac{s \cdot \langle t \rangle \downarrow \langle A, y.y.R, s', t' \rangle}{\text{E-raise}}
\end{align*}
\]
Iterators: \( \Phi = t_1.t_2.t_3 \cdots \)

\[
s \cdot \langle \text{end} \rangle \Downarrow \langle \bot_A, \bot_R, s, \text{end} \rangle
\]

\[
s \cdot \langle t_i \rangle \Downarrow \langle A_1, R_1, s', t'_i \rangle \quad s' \cdot \Phi_{i+1} \Downarrow \langle A_2, R_2, s'', \Phi'_{i+1} \rangle
\]

\[
s \cdot \langle t_i.\Phi_{i+1} \rangle \Downarrow \langle A_1 \lor A_2, R_1 \lor R_2, s'', t'_i.\Phi'_{i+1} \rangle
\]

Within an identity-based framework, a consequence of these definitions is that the observation of strategy application extends over iterators (e.g., traversals).

\[
BUL\{\text{property}\} \leftrightarrow \text{unfold}
\]
Example: Let-block Optimization
Let-block Optimization

Goal

In-line the expression bound to the variable declared in a let-block, but only if the declared variable occurs no more than once in the body of the let-block.

Assumption

There is only one declaration per let-block.

\[
\text{let val } id = \text{ expr in expr end}
\]

Assumption

All declared variables are unique.
Concrete Example Showing Cases to be Considered

let
  val x = let
    val y = 2
    in
    5 + 4
    end
  in
  let
    val z = x * 3
    in
    z + z
  end
end

⇒

let
  val z = ( 5 + 4 ) * 3
  in
  z + z
end;
A Quick Overview of TL - *terms* and *patterns*

- TL is a strategic programming language designed to manipulate *parse trees*, which we also refer to as *terms*.

- TL provides a notation for describing parse tree structures relative to a given (assumed) grammar $G$.

- Trees expressed using this notation are referred to as *patterns*.

- A *pattern* is either a subscripted nonterminal $B_1$ or an expression of the form $B[\alpha']$ which is well-formed if:
  - $B \not\rightarrow \alpha$, and
  - $\alpha'$ is obtained from $\alpha$ by subscripting all nonterminals occurring in $\alpha$.

- Subscripted nonterminals play an important role because they are treated as variables from the perspective of matching.
A Quick Overview of TL - Example

\[
\begin{align*}
\text{eval\_list} & ::= (\text{dec} [\text{";"}] | \text{expr} \text{";"}) \text{eval\_list} | \epsilon \\
\text{dec} & ::= \text{"val" id "\=" expr} | \ldots \\
\text{expr} & ::= \text{id} | \text{let\_block} | \ldots \\
\text{let\_block} & ::= \text{"let" dec "in" expr "end"} \\
\text{id} & ::= \text{identifier}
\end{align*}
\]

\[\text{expr[let val id} = \text{expr}_1 \text{ in expr}_2 \text{ end]}\]
A Quick Overview of TL - *conditional rewrite rules*

A first-order rewrite rule has the following syntactic structure:

\[ lhs \rightarrow rhs \ [ \text{if condition} ] \]

where

▶ *lhs* is a *pattern*,

▶ *rhs* is a *strategic expression* – and by that I mean an expression whose evaluation yields a term,

▶ [ and ] are syntactic meta symbols indicating that the enclosed section (i.e., the conditional portion) of a rule is optional, and

▶ *condition* is an expression consisting of one or more *match expressions* combined using Boolean connectives.
In this context, a *match expression* is an explicit first-order match between two patterns. For example, let $t_1$ denote a pattern, possibly non-ground, and let $t_2$ denote a ground pattern. The expression $t_1 \ll t_2$ denotes a match expression and evaluates to *true* if and only if a substitution $\sigma$ can be constructed so that $\sigma(t_1) = t_2$.

The substitution $\sigma$ is a mapping from subscripted nonterminals to ground terms.
A Quick Overview of TL - Example

\[
\begin{align*}
\text{eval_list} &::= (\text{dec [";" ] | expr ";" }) \text{ eval_list } | \epsilon \\
\text{dec} &::= \text{"val" id "=" expr } | \ldots \\
\text{expr} &::= \text{id } | \text{let_block } | \ldots \\
\text{let_block} &::= \text{"let" dec "in" expr "end"} \\
\text{id} &::= \text{identifier}
\end{align*}
\]

\[
\text{expr[\text{let val id}_1 = expr}_1 \text{ in expr}_2 \text{ end]} \\
\leq\leq \\
\text{expr[\text{let val x = 1 in x + 5 end]}]
\]

\[
\sigma = [id_1 \leftrightarrow x, \text{expr}_1 \leftrightarrow 1, \text{expr}_2 \leftrightarrow x + 5]
\]
Unfolding

\[
expr[let \ val \ id_1 = expr_1 \ in \ expr_2 \ end] \\
\rightarrow BUL{expr[id_1] \rightarrow expr[\(expr_1\)]}(expr_2)
\]

- This strategy can only be successfully applied to a let-block.
- When applied to a let-block it will return a result that is obtained by traversing the body of the let-block \(expr_2\) and replacing all occurrences of \(id_1\) with \((expr_1)\).
Checking a Property

Next, we want to develop an iterator $BUL\{s\}$ to search a term structure looking for two or more occurrences of a given term. In particular:

- The application of the iterator should be *unsuccessful* if less-than two occurrences of the term are encountered;
- otherwise the application of the iterator should be *successful*.
- If this can be accomplished, then unfolding can be controlled using the following strategy:

  $BUL\{s\} \leftarrow unfold$

So in other words, unfolding only occurs if the application of the iterator $BUL\{s\}$ is seen as being unsuccessful.
A Sketch: $BUL\{s\} \leftrightarrow \text{unfold}$

\[
\begin{align*}
\text{\textit{BUL}}\{ & \text{\textit{hide}}( \\
& \quad \text{\textit{transient}}(\text{\textit{expr}}[id_1] \rightarrow \text{\textit{expr}}[id_1])) \\
& \quad \text{\textit{lift}}(\text{\textit{expr}}[id_1] \rightarrow \text{\textit{expr}}[id_1])) \\
& ) \\
\}
\end{align*}
\]

\[(\text{\textit{expr}}[\textit{let \ val \ id}_1 = \text{\textit{expr}}_1 \ \text{\textit{in}} \ \text{\textit{expr}}_2 \ \text{\textit{end}}])\]

\[
\Phi = t_1, t_2, \ldots, \quad \downarrow t_i, \ldots, \quad \downarrow t_j, \ldots \quad \text{where } i < j
\]
optimize let blocks: \[ BUL\{simplify\_let \prec; \text{cleanup}\} \]

simplify let: \[ \text{expr}_0 \rightarrow (BUL\{\text{check}[id_1]\} \prec \text{unfold})(\text{expr}_0) \]  
if \[ \text{expr}_0 \gg \text{expr}[\text{let val id}_1 = \text{expr}_1 \text{ in } \text{expr}_2 \text{ end}] \]

identity: \[ id_1 \rightarrow \text{expr}[id_1] \rightarrow \text{expr}[id_1] \]

check: \[ id_1 \rightarrow \text{hide}(\text{transient} (\text{identity}[id_1]) \prec \text{lift}(\text{identity}[id_1])) \]

unfold: \[ \text{expr}[\text{let val id}_1 = \text{expr}_1 \text{ in } \text{expr}_2 \text{ end}] \]  
\rightarrow \[ BUL\{\text{expr}[id_1] \rightarrow \text{expr}[[\text{expr}_1]]\}(\text{expr}_2) \]

cleanup: \[ \ldots \]
Related Work and Conclusion
Related Work

- Virtually all languages offer some mechanism to describe nonstandard control flows that can be used to escape from nested computations.
  - Older mechanisms: goto, break, continue, return
  - Newer mechanisms: throwing and catching exceptions
  - Exotic mechanisms: call-with-current-continuation (call/cc), dynamic wind

- Stratego supports a transient-like behavior.
Related Work

▶ There are also a number of systems that have identity-based similarities

▶ In *TOM*, all strategies are seen as either an extension of the Identity strategy or the Fail strategy.

▶ The *Conditional Transformation Core* (CTC) is a logic-based system that supports OR-sequences as well as AND-sequences.

▶ *ASF+SDF* is a rewriting system that has been extended with some strategic ideas (i.e., a fixed set of generic traversals).

▶ The $\rho$-calculus is a fully general higher-order framework in which strategies can be applied to other strategies and yield strategy sets as their results.
Conclusion

In TL,

- a rich environment is provided in which the interplay between dynamic strategy creation and strategic reduction are brought together in an identity-based framework,
- strategy application is extended over the domain of iterators, and
- non-standard combinators (e.g., transient, hide, opaque, lift, raise) enable refined control of rewriting, especially in the context of dynamic strategy generation.