

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 *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 (*confluence*), and
 - ▶ the ability to always reach normal forms (when they exist) is also critical (*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: $\leftarrow+$ and $\leftarrow;$ - 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

Rule Label	Rewrite Rule
r_0 :	$a \rightarrow a$
r_1 :	$a \rightarrow b$
r_2 :	$b \rightarrow c$

Strategy	Term	Transformation	Result
r_0	a	\Rightarrow	a
$r_1 <+ r_2$	a	\Rightarrow	b
$r_2 <+ r_1$	a	\Rightarrow	b
$r_1 <; r_2$	a	\Rightarrow	c

Successful/Unsuccessful Application

- ▶ In a strategic framework, standard combinators such as left-biased choice ($\langle + \rangle$) 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

$r_0: a \rightarrow a$

$r_1: a \rightarrow b$

$r_2: b \rightarrow c$

Strategy	Term	Transformation	Result
$r_1 <+ r_2$	a	\Rightarrow	b
$r_2 <+ r_1$	a	\Rightarrow	b
r_2	a	\Rightarrow	?
$r_2 <; r_1$	a	\Rightarrow	?

Failure-based Solution: $t \rightarrow FAIL$

The unsuccessful application of a strategy s to a term t causes the term t to be rewritten to the term $FAIL$ – which is a constant denoting unsuccessful application. In this case, unsuccessful application can be observed at the term level.

$r_0: a \rightarrow a$
$r_1: a \rightarrow b$
$r_2: b \rightarrow c$

Strategy	Term	Transformation	Result
$r_1 <+ r_2$	a	\Rightarrow	b
$r_2 <+ r_1$	a	\Rightarrow	b
r_2	a	\Rightarrow	FAIL
$r_2 <; r_1$	a	\Rightarrow	FAIL

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.

$$r_0: a \rightarrow a$$

$$r_1: a \rightarrow b$$

$$r_2: b \rightarrow c$$

Strategy	Term	Transformation	Result
$r_1 <+ r_2$	a	\Rightarrow	b
$r_2 <+ r_1$	a	\Rightarrow	b
r_2	a	\Rightarrow	a
$r_2 <; r_1$	a	\Rightarrow	b

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.
- ▶ \perp - denotes an infinite stack of Boolean values, all of which are *false*.
- ▶ Stacks are constructed using an infix “dot-notation” whose signature is: $bool \times stack \rightarrow stack$.
- ▶ Stacks are deconstructed using pattern matching.

$x.A$

- ▶ Boolean operations are generalized to stacks.

$\mathcal{A}_1 \vee \mathcal{A}_2$ where \mathcal{A}_1 and \mathcal{A}_2 are stacks

A Stack-based Semantics for a Set of Strategic Combinators

Core Combinators

$$\{\langle +, \langle ;, \textit{hide}, \textit{lift} \rangle\} \cup \{\textit{transient}, \textit{opaque}, \textit{raise}\}$$

- ▶ $\{\langle +, \langle ;, \textit{hide}, \textit{lift} \rangle\}$ - these combinators control **strategy application**. By this I mean that these combinators control which strategies get applied to a given subject term.
- ▶ $\{\textit{transient}, \textit{opaque}, \textit{raise}\}$ - 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, \dots are associated with the stack used to control **strategy application**.
 - ▶ The symbols \mathcal{R} , \mathcal{R}_1 , \mathcal{R}_2, \dots are associated with the stack used to control **strategic reduction**.

Semantics

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:

$$\langle \mathcal{A}, \mathcal{R}, s', t' \rangle$$

where \mathcal{A} and \mathcal{R} denote control stacks, and s' and t' respectively denote the strategy and term that result from the application $s \cdot \langle t \rangle$.

Basic Definitions in this Framework

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.

$$\frac{\text{applies}(r, t) \quad t \xrightarrow{r} t'}{r \cdot \langle t \rangle \Downarrow \langle \text{true}.\perp_{\mathcal{A}}, \text{true}.\perp_{\mathcal{R}}, r, t' \rangle} \text{E-successful}$$

$$\frac{\neg \text{applies}(r, t)}{r \cdot \langle t \rangle \Downarrow \langle \perp_{\mathcal{A}}, \perp_{\mathcal{R}}, r, t \rangle} \text{E-unsuccessful}$$

$$\frac{}{\text{SKIP} \cdot \langle t \rangle \Downarrow \langle \perp_{\mathcal{A}}, \perp_{\mathcal{R}}, \text{SKIP}, t \rangle} \text{E-skip}$$

Two standard combinators: $\langle + \rangle$ and $\langle ; \rangle$;

$$\frac{s_1 \cdot \langle t \rangle \Downarrow \langle \text{true}.\mathcal{A}, \mathcal{R}, s'_1, t' \rangle}{(s_1 \langle + \rangle s_2) \cdot \langle t \rangle \Downarrow \langle \text{true}.\mathcal{A}, \mathcal{R}, s'_1 \langle + \rangle s_2, t' \rangle} \text{E-choice1}$$

$$\frac{s_1 \cdot \langle t \rangle \Downarrow \langle \perp_{\mathcal{A}}, \mathcal{R}_1, s'_1, t' \rangle \quad s_2 \cdot \langle t' \rangle \Downarrow \langle \mathcal{A}_2, \mathcal{R}_2, s'_2, t'' \rangle}{(s_1 \langle + \rangle s_2) \cdot \langle t \rangle \Downarrow \langle \perp_{\mathcal{A}} \vee \mathcal{A}_2, \mathcal{R}_1 \vee \mathcal{R}_2, s'_1 \langle + \rangle s'_2, t'' \rangle} \text{E-choice2}$$

$$\frac{s_1 \cdot \langle t \rangle \Downarrow \langle \mathcal{A}_1, \mathcal{R}_1, s'_1, t' \rangle \quad s_2 \cdot \langle t' \rangle \Downarrow \langle \mathcal{A}_2, \mathcal{R}_2, s'_2, t'' \rangle}{(s_1 \langle ; \rangle s_2) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}_1 \vee \mathcal{A}_2, \mathcal{R}_1 \vee \mathcal{R}_2, s'_1 \langle ; \rangle s'_2, t'' \rangle} \text{E-seq}$$

2.1 Lemma: $\text{false}.\mathcal{A} \Rightarrow \mathcal{A} \equiv \perp$

The “visibility” combinators: *hide* and *lift*

$$\frac{s \cdot \langle t \rangle \Downarrow \langle x.\mathcal{A}, \mathcal{R}, s', t' \rangle}{\mathit{hide}(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, \mathit{hide}(s'), t' \rangle} \text{E-hide}$$

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

$$\Gamma \vdash \mathit{hide}(s_1) <+ s_2 \equiv s_1 <; s_2$$

$$\Gamma \vdash \mathit{hide}(\mathit{lift}(s_1)) <+ s_2 \equiv s_1$$

The combinators: *transient*, *opaque*, and *raise*

$$\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \text{true}.\mathcal{R}, s', t' \rangle}{\text{transient}(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, \text{SKIP}, t' \rangle} \text{E-transient1}$$

$$\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \perp\mathcal{R}, s', t' \rangle}{\text{transient}(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \perp\mathcal{R}, \text{transient}(s'), t' \rangle} \text{E-transient2}$$

$$\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, y.\mathcal{R}, s', t' \rangle}{\text{opaque}(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, \text{opaque}(s'), t' \rangle} \text{E-opaque}$$

$$\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, y.\mathcal{R}, s', t' \rangle}{\text{raise}(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, y.y.\mathcal{R}, \text{raise}(s'), t' \rangle} \text{E-raise}$$

Iterators: $\Phi = t_1.t_2.t_3 \dots$

$$\frac{}{s \cdot \langle end \rangle \Downarrow \langle \perp_{\mathcal{A}}, \perp_{\mathcal{R}}, s, end \rangle} \text{E-iterator1}$$

$$\frac{s \cdot \langle t_i \rangle \Downarrow \langle \mathcal{A}_1, \mathcal{R}_1, s', t'_i \rangle \quad s' \cdot \Phi_{i+1} \Downarrow \langle \mathcal{A}_2, \mathcal{R}_2, s'', \Phi'_{i+1} \rangle}{s \cdot \langle t_i.\Phi_{i+1} \rangle \Downarrow \langle \mathcal{A}_1 \vee \mathcal{A}_2, \mathcal{R}_1 \vee \mathcal{R}_2, s'', t'_i.\Phi'_{i+1} \rangle} \text{E-iterator2}$$

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

$$BUL\{\text{property}\} \triangleleft + \text{unfold}$$

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.

let val id = 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:
 - ▶ $B \stackrel{\pm}{\Rightarrow} \alpha$, and
 - ▶ α' is obtained from α by subscripting all nonterminals occurring in α .
- ▶ Subscripted nonterminals play an important role because they are treated as variables from the perspective of matching.

A Quick Overview of TL - Example

eval_list	::=	(dec [";"] expr ";") eval_list ϵ
dec	::=	"val" id "=" expr ...
expr	::=	id let_block ...
let_block	::=	"let" dec "in" expr "end"
id	::=	<i>identifier</i>

expr \llbracket let val *id*₁ = *expr*₁ in *expr*₂ end \rrbracket

A Quick Overview of TL - *conditional rewrite rules*

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

$$lhs \rightarrow rhs \ [\textit{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.

A Quick Overview of TL - *match expressions*

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 σ can be constructed so that $\sigma(t_1) = t_2$.

The substitution σ is a mapping from subscripted nonterminals to ground terms.

A Quick Overview of TL - Example

eval_list	::=	(dec [";"] expr ";") eval_list ϵ
dec	::=	"val" id "=" expr ...
expr	::=	id let_block ...
let_block	::=	"let" dec "in" expr "end"
id	::=	<i>identifier</i>

$\text{expr} \llbracket \text{let val } id_1 = \text{expr}_1 \text{ in } \text{expr}_2 \text{ end} \rrbracket$
 \ll
 $\text{expr} \llbracket \text{let val } x = 1 \text{ in } x + 5 \text{ end} \rrbracket$

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

Unfolding

$$\begin{aligned} & \text{expr}[\text{let val } id_1 = \text{expr}_1 \text{ in } \text{expr}_2 \text{ end}] \\ & \rightarrow \\ & \text{BUL}\{\text{expr}[id_1] \rightarrow \text{expr}[(\text{expr}_1)]\}(\text{expr}_2) \end{aligned}$$

- ▶ 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 the following strategy:

$$BUL\{s\} \leftarrow \text{unfold}$$

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

A Sketch: $BUL\{s\} \leftarrow + \text{unfold}$

```
BUL{  
  hide(  
    transient(expr[id1] → expr[id1])  
    <+  
    lift(expr[id1] → expr[id1])  
  )  
}
```

$(\text{expr}[\text{let val } id_1 = \text{expr}_1 \text{ in } \text{expr}_2 \text{ end}])$

transient lift

↓ ↓

$\Phi = t_1, t_2, \dots, t_i, \dots, t_j, \dots$ where $i < j$

optimize_let_blocks: $BUL\{simplify_let <; cleanup\}$

simplify_let: $expr_0 \rightarrow (BUL\{check[id_1]\} <+ unfold)(expr_0)$
if $expr_0 \gg expr\llbracket let\ val\ id_1 = expr_1\ in\ expr_2\ end\rrbracket$

identity: $id_1 \rightarrow expr\llbracket id_1\rrbracket \rightarrow expr\llbracket id_1\rrbracket$

check: $id_1 \rightarrow hide(transient(identity[id_1]) <+ lift(identity[id_1]))$

unfold: $expr\llbracket let\ val\ id_1 = expr_1\ in\ expr_2\ end\rrbracket$
 \rightarrow
 $BUL\{expr\llbracket id_1\rrbracket \rightarrow expr\llbracket (expr_1)\rrbracket\}(expr_2)$

cleanup: ...

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 ρ -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.