

# Stack-based Strategic Control

Victor L. Winter<sup>1</sup>

*Department of Computer Science  
University of Nebraska at Omaha  
USA*

---

## Abstract

In a strategic framework, combinators provide a fundamental mechanism for exercising control over rewriting. This type of control is based on the observation of the success or failure of strategy application. This paper describes a framework where information relating to the outcome of strategy application is stored in two internally maintained stacks. These stacks represent an implicit state which is used to control the rewriting process.

*Key words:* program transformation, strategic programming, TL, HATS

---

## 1 Overview

In a strategic framework, combinators provide a fundamental mechanism for exercising control over rewriting. Such control is based on *observing* the outcome of strategy application (i.e., its success or failure). In this paper we describe how rewriting can be controlled in the strategic language TL [11][9]. TL distinguishes itself from standard strategic languages in four important ways: First, the behavior of unsuccessful strategy application (i.e., application failure) is an identity on terms. That is, failure is not communicated at the term level. As a result, we say that TL is an *identity-based* system. Second, TL provides a unique set of combinators for controlling strategy application. Third, in TL it is possible for strategies themselves to change during application through a mechanism known as *strategic reduction*. Thus, the result of applying a strategy  $s$  to a term  $t$  is a tuple  $(s', t')$  denoting the resultant strategy and term. Fourth, TL is higher-order in the sense that it is possible to dynamically create strategies which, in turn, can then be applied to terms.

---

<sup>1</sup> Email: vwinter@mail.unomaha.edu

## 1.1 Contribution

There are two primary contributions made in this paper: First, a novel framework is developed where the operation of strategy application, when viewed abstractly from the perspective of being either successful or unsuccessful, is stored in a structure called a *control stack*. Second, it is shown how control stacks can be used to give a formal big-step style semantics to a core set of TL combinators. Included in the core are the combinators

$$\{ \textit{transient}, \textit{opaque}, \textit{raise}, \textit{hide}, \textit{lift} \}$$

which are unique to TL. Though informally mentioned in other articles, this article is the first place where the semantics of  $\{ \textit{opaque}, \textit{raise}, \textit{hide}, \textit{lift} \}$  are formally defined. Two previously unpublished examples are then presented showing how these non-standard combinators can be used to realize practical transformation objectives.

The rest of this paper is organized as follows: Section 2 gives a brief overview of the basic rewriting framework of TL followed by a big-step style semantics for a set of core TL combinators. Section 3 gives two examples of how the combinators of TL can be used to realize solutions to instances of a class of problems having practical application. Section 4 discusses related work and Section 5 concludes.

## 2 The Semantics of TL

TL is a language that has been developed exclusively for describing transformation-based computation [11,8]. The principle artifacts manipulated during the execution of a TL program are *parse trees*, which we 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 (e.g.,  $B_1$ ) or an expression of the form  $B[\alpha']$ . An expression  $B[\alpha']$  is well-formed if and only if the derivation  $B \xrightarrow{\pm} \alpha$  is possible according to the underlying grammar and  $\alpha'$  is obtained from  $\alpha$  by subscripting all nonterminals occurring in  $\alpha$ .

Transformation is accomplished through the application of rewrite rules to terms. In TL, a first-order rewrite rule has the following syntactic structure:

$$lhs \rightarrow rhs [ \textit{if condition} ]$$

where [ and ] are syntactic meta symbols indicating that the enclosed section (i.e., the conditional portion) of a rule is optional. In order for a first-order rule to be well-formed it is necessary that  $lhs$  be a *pattern*, that  $rhs$  be a *strategic expression* (i.e., an expression whose evaluation yields a term), and that *condition* be an expression consisting of one or more *match expressions* combined using the Boolean connectives: *and*, *or*, *not*.

A *match expression* is a first-order match between two patterns. 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  $\sigma(t_1) = t_2$ .

A basic conditionless  $n$ -order rule has the form:

$$lhs_1 \rightarrow lhs_2 \rightarrow \dots \rightarrow lhs_n \rightarrow rhs$$

The current implementation of TL makes a syntactic distinction between first-order strategy application and higher-order strategy application. Let  $s^1$  and  $s^2$  respectively denote a first and second-order strategy and let  $t$  denote a term. In TL, the first-order application is written  $s^1(t)$  and the second-order application is written  $s^2[t]$ .

TL provides a rich framework for defining traversals. TL also provides a predefined library of traversals that includes both first-order as well as higher-order generic traversals. The TL traversal library contains common first-order generic traversals such as bottom-up left-to-right (BUL), top-down left-to-right (TDL) and so on. The expression  $BUL\{s\}$  denotes a strategy that when applied to a term  $t$  will traverse  $t$  in a bottom-up left-to-right fashion and apply the first-order strategy  $s$  to every term encountered.

TL also provides a predefined top-down (TD) traversal in addition to the top-down left-to-right (TDL) traversal. The traversal TD is distinctly different from TDL. In particular, the expression  $TD\{s\}$  denotes a strategy that when applied to a term  $t$  will traverse  $t$  in a top-down fashion with the following behavior: Let  $s'$  denote the strategy that results from applying  $s$  to  $t$ . Each direct sub-term of  $t$  will be given its own copy of  $s'$  at which point the TD traversal continues. See [8] for a more detailed discussion of TD.

### 2.1 The Semantics of the TL Core Combinator Set

There are two fundamental combinator types in TL: those that control *strategy application*, and those that control *strategic reduction*. Examples of the former include  $\{<+, <;, hide, lift\}$ . Examples of the latter include the unary combinators  $\{transient, opaque, raise\}$ .

In TL, the behavior of all combinators can be formally described using two internally maintained stacks:  $\mathcal{A}$  and  $\mathcal{R}$ . *Strategy application* is controlled by the stack  $\mathcal{A}$ . *Strategic reduction* is controlled by the stack  $\mathcal{R}$ .

In order to abstract away some low-level operational details from our semantics, a stack is modelled here as an infinite structure whose initial entries are all set to the Boolean value *false*. Initial stacks corresponding to  $\mathcal{A}$  and  $\mathcal{R}$  are respectively denoted  $\perp_{\mathcal{A}}$  and  $\perp_{\mathcal{R}}$ . An element can be pushed onto the top of a stack using a “dot” constructor. For example, let  $\mathcal{A}$  denote a stack and let  $x$  denote a Boolean value, then  $x.\mathcal{A}$  denotes the stack  $\mathcal{A}$  with the value  $x$  pushed on to it. Elements can be extracted from the top of a stack using pattern matching. For example, in the pattern  $x.\mathcal{A}$ , the symbol  $x$  denotes the top of the stack and  $\mathcal{A}$  denotes the rest of the stack. Similarly, in the pattern  $x_1.x_2.\mathcal{R}$  the symbol  $x_1$  denotes the top of the stack, the symbol  $x_2$  denotes

the second symbol on the stack and  $\mathcal{R}$  denotes the rest of the stack.

Lastly, we extend Boolean operations (e.g.,  $\vee$ ) to stacks. For example, let  $\mathcal{A}_1$  and  $\mathcal{A}_2$  denote two (infinite) stacks. The Boolean expression  $\mathcal{A}_1 \vee \mathcal{A}_2$  denotes the stack whose elements are obtained from the element-wise disjunction of  $\mathcal{A}_1$  and  $\mathcal{A}_2$ .

## 2.2 Basic Strategy Application

Figure 1 gives a big-step style semantics describing the basic forms of strategy application. In syntax used, the application of a strategy  $s$  to a term  $t$  is denoted  $s \cdot \langle t \rangle$ . The rules *E-success* and *E-failure* in Figure 1 make use of a predicate *applies* which we do not formally define here. Informally, the predicate *applies*( $r, t$ ) evaluates to *true* if the application of the rewrite rule  $r$  to the term  $t$  is successful; otherwise it evaluates to *false*. Recall, that TL models application failure as an identity on terms. Thus, the term-level value *FAIL* is not used to communicate application failure. In the interests of space and to sidestep a digression tangential to the focus of this paper, we assume that the semantics of the predicate *applies*( $r, t$ ) is understood.

$$\begin{array}{c}
 \frac{\textit{applies}(r, t) \quad t \xrightarrow{r} t'}{r \cdot \langle t \rangle \Downarrow \langle \textit{true}.\perp_{\mathcal{A}}, \textit{true}.\perp_{\mathcal{R}}, r, t' \rangle} \text{E-success} \\
 \\
 \frac{\neg \textit{applies}(r, t)}{r \cdot \langle t \rangle \Downarrow \langle \perp_{\mathcal{A}}, \perp_{\mathcal{R}}, r, t \rangle} \text{E-failure} \\
 \\
 \frac{}{SKIP \cdot \langle t \rangle \Downarrow \langle \perp_{\mathcal{A}}, \perp_{\mathcal{R}}, SKIP, t \rangle} \text{E-skip} \\
 \\
 \frac{}{ID \cdot \langle t \rangle \Downarrow \langle \textit{true}.\perp_{\mathcal{A}}, \textit{true}.\perp_{\mathcal{R}}, ID, t \rangle} \text{E-id}
 \end{array}$$

Fig. 1. Atomic rule application and the strategic constants *SKIP* and *ID*

The result of evaluating an application of a strategy  $s$  to a term  $t$  is a tuple of the form  $\langle \mathcal{A}, \mathcal{R}, s', t' \rangle$  where (1)  $\mathcal{A}$  and  $\mathcal{R}$  respectively denote the control stack restricting strategic application and the control stack enabling strategic reduction, and (2)  $s'$  and  $t'$  respectively denote the strategy and term resulting from the application. One thing to notice is that the result of applying the strategic constant *SKIP* to a term  $t$  produces the same result as application failure.

## 2.3 The Strategic Reduction Stack $\mathcal{R}$

Figure 2 gives the semantics for the combinators  $\{\textit{transient}, \textit{opaque}, \textit{raise}\}$ . Informally stated, a strategy *transient*( $s$ ) will reduce to the strategic constant

$$\begin{array}{c}
\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, true.\mathcal{R}, s', t' \rangle}{transient(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, SKIP, t' \rangle} \text{E-transient1} \\
\\
\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \perp_{\mathcal{R}}, s', t' \rangle}{transient(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, transient(s'), t' \rangle} \text{E-transient2} \\
\\
\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, y.\mathcal{R}, s', t' \rangle}{opaque(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, opaque(s'), t' \rangle} \text{E-opaque} \\
\\
\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, y.\mathcal{R}, s', t' \rangle}{raise(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, y.y.\mathcal{R}, raise(s'), t' \rangle} \text{E-raise}
\end{array}$$

Fig. 2. The combinators effecting the control stack  $\mathcal{R}$

*SKIP* after application if it can be observed (via the topmost element of the stack  $\mathcal{R}$ ) that the application of the strategy  $s$  to a term  $t$  has been successful. Note that in this case, the top of the stack  $\mathcal{R}$  is popped. As a result, the reduction caused by a transient is not cascading. To see what we mean by this consider the application of the strategy  $transient(transient(s_1) <+ s_2)$  to a term  $t$  under an assumption  $\Gamma$  that the application of  $s_1$  is successful. Furthermore  $\Gamma$  assumes that after the application of  $s_1$ , the strategic reduction stack has the configuration  $true.\perp_{\mathcal{R}}$ . In this case, the assumption  $\Gamma$  would entail the strategic reduction shown in Equation 1.

$$\Gamma \vdash transient(transient(s_1) <+ s_2) \xrightarrow{reduce} transient(SKIP <+ s_2) \quad (1)$$

The *opaque* combinator restricts (in a limited fashion) the ability of the *transient* combinator to observe that its contents has been successfully applied. This is accomplished by popping the stack  $\mathcal{R}$ . Consider the application of the strategy  $transient(opaque(s_1) <+ s_2)$  to a term  $t$  under an assumption  $\Gamma$  that the application of  $s_1$  is successful. Furthermore,  $\Gamma$  assumes that after the application of  $s_1$ , the strategic reduction stack has the configuration  $true.\perp_{\mathcal{R}}$  and the strategic value resulting from the application is  $s'_1$ . In this case, the assumption  $\Gamma$  would entail the strategic (identity) reduction shown in Equation 2:

$$\Gamma \vdash transient(opaque(s_1) <+ s_2) \xrightarrow{reduce} transient(opaque(s'_1) <+ s_2) \quad (2)$$

In contrast, the *raise* combinator enables strategic reduction to propagate up one level (i.e., to the second enclosing *transient* if it exists). This is accomplished by duplicating the top element of the stack  $\mathcal{R}$ . Consider the application of the strategy  $transient(transient(raise(s_1)) <+ s_2)$  to a term  $t$

under an assumption  $\Gamma$  that the application of  $s_1$  is successful. Furthermore,  $\Gamma$  assumes that after the application of  $s_1$ , the strategic reduction stack has the configuration  $true.\perp_{\mathcal{R}}$ . In this case, the assumption  $\Gamma$  would entail the strategic reduction shown in Equation 3.

$$\Gamma \vdash \text{transient}(\text{transient}(\text{raise}(s_1)) <+ s_2) \xrightarrow{\text{reduce}} \text{SKIP} \quad (3)$$

#### 2.4 The Strategy Application Stack $\mathcal{A}$

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

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

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

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

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

Fig. 3. The combinators effecting the stack  $\mathcal{A}$

Figure 3 gives the semantics of the combinators  $\{<+, <;, \text{hide}, \text{lift}\}$ . Notice that in the rule *E-choice1*, the observation of the successful application of  $s_1$  propagates (i.e.,  $\text{true}.\mathcal{A}$ ). More importantly notice that in the rule *E-choice2*, the value  $(s'_1, t')$  propagates! This is not as disruptive to the overall semantics as first meets the eye. Furthermore, it is precisely these kinds of propagations that gives this identity-based strategic system its power.

The rule *E-seq* gives the semantics of left-to-right sequential composition. Notice that in this case the stack information in the resultant tuple corresponds to the disjunction of the stack information obtained from the application of  $s_1$  and the application of  $s_2$ . We would like to point out that in a failure-based strategic framework a conjunction of these two stacks would be more appropriate. TL has a combinator ( $<*$ ) that has precisely such a semantics, but its description has been omitted from this discussion.

The semantics of the *hide* combinator is given by the rule *E-hide*. Infor-

mally speaking, *hide* restricts the ability of a combinator like  $<+$  to observe whether a strategy has been successfully applied. This is accomplished by popping the stack  $\mathcal{A}$ . Consider the application of the strategy  $hide(s_1) <+ s_2$  to a term  $t$  under an assumption  $\Gamma$  that application of  $s_1$  is successful. Furthermore,  $\Gamma$  assumes that, after the application of  $s_1$ , the strategy application stack has the configuration  $true.\perp_{\mathcal{A}}$ . In this case, the assumption  $\Gamma$  would entail the behavioral equivalence (from the perspective of strategic application) shown in Equation 4.

$$\Gamma \vdash hide(s_1) <+ s_2 \equiv s_1 <; s_2 \quad (4)$$

The equivalence shown in Equation 4 generally raises an initial reaction prompting several questions: (1) Does *hide* do anything new? The answer is yes. (2) Is the *hide* combinator simply a derived form? The answer is no. We kindly ask the reader to suspend judgement on these issues until the end of Section 3. (See [9][10] for an additional example of a nontrivial use of *hide*.)

The *lift* combinator is given by the rule *E-lift*. The *lift* combinator enables the observation of successful application to propagate past an enclosing *hide*. This is accomplished by duplicating the top element of the stack  $\mathcal{A}$ . Consider the application of the strategy  $hide(lift(s_1)) <+ s_2$  under an assumption  $\Gamma$  where the application of  $s_1$  is successful. Furthermore,  $\Gamma$  assumes that, after the application of  $s_1$ , the strategy application stack has the form  $true.\perp_{\mathcal{A}}$ . In this case, the assumption  $\Gamma$  would entail the behavioral equivalence shown in Equation 5.

$$\Gamma \vdash hide(lift(s_1)) <+ s_2 \equiv s_1 \quad (5)$$

## 2.5 Iterators

The strategic system presented so far defines the semantics of *strategy application* to a single term:  $s \cdot \langle t \rangle$ . Iterators can be introduced into this framework to extend the notion of strategy application to term sequences.

In this paper, we are interested in iterators only from the perspective of their interaction with control stacks. This focus suggests viewing an iterator abstractly (in a non-constructive fashion) as a stream-like entity from which terms can be extracted in an incremental fashion. We will use the symbol  $\Phi$ , in subscripted form, to abstractly denote the term stream produced by an iterator.

Term streams can be deconstructed using standard pattern matching. We write  $\Phi = t_i.\Phi_{i+1}$  to denote a match expression that succeeds if the stream  $\Phi_i$  can be deconstructed into the (next) term  $t_i$  and the rest of the term sequence  $\Phi_{i+1}$ ; otherwise the match fails. Finite term streams are terminated by the special symbol *end*. For example, a term stream consisting of a single term has the form:  $t_1.end$ .

Figure 4 gives the semantics of strategy application over term sequences.

$$\begin{array}{c}
\frac{}{s \cdot \langle end \rangle \Downarrow \langle \perp_{\mathcal{A}}, \perp_{\mathcal{R}}, s, end \rangle} \text{E-iterator1} \\
\frac{s \cdot \langle t_i \rangle \Downarrow \langle \mathcal{A}_1, \mathcal{R}_1, s', t'_i \rangle \quad s' \cdot \Phi_{i+1} \Downarrow \langle \mathcal{A}_2, \mathcal{R}_2, s'', \Phi'_{i+1} \rangle}{s \cdot \langle t_i \cdot \Phi_{i+1} \rangle \Downarrow \langle (\mathcal{A}_1 \vee \mathcal{A}_2), (\mathcal{R}_1 \vee \mathcal{R}_2), s'', t'_i \cdot \Phi'_{i+1} \rangle} \text{E-iterator2}
\end{array}$$

Fig. 4. The generalization of application to term sequences

It should be noted that in practice, the term sequences produced by iterators are typically intimately linked to some initial term or set of terms. For example, a bottom-up traversal will visit all sub-terms of the initial term to which it is applied. Because of the dependencies that typically exist between the elements in term sequences, it is helpful to view a term as a mutable value. This helps to understand the global change brought about by the application of an iterator to a term.

### 3 Applications

This section takes a look at two examples in which the interaction between various combinators can be used to achieve specific transformational objectives. Additional examples can be found in [11][8][10][9][7].

#### 3.1 Let-Block Optimization

This example shows how the combinators *hide*, *transient*, and *lift* can be used together with generic traversal to optimize ML-style let-blocks. An important capability highlighted by this example is that *the notion of conditional control is extended over the domain of generic traversals*. Conceptually speaking, the heart of this example is a strategy of the form:  $BUL\{s_1\} <+ s_2$ . In some circumstances, the application of the traversal  $BUL\{s_1\}$  is seen to succeed preventing the application of  $s_2$ . In other circumstances the application of  $BUL\{s_1\}$  is seen to fail thereby allowing the application of  $s_2$ .

The goal of let-block optimization, as described here, is to 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. The strategy described in this example assumes that a let-block is an expression having the following form:

$$let\ val\ id = expr\ in\ expr\ end \tag{6}$$

The strategy also assumes that all identifiers declared in a let-block are unique. Transformations establishing these two assumptions for arbitrary let-blocks are straightforward and well known. Therefore, these assumptions do not represent a loss-of generality. A grammar fragment formalizing the structures of interest, with respect to this example, is given in Figure 5. A strategic

program, written in TL, realizing let-block optimization is shown in Figure 6, and a concrete example showing the results of let-block optimization is given in Figure 7.

<code>eval_list</code>	<code>::= (dec [ “;” ]   expr “;”) eval_list   <math>\epsilon</math></code>
<code>dec</code>	<code>::= “val” id “=” expr   ...</code>
<code>expr</code>	<code>::= id   let_block   ...</code>
<code>let_block</code>	<code>::= “let” dec “in” expr “end”</code>
<code>id</code>	<code>::= <i>identifier</i></code>

Fig. 5. An extended-BNF grammar fragment describing a subset of ML

<b>optimize_let_blocks:</b>	$BUL\{simplify\_let <; cleanup\}$
<b>simplify_let:</b>	$expr_0 \rightarrow (BUL\{check[id_1]\} <+ unfold)(expr_0)$ if $expr_0 \gg expr\llbracket let\ val\ id_1 = expr_1\ in\ expr_2\ end\rrbracket$
<b>identity:</b>	$id_1 \rightarrow expr\llbracket id_1\rrbracket \rightarrow expr\llbracket id_1\rrbracket$
<b>check:</b>	$id_1 \rightarrow hide(transient(identity[id_1]) <+ lift(identity[id_1]))$
<b>unfold:</b>	$expr\llbracket let\ val\ id_1 = expr_1\ in\ expr_2\ end\rrbracket$ $\rightarrow$ $BUL\{expr\llbracket id_1\rrbracket \rightarrow expr\llbracket (expr_1)\rrbracket\}(expr_2)$
<b>cleanup:</b>	...

Fig. 6. A strategic program for optimizing let-blocks

In this example, the strategy *optimize\_let\_blocks* performs overall let-block optimization. The strategy *optimize\_let\_blocks* traverses the term to which it is applied in a bottom-up left-to-right (BUL) fashion and applies the strategy *simplify\_let <; cleanup* to each term encountered during the traversal. The strategy *simplify\_let <; cleanup* consists of the sequential (left-to-right) composition (<;) of the two strategies *simplify\_let* and *cleanup*.

The strategy *simplify\_let* performs the actual optimization of an individual let-block and the strategy *cleanup* removes unnecessary top-level parenthesis that may be introduced during the optimization process. From an operational perspective, the strategy *simplify\_let* is a conditional strategy whose condition assures that *simplify\_let* will be applied to an expression  $expr_0$  only if  $expr_0$  is a let-block<sup>2</sup>. When a let-block is encountered, an instance of the strategy  $BUL\{check[id_1]\} <+ unfold$  is dynamically created (for a specific value

<sup>2</sup> The conditional portion of this strategy is added only for readability. Technically speak-

of  $id_1$ ) and this strategy is then applied to the let-block (i.e., to  $expr_0$ ). The strategy  $BUL\{check[id_1]\} <+ unfold$  consists of the left-to-right conditional composition ( $<+$ ) of the strategy  $BUL\{check[id_1]\}$  and  $unfold$ .

When applied to an expression that is a let-block, the strategy  $unfold$  will remove the let-block and return an expression corresponding to an instance of the body of the let-block in which the expression ( $expr_1$ ) bound to the declared variable ( $id_1$ ) is substituted for all occurrences of the declared variable. This substitution is accomplished by traversing the body of the let-block ( $expr_2$ ) and suitably rewriting all occurrences of the declared variable.

The strategy  $BUL\{check[id_1]\}$  serves as a filter that, due to its conditional composition with  $unfold$ , only passes control to  $unfold$  (i.e., only permits the application of  $unfold$ ) under suitable circumstances. Specifically, the application of  $unfold$  is permitted only to let-blocks whose bodies contain no more than one occurrence of their declared variable.

When applied to a specific identifier  $id_1$ , the higher-order strategy  $check$  will return the following strategy:

$$hide(transient(identity[id_1]) <+ lift(identity[id_1])) \quad (7)$$

Here, the strategic expression  $identity[id_1]$  has been added for readability and evaluates to the first-order identity strategy<sup>3</sup> on the expression  $id_1$ .

When viewed externally (e.g., from the context of the enclosing BUL traversal), the strategy  $hide(transient(identity[id_1]) <+ lift(identity[id_1]))$  is seen to successfully apply if and only if the strategy  $lift(identity[id_1])$  is successfully applied. In this case, since the  $transient$  and  $lift$  combinators are applied to the same strategy, it follows that the strategy enclosed by  $lift$  can only be applied after the strategy enclosed by  $transient$  has been applied<sup>4</sup>. Furthermore, since these strategies are conditionally composed, we can infer that the term to which the  $lift$  strategy is applied must have a position that is different, in the term sequence resulting from the BUL traversal, than the position of term to which the  $transient$  strategy is applied. Specifically, this implies that there are two or more occurrences of terms matching  $expr[id_1]$ . When this arises, control is not passed to the  $unfold$  strategy; otherwise control is passed to the  $unfold$  strategy.

Figure 7 is a concrete example showing let-block optimization. Note that this example contains instances of the three cases that must be accounted for by this optimization. Case 1 arises when there are no occurrences of the declared variable in the body of the let-block. Case 2 arises when there is exactly one occurrence of the declared variable in the body of the let-block. Case 3 arises when there are two (or more) occurrences of the declared variable

---

ing, the pattern  $expr[let val id_1 = expr_1 in expr_2 end]$  could have been in-lined for all occurrences of  $expr_0$  in which case the conditional portion of the strategy could be removed.

<sup>3</sup> It should be noted that the pattern  $expr[id_1]$  will not match with left-hand side of a val declaration, which is precisely what is needed.

<sup>4</sup> Recall that a transient enclosed strategy reduces to skip after its successful application.

in the body of the let-block.

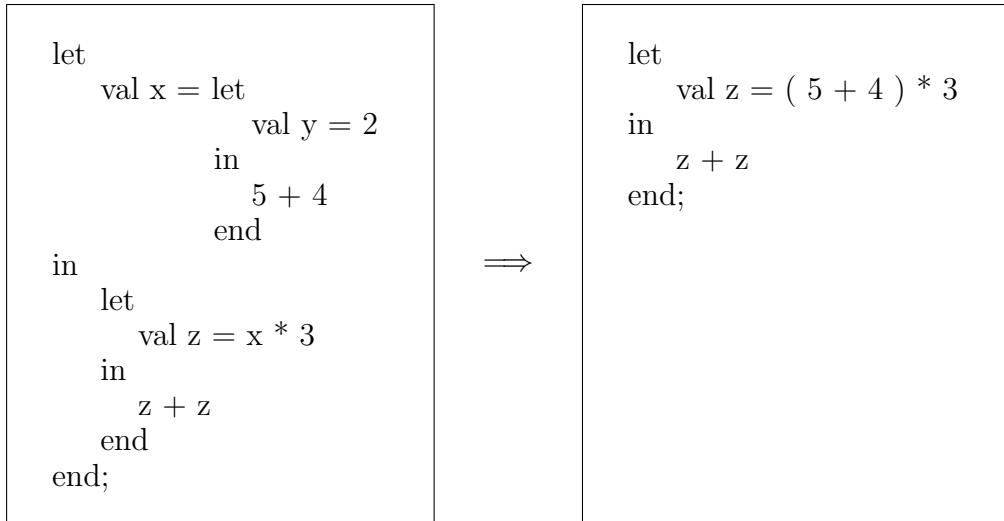


Fig. 7. A concrete example of let-block optimization

### 3.2 Contextual Renaming

In this example, we take a look at how the combinators *transient*, *raise*, and *opaque* can be combined to control renaming within a block-structured class hierarchy. The goal here is to implement a renaming policy that only renames variables within classes having specific structural properties. We consider two policies: (1) rename all occurrences of a specific variable within a particular (sub)class, and (2) rename all occurrences of a specific variable in all classes that are subclasses of a given class.

It should be noted that the kind of renaming policies considered here can be adapted to a variety of weaving tasks that often arise within an aspect-oriented framework. For example, the transformation in this example can be (and has been) adapted to implement the semantics of a variety of pointcut expressions that describe static contexts.

It is important to note that the renaming contexts in this example are “hardwired” into the strategies themselves. This is done to allow the discussion to focus on control issues. Since TL supports higher-order strategies, only a minor modification is needed to remove such hardwiring and instead parameterize renaming contexts with respect to expressions arising within the source program itself (e.g., pointcuts associated with advice).

A grammar fragment formalizing the structures of interest, with respect to this example, is given in Figure 8. A strategic program, written in TL, realizing contextual renaming is shown in Figure 9, and a concrete example showing the results of contextual renaming is given in Figure 10.

In this example, the strategy *contextual\_rename* performs the overall renaming. This is accomplished by applying, to the input term, the strategy

<code>class_list</code>	<code>::=</code>	<code>class_def class_list   <math>\epsilon</math></code>
<code>class_def</code>	<code>::=</code>	<code>“class” id “{” dec_list “}”</code>
<code>dec_list</code>	<code>::=</code>	<code>dec [ “;” ] dec_list   <math>\epsilon</math></code>
<code>dec</code>	<code>::=</code>	<code>class_def   field_def   ...</code>
<code>field_def</code>	<code>::=</code>	<code>type id [ “=” expr ]</code>
<code>expr</code>	<code>::=</code>	<code>id   ...</code>

Fig. 8. An extended-BNF grammar fragment for contextual renaming

<b>contextual_rename:</b>	$TD\{rename\_henceforth<;rename\_here\}$
<hr/>	
<b>r1:</b>	$expr\llbracket henceforth \rrbracket \rightarrow expr\llbracket HENCEFORTH \rrbracket$
<b>rename_henceforth:</b>	$transient(filter[id\llbracket A \rrbracket] <+ filter[id\llbracket B \rrbracket] <+ r1)$
<b>r2:</b>	$expr\llbracket here \rrbracket \rightarrow expr\llbracket HERE \rrbracket$
<b>rename_here:</b>	$transient(filter[id\llbracket A \rrbracket] <+ filter[id\llbracket B \rrbracket] <+ bad\_context <+ r2)$
<hr/>	
<b>bad_context:</b>	$class\_def\llbracket class\ id_1\{dec\_list_1\} \rrbracket \rightarrow class\_def\llbracket class\ id_1\{dec\_list_1\} \rrbracket$
<b>filter:</b>	$id_1$ $\rightarrow$ $transient($ $\quad class\_def\llbracket class\ id_1\{dec\_list_1\} \rrbracket$ $\quad \rightarrow$ $\quad class\_def\llbracket class\ id_1\{dec\_list_1\} \rrbracket$ $\quad <+$ $\quad raise(bad\_context)$ $\quad <+$ $\quad opaque(ID)$ $)$

Fig. 9. A strategic program for contextual renaming

$rename\_henceforth <; rename\_here$  using the traversal  $TD$ . The strategy  $rename\_henceforth <; rename\_here$  is the sequential left-to-right composition ( $<;$ ) of the strategy  $rename\_henceforth$  and  $rename\_here$ .

The strategy  $rename\_henceforth$  implements a hardwired renaming policy in which the rewrite rule  $r1$  is used to rewrite occurrences of the term  $expr\llbracket henceforth \rrbracket$  to  $expr\llbracket HENCEFORTH \rrbracket$ . Control is only passed to  $r1$

provided the subject term occurs (1) in the context of a *class B* which is the (immediate) inner class of *class A*, or (2) in any inner class extending the context *class B* <: *class A*, where <: denotes the standard subtype relation; otherwise the application of *r1* is prevented.

Within the strategy *rename\_henceforth* control over the application of *r1* is exercised by the strategic expression  $filter[id[[A]]] <+ filter[id[[B]]]$ . When evaluated, *rename\_henceforth* has the form

$$transient(\varepsilon_1 <+ \varepsilon_2 <+ r1) \tag{8}$$

where  $\varepsilon_1$  has the form:

```

transient(
  (* — Case 1: Positive Match — *)
  class_def[[class A {dec_list1}]] → class_def[[class A {dec_list1}]]
  <+
  (* — Case 2: Negative Match — *)
  raise(bad_context)
  <+
  (* — Case 3: Short-circuit — *)
  opaque(ID)
)

```

It should be noted that the strategy  $\varepsilon_1$  is a transient enclosed strategy whose body, as the comments suggest, consists of three cases. **Case 1** arises when a term *t* corresponding to *class A* is encountered. When this occurs the rewrite associated with case 1 applies and, due to the fact that the body of  $\varepsilon_1$  is enclosed in a *transient*, the entire strategy  $\varepsilon_1$  is reduced to the strategy *SKIP*.

**Case 2** arises when a term *t* corresponding to a class other than *class A* is encountered (i.e., we are in the wrong context). When this occurs the strategy associated with case 2 applies. Since this strategy involves the combinator *raise*, its successful application to *t* will cause the contents of the two enclosing transients to reduce to *SKIP*. In other words, the strategy *rename\_henceforth* which has the form  $transient(\varepsilon_1 <+ \varepsilon_2 <+ r1)$  will be reduced to *SKIP*. Thus, no renaming of any sub-term of *t*, visited by the traversal *TD*, will take place.

**Case 3** applies to all terms *t* that do not correspond to a class definition (e.g., *expr*, *field\_def*, etc.). In this case, we would like our strategy  $\varepsilon_1$  to ignore such terms since they are of the wrong type. However, we also simultaneously want to prevent the application of rewrite *r1* with which  $\varepsilon_1 <+ \varepsilon_2$  is conditionally composed. To prevent an application of *r1*, the <+ combinator must perceive the application of  $\varepsilon_1$  as succeeding. This is accomplished by the strategic constant *ID*. In contrast, the *transient* combinator must perceive the application of  $\varepsilon_1$  as failing. This is accomplished by applying the *opaque*

combinator to the strategic constant  $ID$ .

The description of the strategy  $\varepsilon_2$  is similar to  $\varepsilon_1$ . The only difference is that  $\varepsilon_2$  involves the class id  $B$  whereas  $\varepsilon_1$  involves the class id  $A$ .

The behavior of the strategy  $rename\_here$  is similar to that of the strategy  $rename\_henceforth$ . The only difference is that, after evaluation, the strategy  $rename\_here$  will have the form:

$$transient(\varepsilon_1 <+ \varepsilon_2 <+ bad\_context <+ r1) \quad (9)$$

The addition of of the strategy  $bad\_context$  assures that renaming will only occur within a designated class. In particular, renaming will not continue on into the inner classes of the designated class.

Figure 10 is a concrete example showing the results of contextual renaming. Notice that there are two occurrences of *class B* but the rewrite  $here \rightarrow HERE$  only occurs in the *class B* which is an (immediate) inner class of *class A*. Furthermore, note that the strategy  $rename\_henceforth$  also avoids rewriting the occurrence of the identifier *henceforth* in the occurrence of *class B* which is an inner class of *class D*.

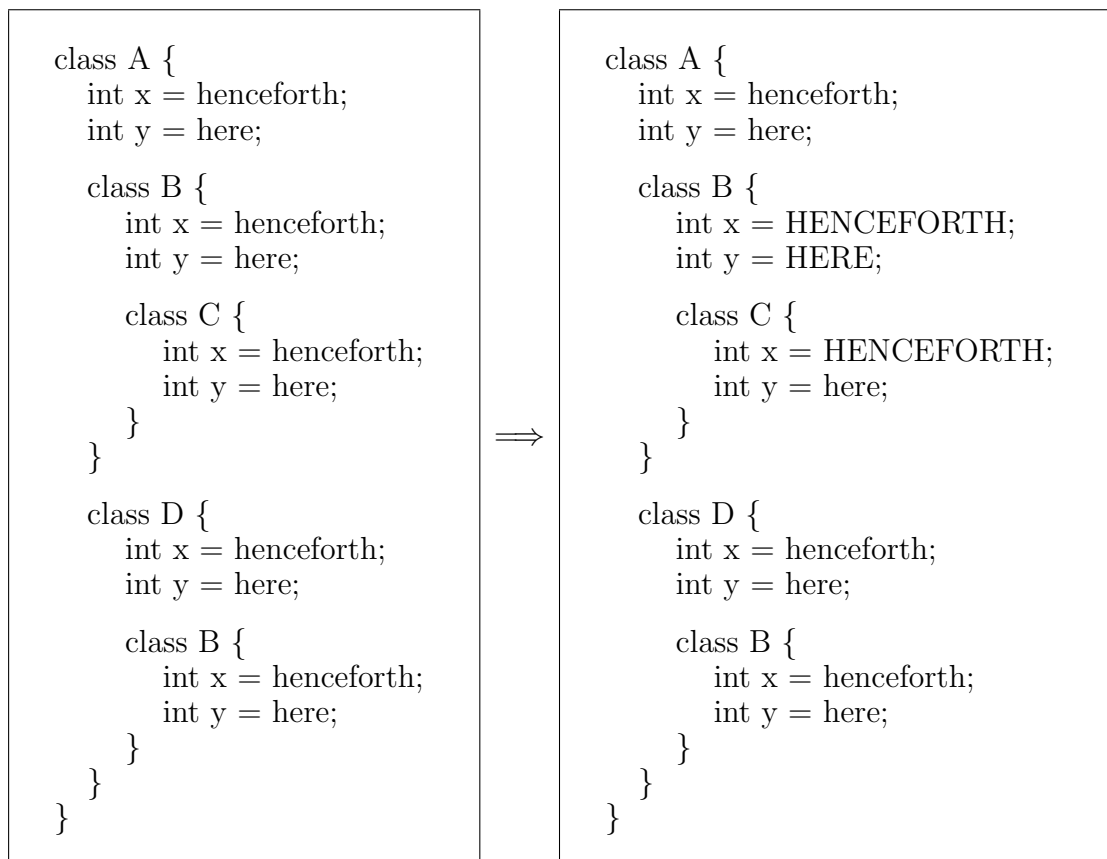


Fig. 10. A concrete example of contextual renaming

## 4 Related Work

In TL, the *raise/opaque* and *hide/lift* combinators provide an exception-like mechanism for communicating information pertaining to strategy application between combinators within a strategy. Similarly, virtually all programming languages offer some mechanisms to describe nonstandard control flows that can be used to escape from nested computations. Basic mechanisms found in early programming languages include *goto*, *break*, *continue*, and *return*. Modern programming languages support more sophisticated abstractions that enable nonstandard flow of control via the throwing and catching of exceptions. Scheme takes this idea further and offers exotic control abstractions such as *call-with-current-continuation* (call/cc) and *dynamic-wind*.

Stratego [6] is a first-order strategic programming language providing a rich set of primitives for controlling the application of rewrite rules. In Stratego, *dynamic rules* can be created at run-time [2]. Such rules inherit information from the context in which they are created and are analogous to the higher-order rules of TL. Stratego also provides a transient-like ability supporting “the application of dynamic rules only once” [2].

TOM [1] is a system that extends an imperative language (e.g., Java) with matching primitives. TOM supports a standard set of strategic constructs including *Choice*, *Sequence*, *All*, and *One*. In TOM, all strategies are seen as an extension of either the Identity strategy or the Fail strategy. In [1], it is shown how TOM’s strategic primitives together with a generalized recursion operator  $\mu$  can be used to express computational tree logic (CTL) formulae.

In [4], *conditional transformations* are defined within a logic-based framework. Transformations can be composed into OR-sequences as well as AND-sequences. The behavior of OR-sequences has similarities with the identity-based semantics of TL.

In [5], an extension to the first-order (identity-based) rewriting system ASF+SDF is described in which one can combine parameterized rewrite rules with a fixed set of generic traversals.

The  $\rho$ -calculus [3] provides a fully general (higher-order) framework in which strategies can be applied to other strategies and yield strategy sets as their results.

## 5 Conclusion

TL provides a rich environment in which the interplay between dynamic strategy creation and strategic reduction (via the *transient* combinator) are brought together in an identity-based framework. The identity-based nature of TL enables the notion of conditional application to be seamlessly extended over the domain of iterators. The more exotic combinators of TL (*hide*, *opaque*, etc.) greatly enhance the control that can be embedded within strategies. This kind of control is especially useful in the context of dynamic strategy

generation.

## References

- [1] E. Balland, P.-E. Moreau, and A. Reilles. Bytecode rewriting in Tom. In *Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 07)*, Braga/Portugal, 2007.
- [2] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69:1–56, 2005.
- [3] Cirstea, Horatiu and Kirchner, Claude. An introduction to the rewriting calculus. Research Report RR-3818, INRIA, Dec. 1999.
- [4] G. Kniesel. A Logic Foundation for Conditional Program Transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, January 2006. ISSN 0944-8535.
- [5] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.
- [6] E. Visser, Z. el Abidine Benaïssa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98: Proc. of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26. ACM Press, 1998.
- [7] V. Winter. Model-driven Transformation-based Generation of Java Stress Tests. *Electronic Notes in Theoretical Computer Science (ENTCS)*.
- [8] V. Winter. Strategy Construction in the Higher-Order Framework of TL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 124, 2004.
- [9] V. Winter and J. Beranek. Program Transformation Using HATS 1.84. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 378–396, 2006.
- [10] V. Winter, J. Beranek, F. Fraij, S. Roach, and G. Wickstrom. A Transformational Perspective into the Core of an Abstract Class Loader for the SSP. *ACM Trans. on Embedded Computing Sys.*, 5(4):0–0, 2007.
- [11] V. Winter and M. Subramaniam. The Transient Combinator, Higher-order Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.