Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 124 (2005) 149–170
www.elsevier.com/locate/entcs

Strategy Construction in the Higher-Order Framework of TL

Victor L. Winter [1,2]

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

Abstract

When viewed from a strategic perspective, a labeled rule base in a rewriting system can be seen as a restricted form of strategic expression (e.g., a collection of rules strictly composed using the left-biased choice combinator). This paper describes higher-order mechanisms capable of dynamically constructing strategic expressions that are similar to rule bases. One notable difference between these strategic expressions and rule bases is that strategic expressions can be constructed using arbitrary binary combinators (e.g., left-biased choice, right-biased choice, sequential composition, or user defined). Furthermore, the data used in these strategic expressions can be obtained through term traversals.
A higher-order strategic programming framework called TL is described. In TL it is possible to dynamically construct strategic expression of the kind mentioned in the previous paragraph. A demonstration follows showing how the higher-order constructs available in TL can be used to solve several problems common to the area of program transformation.

Keywords: Program transformation, rewriting, strategic programming, higher-order rewriting, transient combinator, TL

1 Introduction

The concept of distributing data within a term structure is central to rewrite-based computation [11]. In [14] this problem is characterized and referred to

[1] This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy. Victor Winter was also partially supported by NSF grant number CCR-0209187.
[2] Email: vwinter@mail.unomaha.edu

doi:10.1016/j.entcs.2004.07.020

as the *distributed data problem* (DDP). When the data to be distributed is independent of the input (i.e., constant for all input terms), simple strategies for distributing data can oftentimes be constructed statically. For example, consider constructing a strategy that will rewrite every integer in a term to the integer 2. Here the objective is to distribute the integer 2 throughout a term structure by rewriting every integer encountered. This is an example of data distribution involving data that is independent of any specific input term.

In contrast, consider constructing a strategy that will rewrite every integer in a term so that all integers are equal to the first integer in the term. For example, for a given term t if the first integer in t is 27 then all integers in t should be rewritten to 27. This is an example of data distribution involving data that is dependent on the input term. In the area of program transformation, the distribution of dependent data throughout a term is typically more common than the independent distribution of data. For example, variable renaming, function in-lining, and constant propagation all require the distribution of dependent data through a term structure.

Strategic/rewriting systems are often provided with extensions in order to enhance their ability to describe the distribution of data. Parameterization is one extension that is widely used as a mechanism for data distribution. For example, ASF+SDF [1] has been extended with a fixed collection of parameterizable traversal functions [4]. Another extension is to allow rule instances to be dynamically constructed using problem dependent data. In Stratego [10] for example, a mechanism is provided making it possible to alter rule bases at runtime through the dynamic construction and destruction of rules.

In this paper we look at higher-order extensions to strategic programming. Specifically we will describe how the higher-order rules, strategies, and traversals of a strategic programming language called TL can be used to effectively distribute (dependent) data throughout term structures. Though TL is presently a theoretical framework, a restricted dialect of TL has been in implemented in the HATS[3] system [6] and is freely available. All of the examples presented in this paper have been implemented in HATS.

The remainder of the paper is organized as follows. Section 2 gives an overview of TL. Section 3 takes an in-depth look at the inner workings of a strategic implementation of set union in TL. Section 4 looks at two manipulations common in the area of program transformation. Section 5 discusses some related work, and Section 6 concludes.

[3] Other than differences in syntax, the primary restriction is that the construction of user-defined strategies is not supported in HATS.

2 An Overview of TL

TL [14] is an *identity-based* higher-order strategic system for rewriting parse trees. We use the term identity-based to denote rewriting systems in which the failure of rule application to a term leaves the term unchanged. We use the term *failure-based* to denote systems where the unsuccessful application of a rule to a term yields a special failure value. In contrast to TL, the strategic programming systems Stratego [11] and Elan [2] are *failure-based*.

In TL, a domain (i.e., a term language) is defined using an Extended-BNF and terms also called *parse expressions* are described using a special notation. TL supports the constructs, combinators and strategic constants shown in Figure 1.

skip	A strategy constant that never applies
$lhs \rightarrow rhs$ if *condition*	A conditional first-order strategy
$lhs \rightarrow s^n$ if *condition*	A conditional strategy of order $n+1$
$s_1^n ; s_2^n$	Sequential composition
s_1^n <+ s_2^n	Left-biased choice
s_1^n +> s_2	Right-biased choice
$I(s^n)$	A unary combinator that does nothing
$fix(s^1)$	The fixed point application of the first-order strategy s^1
$transient(s^n)$	A unary combinator restricting the application of s^n

Fig. 1. The basic constructs of TL

In addition to the constructs shown in Figure 1, TL also provides a number of one-layer generic traversals providing the ability to construct user-defined traversals. These constructs are not central to the topic of this paper and are therefore omitted. Instead we simply present a number of generic traversals that form part of the TL traversal library.

2.1 Term Notation

Let $G = (N, T, P, S)$ denote a context-free grammar where N is the set of nonterminals, T is the set of terminals, P is the set of productions, and S is the start symbol. Given an arbitrary symbol $B \in N$ and a string of symbols $\alpha = X_1 X_2 ... X_m$ where for all $1 \leq i \leq m : X_i \in N \cup T$, we say B derives α iff the productions in P can be used to expand B to α. Traditionally, the expression $B \stackrel{*}{\Rightarrow} \alpha$ is used to denote that B can derive α in zero or more expansion steps. Similarly, one can write $B \stackrel{+}{\Rightarrow} \alpha$ to denote a derivation consisting of one or more expansion steps.

In TL, we write $B[[\alpha']]$ to denote an *instance* of the derivation $B \stackrel{+}{\Rightarrow} \alpha$ whose resulting value is a parse tree having B as its root symbol. In TL,

expressions of the form $B[[\alpha']]$ are referred to as *parse expressions*. In the parse expression $B[[\alpha']]$ the string α' is an *instance* of α because nonterminal symbols in α' are constrained through the use of subscripts. Subscripted nonterminal symbols are referred to a *schema variables* or simply *variables* for short. TL also considers a schema variable (e.g., B_i) to be a parse expression in its own right.

Within a given scope all occurrences of schema variables having the same subscript denote the same variable. The purpose of placing subscripts on schema variables is to enable grammar derivations to be restricted with respect to one or more equality-oriented constraints. The difference between a nonterminal B and a schema variable B_i is that B is traditionally viewed as a set (or syntactic category) while B_i is a typed variable quantified over the syntactic category B.

When the dominating symbol and specific structure of a parse expression is unimportant the parse expression will be denoted by variables of the form $t, t_1, ...$ or variables of the form $tree, tree_1, tree_2$, and so on. Parse expressions containing no schema variables are called *ground* and parse expressions containing one or more schema variables are called *non-ground*. And finally, within the context of rewriting or strategic programming, *trees* as described here can and generally are viewed as *terms*. When the distinction is unimportant, we will refer to *trees* and *terms* interchangeably.

2.2 Rules

TL supports conditional labeled first-order rewrite rules of the form:

$$\text{label} : lhs \rightarrow rhs \text{ if } condition$$

where lhs is a term, rhs is a strategic expression whose evaluation yields a term, and the label and conditional portion are optional components. Higher-order rules have the form:

$$\text{label} : lhs \rightarrow s^n \text{ if } condition$$

where s^n is a strategic expression whose evaluation yields a (possibly higher-order) rule. When parsing higher-order rules, the $\rightarrow$ associates to the right. An abstract example of a second-order condition-free rule is:

$$r : lhs_1 \rightarrow lhs_2 \rightarrow rhs_2$$

In order to disambiguate the internal structure (e.g., conditional components) of higher-order rules one may enclose the righthand side of a rule in parenthesis.

$$\text{label} : lhs \rightarrow (s^n) \text{ if } condition$$

As a notational convenience, labeled higher-order rules without conditions may be written in curried form when appropriate. For example, a rule of the form:

$$r : x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4$$

can be equivalently written as:

$$r\ x_1 : x_2 \rightarrow x_3 \rightarrow x_4$$

or even as:

$$r\ x_1\ x_2 : x_3 \rightarrow x_4$$

2.2.1 Rule Conditions

The conditional portion of a rule is a *match expression* consisting of one or more *match equations*. The symbol $\ll$, adapted from the ρ-calculus [5], is used to denote first-order matching modulo an empty equational theory. Let t_2 denote a ground tree and let t_1 denote a parse expression which may contain one or more schema variables. The equation $t_1 \ll t_2$ is a match equation. Equivalently we may also write $t_2 \gg t_1$. A match equation is a boolean valued operation that produces a substitution σ as a by-product. A substitution σ binding schema variables to ground parse expressions is a solution to $t_1 \ll t_2$ if $\sigma(t_1) = t_2$ with $=$ denoting a boolean valued test for syntactic equality.

A *match expression* is a boolean expression involving one or more match equations. Match expressions may be constructed using the standard boolean operators: $\wedge, \vee, \neg$. A substitution σ is a solution to a match expression m iff $\sigma(m)$ evaluates to true using the standard semantics for boolean operators.

2.2.2 Rule Application

The application of a conditional rewrite rule r to a tree t is expressed as $r(t)$ where r is either a label or an anonymous rule value e.g., $lhs \rightarrow s^n$. We adopt a curried notation in the style of ML where application is a left-associative implicit operator and parentheses are used to override precedence or may be optionally included to enhance readability. For example, $r\ t$ denotes the application of r to t and has the same meaning as $r(t)$.

2.3 Some First-Order Traversals from the TL Library

TL provides support for user-defined first-order traversals. TL also provides a number of standard generic first-order traversals. There are three degrees of freedom for a generic traversal: (1) whether a term is traversed bottom-up or top-down, (2) whether the children of a term are traversed from left-to-right

or right-to-left, and (3) whether a standard *threaded* semantics or a *broadcast* semantics is used to propagate strategies within the traversal (see Section 2.6).

Figure 2 gives a list of the most commonly used generic traversals. The first traversal is TDL. This traversal will traverse the term it is applied to in a top-down left-to-right fashion. With the exception of TD_BR which is discussed in Section 2.6, the remaining entries in the table have similar descriptions. The last two traversals perform a fixed point computation with respect to a given traversal scheme.

Traversal	bottom-up	top-down	left-to-right	right-to-left	threaded	broadcast
TDL		√	√		√	
TDR		√		√	√	
TD_BR		√				√
BUL	√		√		√	
BUR	√			√	√	
FIX_TDL		√	√		√	
FIX_TDR		√		√	√	

Fig. 2. General first-order traversals

2.4 Higher-Order Strategies

TL is a restricted higher-order strategic programming framework. TL is restricted because it only permits the application of higher-order strategies to ground terms. For example, strategies may not be applied to other strategies or rules as is allowed in the ρ-calculus [5]. In TL, the result of applying an order n strategy to a (ground) term is a strategy of order $n-1$.

From an operational perspective, a higher-order traversal traverses a term and applies a higher-order strategy s^n to every term encountered. Because the strategy being applied is of order n, the result of an application will be a strategy of order $n-1$. If a traversal visits m terms, then m strategies of order $n-1$ will be produced. Let s_1^{n-1}, s_2^{n-1} , ... , s_m^{n-1} denote the strategies resulting from such a traversal. In TL, a variety of binary strategic combinators can be used to combine the strategic results s_1^{n-1}, s_2^{n-1}, ... , s_m^{n-1} into a strategic expression (i.e., a strategy). Let $\oplus$ denote a binary combinator such as sequential composition, left-biased choice, right-biased choice, or a user-defined binary combinator. Higher-order traversals will combine these strategies into a strategic expression of the form:

$$s_1^{n-1} \oplus s_2^{n-1} \oplus ... \oplus \ s_m^{n-1}$$

Traversal	**bottom-up**	**top-down**	**left-to-right**	**right-to-left**	$\oplus$	τ
rcond_tdl		√	√		+>	*I*
rcond_tdr		√		√	+>	*I*
lcond_tdl		√	√		<+	*I*
lcond_tdr		√		√	<+	*I*
rcond_bul	√		√		+>	*I*
rcond_bur	√			√	+>	*I*
lcond_bul	√		√		<+	*I*
lcond_bur	√			√	<+	*I*
seq_tdl		√	√		;	*I*
seq_tdr		√		√	;	*I*
seq_bul	√		√		;	*I*
seq_bur	√			√	;	*I*

Fig. 3. General higher-order traversals

There is one technical detail that has been omitted from the above explanation. In addition to combining strategies using a binary combinator, a higher-order traversal also uniformly applies a unary combinator τ to every resultant strategy. Thus, the actual strategy produced is:

$$\tau(s_1^{n-1}) \oplus \tau(s_2^{n-1}) \oplus ... \oplus \tau(s_m^{n-1})$$

In practice, the unary combinator that is most useful is the *transient* combinator with the I combinator playing the role of a default. The *transient* combinator is described in Section 2.5.

TL provides support for user-defined higher-order traversals. TL also provides a number of standard generic higher-order traversals. There are four degrees of freedom for a generic higher-order traversal: (1) whether a term is traversed bottom-up or top-down, (2) whether the children of a term are traversed from left-to-right or right-to-left, (3) which binary combinator should be used to compose the result strategies, and (4) which unary combinator should be used to wrap each resulting strategy.

Figure 3 gives a list of the most commonly used generic traversals. The first traversal in this list is *rcond_tdl*. This traversal will traverse the term it is applied to in a top-down left-to-right fashion. The result strategies will then be composed using the right-biased choice combinator and finally each result strategy will be wrapped in the unary combinator I. The remaining entries in the table have similar descriptions.

2.5 The transient Combinator

The transient combinator is a very special combinator in TL. This combinator restricts a strategy so that it may be applied *at most once*. The "at most once" property characterizes the *transient* combinator and motivates the introduction of *skip* into the framework of TL. We define *skip* as a strategy whose application never succeeds.

Figure 4 gives some relationships between two abstract strategic constants ϵ and δ and the combinators $<+$ and ;. These relationships are considered from the perspective of a failure-based framework as well as an identity-based framework. In failure-based systems such as Stratego and ELAN, ϵ is typically called *id* or *identity* and δ is typically called *fail*. In the identity-based framework of TL, ϵ is called *id* and δ is called *skip*.

Strategy	Failure-Based Semantics	Identity-based Semantics
$\epsilon\ t$	t	t
$\delta\ t$	δ	t
$\epsilon <+ s$	ϵ	ϵ
$s <+ \epsilon$	$s <+ \epsilon$	$s <+ \epsilon$
$\delta <+ s$	s	s
$s <+ \delta$	s	s
$\epsilon\ ;\ s$	s	s
$s\ ;\ \epsilon$	s	s
$\delta\ ;\ s$	δ	s
$s\ ;\ \delta$	δ	s

Fig. 4. The semantics of *id*, *skip*, and *fail*

TL defines a strategy of the form $transient(s)$ as a strategy that *reduces* to the strategy *skip* if the application of the strategy s has been observed. Furthermore, only the innermost (i.e., closest enclosing) transient can observe the application of a strategy. This restriction is needed to prevent a cascading sequence of reductions for strategies containing nested transients.

Transients open the door to *self-modifying* strategies. When using a traver-

sal to apply a self-modifying strategy to a term, a different strategy may be applied to every term encountered during a traversal. For example, let $int_1 \rightarrow int[[2]]$ denote a rule that rewrites an arbitrary integer to the value 2. If such a rule is applied to a term in a top-down fashion all of the integers in the term will be rewritten to 2. Now consider the following self-modifying transient strategy:

$$transient(int_1 \rightarrow int[[1]]) <+$$
$$transient(int_1 \rightarrow int[[2]]) <+$$
$$transient(int_1 \rightarrow int[[3]])$$

When applied to a term in a top-down fashion, this strategy will rewrite the first integer encountered to 1, the second integer encountered to 2, and the third integer encountered to 3. All other integers will remain unchanged.

2.6 Traversal Mechanisms

TL provides two types of term traversal: a *threaded* traversal and a *broadcasting* traversal. In a *threaded* traversal (e.g., TDL, TDR, BUL, BUR), terms are visited in sequential order and a single strategy is passed from term to term. A diagram showing the behavior of a threaded traversal can be seen in Figure 5.

In a *broadcasting* traversal (e.g., TDL_BR) a distinct copy of the strategy resulting from an application will be given to all of the children of a term. For example, the evaluation of the strategic expression $TDL\_BR(s)t$ will first apply the strategy s to the term t. Recall that in the most general case (i.e., when transients are present in the strategy), the result of such an application will alter both s as well as t. Let s' and t' respectively denote the strategy and term resulting from the application of s to t. Since TDL_BR is a broadcasting traversal, a distinct copy of s' will be applied to each of the sub-terms of t'. A diagram showing the behavior of a broadcasting traversal can be seen in Figure 6.

3 A Benchmark: Set Union

We believe that set union has characteristics similar to a number of common transformational activities. For example, variations of set union can be used as the basis for variable renaming, data flow analysis, control flow analysis, symbolic resolution in Java class files [14], as well as field distribution and

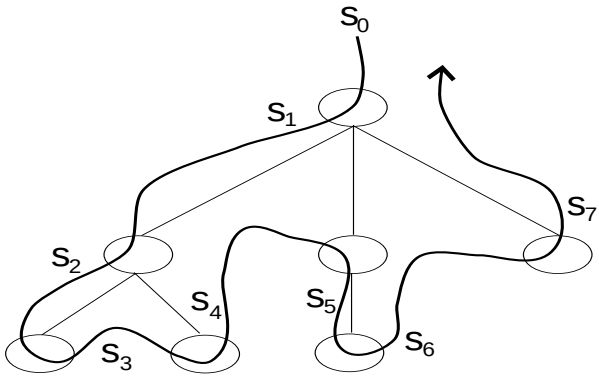

Fig. 5. Diagram of the threaded traversal *TDL* from the perspective of strategy application

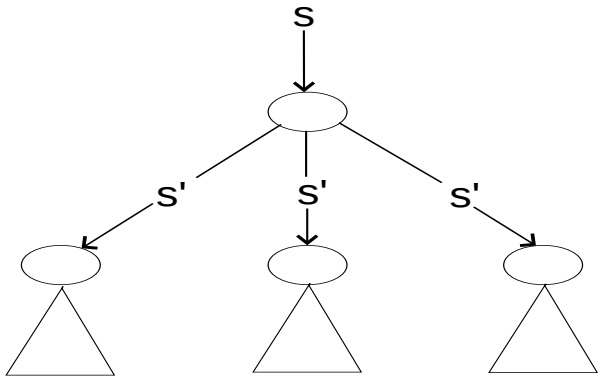

Fig. 6. Diagram of the broadcasting traversal *TDL_BR* from the perspective of strategy application

method method table construction [15] in Java class files. Thus, because of its wide range of applicability, we consider set union to be a benchmark problem for a strategic programming system.

In this section we look at how the union benchmark can be solved in TL. Our approach is to lift basic operations on data (e.g., insertion of an element into a set, etc.) to the strategy level. For example, when implementing union, we wish to create a strategy that inserts a particular element into our union set only if the element does not already occur in the set. In TL the construction of these types of problem specific first-order strategies can be accomplished though higher-order strategies.

In Figure 7 a BNF grammar is given describing a language of set/sequence expressions. The meta-symbols of the grammar are ::=, (), |, <, >, “, and ”. The symbol () is used to denote the epsilon symbol, domain variables are

enclosed in pointy brackets and terminal symbols are enclosed in quotes.

In Figure 8, *keep* and *add* are strategies realizing primitive operations on sets such as adding an element to an empty set. The strategy *union_s* is higher-order and defines a single computational step (e.g., a strategy that will "union" one element to a set). And finally, the strategy *make_union* performs its respective set operation by first properly instantiating *union_s* with respect to every element in set_1 and then applying the resulting strategy to the set_2.

set_expr	::= set set_op set | set
set	::= "{" es "}"
es	::= e es | ()
e	::= <id> | "(" <id> <id> ")"
set_op	::= "union"

Fig. 7. A BNF describing set/sequence expressions

keep e_1	:	$es[[e_1\ es_2]] \rightarrow es[[e_1\ es_2]]$
add e_1	:	$es[[\ ]] \rightarrow es[[e_1]]$
union_s	:	$es[[\ e_1\ es_1\ ]] \rightarrow transient((keep\ e_1) <+ (add\ e_1))$
make_union	:	$set\_expr[[set_1\ union\ set_2]] \rightarrow TDL(lcond\_tdl\ union\_s\ set_1)\ set_2$

Fig. 8. Instantiation and application of second-order strategies to terms

3.1 A Closer Look at the Implementation of Union in TL

The strategic theme here is to decompose a set expression $\{a_1, a_2, ..., a_n\} \cup \{e_1, e_2, ..., e_m\}$ into a sequence of incremental strategies each of which can be used to evaluate an expression of the form: $S \cup \{e_i\}$. The higher-order strategy *union_s* generates such incremental strategies. Specifically, when given the context $es[[\ e_1\ es_1\ ]]$, *union_s* will extract the element e_1 and produce a *transient* strategy consisting of the conditional composition $keep(e_1) <+ add(e_1)$.

Building on *union_s* is the strategic expression $(lcond\_tdl\ union\_s\ set_1)$ which traverses set_1 producing the conditional composition of instances of *union_s*; one instance for each element in set_1. The resulting strategy is then applied to set_2 using the traversal TDL. Keeping this in mind, let us trace the strategic evaluation of the expression $set_1 \cup set_2$ where $set_1 = \{x_1\ x_2\ x_3\ x_4\}$ and $set_2 = \{y_1\ x_2\ x_3\ y_2\}$.

The result of $(lcond\_tdl\ union\_s\ set_1)$ and its application to the first term in set_2 are shown in Figures 9 through 13. Figure 9 shows the initial strategy

$\{\overset{\Downarrow}{y_1}\ x_2\ x_3\ y_2\}$

transient($es[[x1\ es_2]] \rightarrow es[[x1\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x1]]$)
$<\!\!+$ transient($es[[x2\ es_2]] \rightarrow es[[x2\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x2]]$)
$<\!\!+$ transient($es[[x3\ es_2]] \rightarrow es[[x3\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x3]]$)
$<\!\!+$ transient($es[[x4\ es_2]] \rightarrow es[[x4\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x4]]$) $\quad \{\overset{\Downarrow}{y_1}\ x_2\ x_3\ y_2\}$

Fig. 9. Union with TDL traversal – The term y_1 in set_2 is unaffected

transient($es[[x1\ es_2]] \rightarrow es[[x1\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x1]]$) $\quad \{y_1\ \overset{\Downarrow}{x_2}\ x_3\ y_2\}$
~~$<\!\!+$ transient($es[[x2\ es_2]] \rightarrow es[[x2\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x2]]$)~~ $\quad \{y_1\ \overset{\Downarrow}{x_2}\ x_3\ y_2\}$
$<\!\!+$ transient($es[[x3\ es_2]] \rightarrow es[[x3\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x3]]$)
$<\!\!+$ transient($es[[x4\ es_2]] \rightarrow es[[x4\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x4]]$)

Fig. 10. Union with TDL traversal – The term x_2 changes the strategy

transient($es[[x1\ es_2]] \rightarrow es[[x1\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x1]]$) $\quad \{y_1\ x_2\ \overset{\Downarrow}{x_3}\ y_2\}$
~~$<\!\!+$ transient($es[[x2\ es_2]] \rightarrow es[[x2\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x2]]$)~~
~~$<\!\!+$ transient($es[[x3\ es_2]] \rightarrow es[[x3\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x3]]$)~~ $\quad \{y_1\ x_2\ \overset{\Downarrow}{x_3}\ y_2\}$
$<\!\!+$ transient($es[[x4\ es_2]] \rightarrow es[[x4\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x4]]$)

Fig. 11. Union with TDL traversal – The term x_3 changes the strategy

transient($es[[x1\ es_2]] \rightarrow es[[x1\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x1]]$) $\quad \{y_1\ x_2\ x_3\ \overset{\Downarrow}{y_2}\ \}$
~~$<\!\!+$ transient($es[[x2\ es_2]] \rightarrow es[[x2\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x2]]$)~~
~~$<\!\!+$ transient($es[[x3\ es_2]] \rightarrow es[[x3\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x3]]$)~~
$<\!\!+$ transient($es[[x4\ es_2]] \rightarrow es[[x4\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x4]]$) $\quad \{y_1\ x_2\ x_3\ \overset{\Downarrow}{y_2}\ \}$

Fig. 12. Union with TDL traversal – The processing the term y_2 has no effect

$\{y_1\ x_2\ x_3\ y_2\ \Downarrow\}$
~~transient($es[[x1\ es_2]] \rightarrow es[[x1\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x1]]$)~~ $\quad \{y_1\ x_2\ x_3\ y_2\ x_1\ \}$
~~$<\!\!+$ transient($es[[x2\ es_2]] \rightarrow es[[x2\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x2]]$)~~
~~$<\!\!+$ transient($es[[x3\ es_2]] \rightarrow es[[x3\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x3]]$)~~
$<\!\!+$ transient($es[[x4\ es_2]] \rightarrow es[[x4\ es_2]] <\!\!+\ es[[\ ]] \rightarrow es[[x4]]$)

Fig. 13. Union with TDL traversal – The term x_1 is added to the union

applied to set_2. Figures 10 and 11 show how the strategy changes as it encounters (is applied to) the elements x_2 and x_3 respectively. The application of the of the strategy to the element y_2 has no effect and is shown in Figure 12. And finally, in Figure 13 the traversal reaches the end of set_2 at which time the element x_1 is added. Note that in this case, both the strategy and set_2 are changed by the application. In a similar fashion, x_4 is added yielding $\{y_1\ x_2\ x_3\ y_2\ x_1\ x_4\}$ as the final term and *skip* as the final strategy.

4 Adaptations to Common Transformational Objectives

In this section we look at TL solutions to two common transformational objectives that arise in the area of program transformation. We would like to mention that these examples were inspired from similar examples presented

in [10].

Both examples are considered with respect to the grammar fragment defined in Figure 14. The meta-symbols of the grammar are ::=, (), |, <, >, “, ”, [, and]. The symbol () is used to denote the epsilon symbol, domain variables are enclosed in pointy brackets, terminal symbols are enclosed in quotes, and optional portions of productions are enclosed in square brackets.

4.1 Variable Renaming

In this example, we consider the variable renaming problem for a small block structured imperative language. (Note that the grammar given Figure 14 permits blocks to be nested). The TL solution makes use of a function *new* that has the ability to generate unique variable names.

The code in Figure 15 highlights some of the issues that must be addressed when renaming variables in this setting. First, variables may be redeclared within a nested block. However, it is assumed that variables may not be redundantly declared within a given declaration list. Second, variable declarations may include an assignment to an initial expression which may contain occurrences of previously declared variables.

When dealing with declarations having initialization expressions, one must be careful to associate variables with their proper declarations. For example, in Figure 15 in the declaration $int\ x1 = x1+1$ in the inner block, the reference to the variable $x1$ occurring in the initialization expression $x1 + 1$ is actually a reference to the previous declaration of $x1$ in the outer block. Thus it would be incorrect to rename $int\ x1 = x1 + 1$ to $int\ new\_x1 = new\_x1 + 1$. Instead, the renaming should result in something like $int\ new\_x1 = x1 + 1$.

Another difficulty in this example results from the structure of a block as defined by the grammar. Specifically, a block has intentionally been defined to consist of a declaration list followed by a statement list. Note that renaming must occur both within the declaration list as well as the statement list.

Figure 16 gives a TL implementation of variable renaming. An overview of our strategic approach to the variable renaming problem is as follows. Blocks are processed in an inside-out manner (i.e., nested blocks first). When a block is encountered, its declaration list will be traversed in a top-down fashion and a strategic expression will be constructed that is capable of renaming all variables within the block (variables occurring in both the declaration list as well as the statement list). Special care is taken to assure that variables occurring in *initializing expressions* (i.e., expressions on the right-hand sides of assignments in declarations) do not have their variables inappropriately renamed.

prog	::=	block
block	::=	“{” dec_list stmt_list “}”
dec_list	::=	dec “;” dec_list | ()
dec	::=	type id
	|	type id “=” expr
	|	“fun” id “(” id_list “)” “=” expr
type	::=	“int” | “bool” | ...
stmt_list	::=	stmt “;” stmt_list | ()
stmt	::=	assign | block | ...
assign	::=	id “=” expr
expr	::=	cond | logical_expr
cond	::=	“if” expr “then” expr “else” expr
logical_expr	::=	rel | ...
rel	::=	expr “=” expr
	|	E
	|	...
E	::=	E “+” T | E “-” T | T
T	::=	T “*” F | F “/” F | F
F	::=	id | num | “(” expr “)” | id “(” expr_list “)” | ...
id_list	::=	id [“,” id_list] | ()
expr_list	::=	actual [“,” expr_list | ()
actual	::=	expr
id	::=	<ident>
num	::=	<integer>
...		

Fig. 14. A grammar fragment of a small block structured imperative language

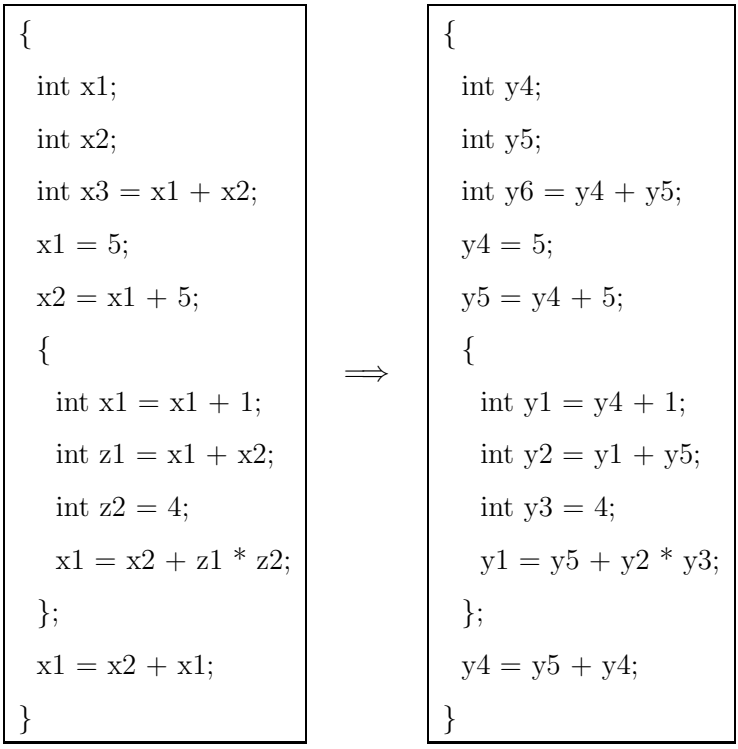

Fig. 15. Considerations when renaming variables: A block before and after variable renaming

$$
\begin{array}{lll}
restricted\ id_1\ id_2 : & dec[[\ type_1\ id_1 = expr_1\ ]] \rightarrow dec[[\ type_1\ id_2 = expr_1\ ]]) \\
\\
free\ id_1\ id_2 : & id_1 \rightarrow id_2 \\
\\
gen\_rename : & dec_1 \rightarrow transient((restricted\ id_1\ id_2) <+ (free\ id_1\ id_2)) \\
\text{if} & id_2 \ll new \wedge \\
& (dec_1 \ll dec[[\ type_1\ id_1\ ]] \vee dec_1 \ll dec[[\ type_1\ id_1 = expr_1]]) \\
\\
rename : & block_1 \rightarrow TD\_BR(lcond\_tdl\ gen\_rename\ dec\_list_1)\ block_1 \\
\text{if} & block_1 \ll block[[dec\_list_1\ stmt\_list_1]] \\
var\_rename : & prog_1 \rightarrow BUL\ rename\ prog_1
\end{array}
$$

Fig. 16. The Strategies for renaming variables

In the TL implementation shown in Figure 16 the strategies *restricted* and $free$ are third-order strategies in curried form that when given a variable name id_1 and a corresponding fresh variable name id_2 will yield a first-order rule describing a specific kind of renaming. The strategy $restricted\ id_1\ id_2$ describes the rewriting that should occur when the declaration of id_1 is encountered. In particular, the declaration of id_1 should be renamed to id_2, but the initializing expression should remain untouched. The strategy $free\ id_1\ id_2$ describes the rewriting that should occur in all other cases.

Building on the *restricted* and $free$ rules, is the higher-order strategy *gen_rename*. When applied to a declaration, *gen_rename* will create a transient of the form:

$$transient((restricted\ id_1\ id_2) <+ (free\ id_1\ id_2))$$

Note that this transient strategy that can only be applied once and will perform either a restricted or free rename. During the course of a top-down traversal, the idea is to have this transient apply to the declaration which generated it after which it will reduce to *skip* for all subtrees of that declaration. If this can be accomplished, then any traversal that continues on to the initialization expression will leave all occurrences of the declared variable unchanged. In addition to this behavior, we would like the renaming to continue for the rest of the block (e.g., the remaining declarations and statements). It is precisely this behavior that can be accomplished by TD_BR.

One way of understanding the effect of TD_BR when used in conjunction with a transient is that TD_BR captures the notion of "*not below*" with respect to a tree structure. The notion of "*not below*" was first used in TAMPR [3]. For a given tree t and a given leaf x, let p denote a *path* from the root of t to the leaf x. Let s denote a first-order strategy (containing no transient combinators). The traversal $TD\_BR\ transient(s)\ t$ will apply s at most once on every *path* in t. For example, if s applies at a particular point in a path, then $transient(s)$ will reduce to *skip* after this application and will therefore not apply anywhere else on the path.

Given this understanding of the interaction between TD_BR and the *transient* combinator, let us consider the parse expression $dec\_list[[\ dec_1;\ dec\_list_1\ ]]$. When applied to this term, the strategy $TD\_BR\ s$ will first apply s to $dec\_list[[\ dec_1;\ dec\_list_1\ ]]$ yielding the strategy s'. A **copy** of the strategy s' is then broadcast to each of the children of $dec\_list[[dec_1; dec\_list_1]]$. In particular, both dec_1 and dec_list_1 will receive their own copy of s'. More specifically, let us consider what happens when s is $transient(\ (restricted\ id_1\ id_2) <+ (free\ id_1\ id_2))$. In this case, the application of s to $dec\_list[[dec_1; dec\_list_1]]$ will leave s unchanged (e.g., $s = s'$). Next a copy of s' will be broadcast to both dec_1 and dec_list_1. If dec_1 is the declaration responsible for generating

s', then s' will apply to dec_1 but will not apply to any subterm below dec_1 (e.g., the initializing expression in dec_1). In contrast, within dec_list_1 s' will continue attempting to apply and broadcast its own copy of s' to its children. This will enable the strategy $(free\ id_1\ id_2)$ within the transient s' to rename all remaining occurrences of id_1 to id_2 within the block which is what is desired.

And finally, in the strategy *var_rename* the traversal BUL causes all the blocks in a program to be renamed in an inside-out fashion.

4.2 Naïve Function In-lining

When performing function in-lining the goal is to replace a function call with an instance of its body. This body instance is obtained by substituting the formal parameters associated with the function definition by the actual parameters associated with the call. An example of in-lining is shown in Figure 17. In Figure 18, a TL implementation for performing naïve function in-lining is given. The strategy *fun_inline* is naïve because it does not consider problems that may arise as a result of recursive and mutually recursive function definitions or address efficiency issues resulting from the duplication of expressions corresponding to actual arguments.

The strategy *fun_inline* uses matching to split a block into its declaration list and statement list. The declaration list is then processed by the strategy *fun_dec* which creates an in-lining strategy for each function declaration and composes the results into a strategic expression. This strategic expression is then applied to the original declaration list in order to in-line all the function calls within the declaration list. Then this in-lined declaration list is again processed by the strategy *fun_dec*. This time the resulting strategy is applied to the statement list which has the effect of in-lining all function calls. The resulting statement list is then cleaned up (e.g., excess parenthesis are removed from expressions) by the strategy *remove_parens* whose implementation is not shown. Finally, the resulting statement list is substituted for the statement list in the original block, as is the in-lined declaration list.

The strategy *fun_dec* accomplishes its transformational objective through the help of the strategy *inline*. This strategy is given the name of a function id_1, its formal parameter list id_list_1, and its body $expr_1$ in curried form. With this information, the strategy *inline* is capable of rewriting a function call $F[[\ id_1(expr\_list_1)\ ]]$ to an appropriately in-lined body $F[[\ (expr_2)\ ]]$. It accomplishes this with the help of the strategy *zip*.

As a definition the strategy *zip* is simply a macro and serves no other purpose than to enhance the readability of the conditional portion of the *inline* strategy. Operationally, the body of *zip* will first perform a traversal on id_list

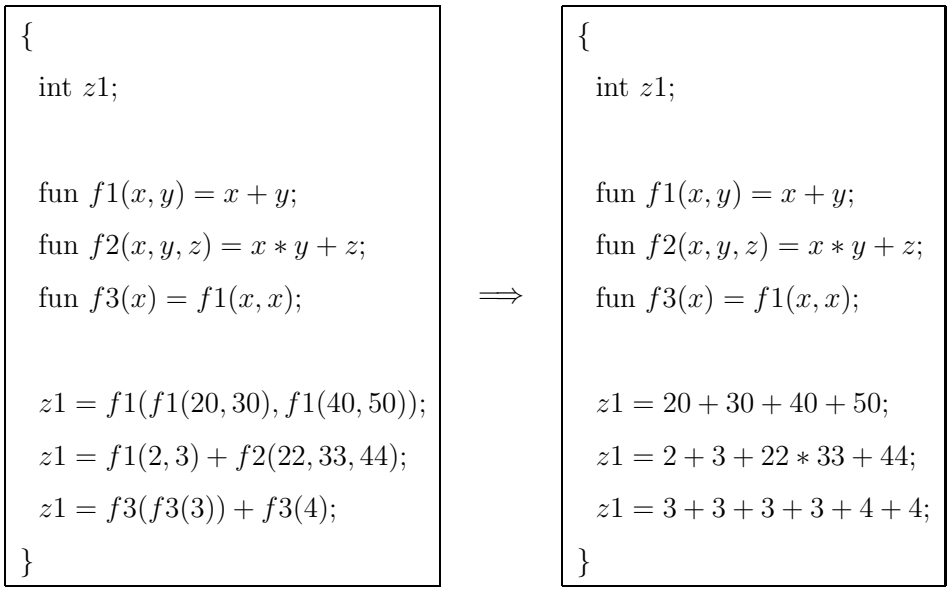

Fig. 17. An example of function in-lining

$$
\begin{array}{lll}
\mathit{formal\_to\_actual} & : & \mathit{id}_1 \rightarrow \mathit{transient}(\mathit{actual}[[\mathit{expr}_1]] \rightarrow F[[\ \mathit{id}_1\ ]] \rightarrow F[[\ (\mathit{expr}_1)\ ]]) \\
\\
\mathit{zip\ id\_list}_1\ \mathit{expr\_list}_1 & : & \mathit{lcond\_tdl}(\mathit{lcond\_tdl\ formal\_to\_actual\ id\_list}_1)\ \mathit{expr\_list}_1 \\
\\
\mathit{inline\ id}_1\ \mathit{id\_list}_1\ \mathit{expr}_1 & : & F[[\ \mathit{id}_1(\mathit{expr\_list}_1)\ ]] \rightarrow F[[\ (\mathit{expr}_2)\ ]] \\
 & \text{if} & \mathit{expr}_2 \ll \mathit{TDL}(\mathit{zip\ id\_list}_1\ \mathit{expr\_list}_1)\ \mathit{expr}_1 \\
\\
\mathit{fun\_dec} & : & \mathit{dec}[[\mathit{fun\ id}_1(\mathit{id\_list}_1) = \mathit{expr}_1]] \rightarrow \mathit{inline\ id}_1\ \mathit{id\_list}_1\ \mathit{expr}_1 \\
\\
\mathit{remove\_parens} & : & \ldots \\
\\
\mathit{fun\_inline} & : & \mathit{block}[[\mathit{dec\_list}_1\ \mathit{stmt\_list}_1]] \rightarrow \mathit{block}[[\mathit{dec\_list}_1\ \mathit{stmt\_list}_3]] \\
 & \text{if} & \mathit{dec\_list}_2 \ll \mathit{TDL}(\mathit{lcond\_tdl}\ \mathit{fun\_dec}\ \mathit{dec\_list}_1)\ \mathit{dec\_list}_1 \\
 & & \mathit{stmt\_list}_2 \ll \mathit{TDL}(\mathit{lcond\_tdl}\ \mathit{fun\_dec}\ \mathit{dec\_list}_2)\ \mathit{stmt\_list}_1 \\
 & & \mathit{stmt\_list}_3 \ll \mathit{TDL}\ \mathit{remove\_parens}\ \mathit{stmt\_list}_2
\end{array}
$$

Fig. 18. A TL implementation of naïve function in-lining

the formal parameter list of a function. This traversal will create one *transient* strategy for each formal parameter *id* in *id_list*. Let s denote the resulting strategic expression. Next, a traversal on the actual parameter list *expr_list* is performed with the strategy s. This will result in a strategic expression consisting of a collection of rules of the form $F[[\ id_1\ ]] \rightarrow F[[\ (expr_1)\ ]]$, where id_1 is a formal parameter and $expr_1$ is a corresponding actual parameter. The transient combinator mentioned previously is needed to assure that the proper correspondences between formals and actuals are created. When viewed collectively, the resulting rules are capable of rewriting formal parameters to actual parameters within the body of a function yielding an in-lined instance of that function.

5 Related Work

The higher-order nature of TL rules can be understood as a form of currried rewrite rule. In this context, curried arguments can be bound during the course of a higher-order generic traversal. The composition of strategies created during such generic traversal is related to a morphism. Specifically, the one-layer generic traversal combinators that are used to construct full traversals are similar but not identical to hylomorphisms over rose trees found in functional programming frameworks [8][9]. Similar observations have been made by others. For example, the catamorphism *fold* $b \oplus$ can be understood in strategic terms as performing a bottom-up term traversal on the structure of a list where the binary function $\oplus$ of the fold could be used to realize either a type-preserving rewriting function or a type-unifying accumulating function. This connection between catamorphisms and strategic driven term traversal is made in [7].

The ρ-calculus [5] is a failure-based rewriting framework in which matching modulo an equational theory provides the mechanism for the syntactic comparison of terms. In the ρ-calculus the distinction between a rule and a term to which a rule is applied is blurred. Both rules and terms are considered ρ-terms. This uniform treatment is reminiscent of the relationship between functions and terms in the λ-calculus. And, similar to the λ-calculus, in the ρ-calculus there are no restrictions regarding variable occurrences within a term. In particular, free variables may be introduced on the right-hand side of a rule. In fact, the right-hand side of a rule may itself be a rule, seamlessly opening the door to higher-order strategies.

In contrast to the ρ-calculus, TL is a restricted higher-order language. In TL, the name capture problem is sidestepped by the restriction that higher-order strategies only be applied to ground terms (and not to other strategies).

Recall that ground terms do not contain (free) schema variables. As a result of this restriction, alpha-conversion, as it is defined in the lambda-calculus is not required. In TL, all schema variables within a higher-order strategy fall within a single scope and must be (statically) distinguished accordingly within the definition.

The notion of creating problem specific instances of rules is a core capability of Stratego [10]. These dynamic rewrite rules are named rules that can be instantiated at runtime (i.e., dynamically) yielding a rule instance which is then added to the existing rule base. Dynamic rewrite rules are placed in the "where" portion of another rule and thus have access to information from their surrounding context. Similar to our approach, the input term itself is the driver behind the instantiation of rule variables. The lifetime of dynamic rules can be explicitly constrained in strategy definitions by the scoping operator $\{ | \ldots | \}$.

Primary differences between the higher-order strategies in TL and the scoped dynamic rules described in [10] are the following:

(i) TL higher-order strategies can be used as the basis of constructing *strategic expressions* that are created dynamically. The $\oplus$ and τ combinators provide the user explicit control over the combinators used to construct this strategy. Stratego views the dynamic instantiation of rules as a rule base (i.e., a strategy where rules are composed using the left-biased combinator and newly created rules are placed on the left-most end of the rule base). It would be interesting to extend the dynamic rule generation mechanism of Stratego to enable more control over the structure of dynamically generated rule bases. This idea has been recently proposed by Martin Bravenboer [13].

(ii) In Stratego, rule instances can be incrementally added and removed from a rule base. In TL, strategic expressions are created during the course of a separate pass(es) over a term structure. We believe that a separate pass is conceptually cleaner from the perspective of reasoning about the correctness of such structures. However, Stratego's incremental approach is more efficient and also allows a refined control over the contents of such rule bases. On the other hand, the *transient* combinator of TL also allows some degree of control over the contents of strategic expressions.

(iii) The incremental nature of Stratego's rule bases is similar to the operational or denotational environment models used to describe the semantics of scope. This facilitates thinking about the construction of rule bases in an incremental fashion. In TL, the user is strongly encouraged to think of strategic expressions in a more holistic manner [16].

(iv) Though the transient combinator has no direct analogy within scoped dynamic rewrite rules, its effects can be simulated in Stratego [13]. However, it is somewhat unclear whether a single approach/method can be used in Stratego to simulate all the behaviors resulting from the interaction between higher-order strategies and transients.

6 Conclusion

TL is based on the premise that higher-order rewriting provides a mechanism for dealing with the distribution of data conforming to the tenets of rewriting. In a higher-order framework, the distribution of data is expressed as rule. Instantiation of such rules can be done using standard (albeit higher-order) mechanisms controlling rule application (e.g., traversal). Typically, a traversal-driven application of a higher-order rule will result in a number of instantiations. If left unstructured, these instantiations can be collectively seen as constituting a rule base whose creation takes place dynamically. However, such rule bases can encounter difficulties with respect to confluence and termination. In order to address this concern we also lift the notion of strategy construction to the higher-order as well. That is, instantiations are structured to form strategic expressions. Nevertheless, in many cases, simply lifting first-order control mechanisms to the higher-order does not permit the construction of strategic expressions that are sufficiently refined. This difficulty is alleviated though the introduction of the *transient* combinator. The interplay between transients and more traditional control mechanisms enables a variety of instances of the distributed data problem to be elegantly solved in a higher-order setting.

References

[1] Bergstra, J. A., Heering, J., and Klint, P., editors, "Algebraic Specification". ACM Press/Addison-Wesley, 1989.

[2] Borovansky, P., Kirchner, C., Kirchner, H., Moreau, P.-E., and Ringeissen, C., *An Overview of ELAN*. In C. Kirchner and H. Kirchner, eds., International Workshop on Rewriting Logic and its Applications, Vol. **15** (1998) of Electronic Notes in Theoretical Computer Science, France, Elsevier Science, http://www.elsevier.com/locate/entcs/volume15.html.

[3] Boyle, J. M., Harmer, T. J., and Winter, V. L., *The TAMPR program transformation system: Simplifying the development of numerical software*. In Arge, E., Pettersen, A. M., and Langtangen, H. P., editors, "Modern Software Tools for Scientific Computing", pages 353–372. Birkhauser, 1997.

[4] Brand, M. G. J. van den, Klint, P., and Vinju, J. J., *Term Rewriting with Traversal Functions*. ACM Transactions on Software Engineering and Methodology (TOSEM), pp 152-190, **12:2** (2003).

[5] Cirstea, H., and Kirchner, C., *Intoduction to the rewriting calculus*. INRIA Research Report RR-3818, December 1999.

[6] HATS. http://faculty.ist.unomaha.edu/winter/hats-uno/HATSWEB/index.html

[7] Lämmel, R., Visser, J., and Kort, J., *Dealing with Large Bananas*. In Johan Jeuring, editor, Workshop on Generic Programming, Ponte de Lima, July 2000. Technical Report, Universiteit Utrecht.

[8] Meijer, E., Fokkinga, M. M., and Paterson, R., *Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire*. In J. Hughes, editor, FPCA'91: Functional Programming Languages and Computer Architecture, LNCS, pp 124–144, Vol. **523** (1991), Springer-Verlag.

[9] Meijer, E., and Jeuring, J., *Merging Monads and Folds for Functional Programming*. In J. Jeuring and E. Meijer, editors, 1st International Spring School on Advanced Functional Programming Techniques, B astad, Sweden, LNCS, pp 228–266, Vol. **925** (1995), Springer-Verlag.

[10] Visser, E., *Scoped dynamic rewrite rules*. In M. van den Brand and R. Verma, editors, Rule Based Programming (RULE'01), Vol. **59/4** (2001) of Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers, http://www.elsevier.com/locate/entcs/volume15.html.

[11] Visser, E., Benaissa, Z., and Tolmach, A., *Building Program Optimizers with Rewriting Strategies*. Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98).

[12] Visser, E. and Benaissa, Z., *A Core Language for Rewriting*. Eds. C. Kirchner and H. Kirchner Electronic Notes in Theoretical Computer Science Vol. **15** (1998), Elsevier Science Publishers, http://www.elsevier.com/locate/entcs/volume15.html.

[13] Visser, E., Personal communication, Feb. 18, 2004.

[14] Winter, V. L., and Subramaniam, M., *The Transient Combinator, Higher-Order Strategies, and the Distributed Data Problem*. Science of Computer Programming (accepted).

[15] Winter, V. L., Roach, S., and Fraij, F., *Higher-Order Strategic Programming: A Road to Dependable Software*. Submitted to IEEE Transactions on Dependable and Secure Computing.

[16] Winter, V., Roach, S., Wickstrom, G., *Transformation-Oriented Programming: A Development Methodology for High Assurance Software*. "Advances in Computers" Vol. **58**, (2003).