Electronic Notes in Theoretical Computer Science 124 (2005) 149–170 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Strategy Construction in the Higher-Order Framework of TL

# Victor L. Winter [1](#_bookmark0) *,*[2](#_bookmark0)

*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

# Introduction

The concept of distributing data within a term structure is central to rewrite- based computation [[11](#_bookmark37)]. In [[14](#_bookmark38)] 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](mailto:vwinter@mail.unomaha.edu)

1571-0661 © 2005 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

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 inte- ger 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 in- volving data that is dependent on the input term. In the area of program transformation, the distribution of dependent data throughout a term is typ- ically 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](#_bookmark27)] has been extended with a fixed collection of parame- terizable traversal functions [[4](#_bookmark30)]. Another extension is to allow rule instances to be dynamically constructed using problem dependent data. In Stratego [[10](#_bookmark32)] 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 program- ming. Specifically we will describe how the higher-order rules, strategies, and traversals of a strategic programming language called TL can be used to ef- fectively 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](#_bookmark1) system [[6](#_bookmark33)] 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](#_bookmark2) gives an overview of TL. Section [3](#_bookmark9) takes an in-depth look at the inner workings of a strategic implementation of set union in TL. Section [4](#_bookmark19) looks at two manip- ulations common in the area of program transformation. Section [5](#_bookmark25) discusses some related work, and Section [6](#_bookmark26) concludes.

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

# An Overview of TL

TL [[14](#_bookmark38)] 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](#_bookmark37)] and Elan [[2](#_bookmark28)] 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](#_bookmark3).

*skip*

*lhs* → *rhs* if *condition*

*lhs* → *s* if *condition*

n

*s*n; *s*n

1 2

*s*n *<*+ *s*n

1 2

*s*n +*> s*2

1

*I*(*s*n)

*fix*(*s*1) *transient*(*s*n)

A strategy constant that never applies

A conditional first-order strategy

A conditional strategy of order *n* +1 Sequential composition

Left-biased choice Right-biased choice

A unary combinator that does nothing

The fixed point application of the first-order strategy *s*1

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](#_bookmark3), TL also providesa 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.

* 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* ∈ *N* and a string of symbols *α* = *X*1*X*2*...Xm* where for all 1 ≤ *i* ≤ *m* : *Xi* ∈ *N* ∪*T* , we say *B* derives *α* iff the productions in *P* can be used to expand *B* to *α*. Traditionally, the expression

*B* ⇒∗ *α* is used to denote that *B* can derive *α* in zero or more expansion steps.

Similarly, one can write *B* ⇒+

more expansion steps.

*α* to denote a derivation consisting of one or

In TL, we write *B*[[*α*']] to denote an *instance* of the derivation *B* ⇒+ *α*

whose resulting value is a parse tree having *B* as its root symbol. In TL,

expressions of the form *B*[[*α*']] are referred to as *parse expressions*. In the parse expression *B*[[*α*']] 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., *Bi*) 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 re- spect to one or more equality-oriented constraints. The difference between a nonterminal *B* and a schema variable *Bi* is that *B* is traditionally viewed as a set (or syntactic category) while *Bi* 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 expres- sions 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 unim- portant, we will refer to *trees* and *terms* interchangeably.

* 1. *Rules*

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

label : *lhs* → *rhs* 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:

label : *lhs* → *sn* if *condition*

order) rule. When parsing higher-order rules, the → associates to the right. where *sn* is a strategic expression whose evaluation yields a (possibly higher- An abstract example of a second-order condition-free rule is:

*r* : *lhs*1 → *lhs*2 → *rhs*2

In order to disambiguate the internal structure (e.g., conditional compo-

nents) of higher-order rules one may enclose the righthand side of a rule in parenthesis.

label : *lhs* → (*sn*) 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 → *x*2 → *x*3 → *x*4

can be equivalently written as:

*r x*1 : *x*2 → *x*3 → *x*4

or even as:

*r x*1 *x*2 : *x*3 → *x*4

* + 1. *Rule Conditions*

The conditional portion of a rule is a *match expression* consisting of one or more *match equations*. The symbol , adapted from the *ρ*-calculus [[5](#_bookmark31)], 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 *t*2 is a match equation. Equivalently we may also write *t*2 *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 *t*2 if *σ*(*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

operators: ∧*,* ∨*,* ¬. A substitution *σ* is a solution to a match expression *m* iff equations. Match expressions may be constructed using the standard boolean *σ*(*m*) evaluates to true using the standard semantics for boolean operators.

* + 1. *Rule Application*

where *r* is either a label or an anonymous rule value e.g., *lhs* → *sn*. We adopt The application of a conditional rewrite rule *r* to a tree *t* is expressed as *r*(*t*) 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*).

* 1. *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](#_bookmark8)). Figure [2](#_bookmark4) 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](#_bookmark8), 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

* 1. *Higher-Order Strategies*

TL is a restricted higher-order strategic programming framework. TL is re- stricted 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](#_bookmark31)]. 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 *sn* 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 *sn*−1, *sn*−1 , ... , *sn*−1 denote the strategies resulting

1 2 *m*

from such a traversal. In TL, a variety of binary strategic combinators can

be used to combine the strategic results *sn*−1, *sn*−1, ... , *sn*−1 into a strategic

1 2 *m*

expression (i.e., a strategy). Let ⊕ denote a binary combinator such as se-

quential 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:

*sn*−1 ⊕ *sn*−1 ⊕ *...* ⊕ *sn*−1

1 2 *m*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Traversal | bottom-up | top-down | left-to-right | right-to-left | ⊕ | *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 ex- planation. 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:

*τ* (*sn*−1) ⊕ *τ* (*sn*−1) ⊕ *...* ⊕ *τ* (*sn*−1)

1 2 *m*

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](#_bookmark6).

TL provides support for user-defined higher-order traversals. TL also pro- vides 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 tra- versed 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](#_bookmark5) 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.

* 1. *The transient Combinator*

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

Figure [4](#_bookmark7) 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 typi- cally 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

|  |  |  |
| --- | --- | --- |
| *ϵ t δ t* | *t δ* | *t t* |
| *ϵ <*+*s s <*+*ϵ δ <*+*s s <*+*δ* | *ϵ*  *s <*+*ϵ s*  *s* | *ϵ*  *s <*+*ϵ s*  *s* |
| *ϵ* ; *s*  *s* ; *ϵ*  *δ* ; *s*  *s* ; *δ* | *s s δ*  *δ* | *s s s*  *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 → *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 → *int*[[1]]) *<*+ *transient*(*int*1 → *int*[[2]]) *<*+ *transient*(*int*1 → *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.

* 1. *Traversal Mechanisms*

TL provides two types of term traversal: a *threaded* traversal and a *broadcast- ing* 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](#_bookmark10).

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](#_bookmark11).

# 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](#_bookmark38)], as well as field distribution and

s0



s1

s7

s2

s4

s5

s3

s6

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

# s

s'

s' s'



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

method method table construction [[15](#_bookmark40)] 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.

expressions. The meta-symbols of the grammar are ::=, (), |, *<*, *>*, “, and In Figure [7](#_bookmark12) a BNF grammar is given describing a language of set/sequence ”. 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](#_bookmark13), *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

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

*keep e*1

*add e*1

:

:

*union s*

:

*make union* :

*es*[[*e*1 *es*2]] → *es*[[*e*1 *es*2]]

*es*[[ ]] → *es*[[*e*1]]

*es*[[ *e*1 *es*1 ]] → *transient*((*keep e*1) *<*+ (*add e*1 ))

*set expr*[[*set*1 *union set*2]] → *T DL*(*lcond tdl union s set*1) *set*2

* 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*, ..., an*}∪

{*e*1*, e*2*, ..., em*} into a sequence of incremental strategies each of which can be used to evaluate an expression of the form: *S* ∪ {*ei*}. 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 ∪ *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](#_bookmark14) through [13](#_bookmark18). Figure [9](#_bookmark14) shows the initial strategy

transient(*es*[[*x*1 *es*2]] → *es*[[*x*1 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*1]])

*<*+ transient(*es*[[*x*2 *es*2]] → *es*[[*x*2 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*2]])

*<*+ transient(*es*[[*x*3 *es*2]] → *es*[[*x*3 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*3]])

⇓

{*y*1 *x*2 *x*3 *y*2 }

⇓

*<*+ transient(*es*[[*x*4 *es*2]] → *es*[[*x*4 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*4]]) {*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*[[*x*1 *es*2]] → *es*[[*x*1 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*1]]) {*y*1 *x*2 *x*3 *y*2}

⇓

—*<*+—t—ran—si—en—t(*e*—*s*[—[*x*2—*e*—*s*2—]] →——*es*—[[*x*—2 —*es*—2]]—*<*+——*es*—[[ ]—] →—*e*—*s*[—[*x*—2]]—)

*<*+ transient(*es*[[*x*3 *es*2]] → *es*[[*x*3 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*3]])

*<*+ transient(*es*[[*x*4 *es*2]] → *es*[[*x*4 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*4]])

{*y*1 *x*2 *x*3 *y*2}

Fig. 10. Union with TDL traversal – The term *x*2 changes the strategy

⇓

transient(*es*[[*x*1 *es*2]] → *es*[[*x*1 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*1]]) {*y*1 *x*2 *x*3 *y*2}

—*<*+—t—ran—si—en—t(*e*—*s*[—[*x*2—*e*—*s*2—]] →——*es*—[[*x*—2 —*es*—2]]—*<*+——*es*—[[ ]—] →—*e*—*s*[—[*x*—2]]—)

—*<*+—t—ran—si—en—t(*e*—*s*[—[*x*3—*e*—*s*2—]] →——*es*—[[*x*—3 —*es*—2]]—*<*+——*es*—[[ ]—] →—*e*—*s*[—[*x*—3]]—)

*<*+ transient(*es*[[*x*4 *es*2]] → *es*[[*x*4 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*4]])

⇓

{*y*1 *x*2 *x*3 *y*2}

Fig. 11. Union with TDL traversal – The term *x*3 changes the strategy

⇓

transient(*es*[[*x*1 *es*2]] → *es*[[*x*1 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*1]]) {*y*1 *x*2 *x*3 *y*2 }

—*<*+—t—ran—si—en—t(*e*—*s*[—[*x*2—*e*—*s*2—]] →——*es*—[[*x*—2 —*es*—2]]—*<*+——*es*—[[ ]—] →—*e*—*s*[—[*x*—2]]—)

—*<*+—t—ran—si—en—t(*e*—*s*[—[*x*3—*e*—*s*2—]] →——*es*—[[*x*—3 —*es*—2]]—*<*+——*es*—[[ ]—] →—*e*—*s*[—[*x*—3]]—)

⇓

*<*+ transient(*es*[[*x*4 *es*2]] → *es*[[*x*4 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*4]]) {*y*1 *x*2 *x*3 *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 ⇓ }

——t—ra—ns—ien—t(—*es*—[[*x*—1 *e*—*s*2—]]—→—*es*—[[*x*—1—*es*—2]]—*<*—+—*es*—[[ —]] →——*es*—[[*x*—1]—])

—*<*+—t—ran—si—en—t(*e*—*s*[—[*x*2—*e*—*s*2—]] →——*es*—[[*x*—2 —*es*—2]]—*<*+——*es*—[[ ]—] →—*e*—*s*[—[*x*—2]]—)

—*<*+—t—ran—si—en—t(*e*—*s*[—[*x*3—*e*—*s*2—]] →——*es*—[[*x*—3 —*es*—2]]—*<*+——*es*—[[ ]—] →—*e*—*s*[—[*x*—3]]—)

*<*+ transient(*es*[[*x*4 *es*2]] → *es*[[*x*4 *es*2]] *<*+ *es*[[ ]] → *es*[[*x*4]])

{*y*1 *x*2 *x*3 *y*2 *x*1 }

Fig. 13. Union with TDL traversal – The term *x*1 is added to the union

applied to *set*2. Figures [10](#_bookmark15) and [11](#_bookmark16) show how the strategy changes as it en- counters (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](#_bookmark17). And finally, in Figure [13](#_bookmark18) 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.

# Adaptations to Common Transformational Objectives

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

in [[10](#_bookmark32)].

fined in Figure [14](#_bookmark20). The meta-symbols of the grammar are ::=, (), |, *<*, *>*, “, ”, Both examples are considered with respect to the grammar fragment de- [, 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.

* 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](#_bookmark20) 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](#_bookmark21) 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 decla- rations 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](#_bookmark21) in the declaration *int x*1= *x*1 + 1 in the inner block, the reference to the variable *x*1 occurring in the initialization expression *x*1 + 1 is actually a reference to the previous declaration of *x*1 in the outer block. Thus it would be incorrect to rename *int x*1= *x*1+1 to *int new x*1= *new x*1+ 1. Instead, the renaming should result in something like *int new x*1= *x*1+ 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](#_bookmark22) 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

dec list dec

::=

::=

::=

::=

|

|

type ::=

stmt list ::=

stmt ::=

assign ::=

expr ::=

cond ::=

logical expr ::= rel ::=

|

|

E

T F

id list expr list actual id

num

...

::=

::=

::=

::=

::=

::=

::=

::=

block

“{” dec list stmt list “}” dec “;” dec list | ()

type id

type id “=” expr

“int” | “bool” | ... “fun” id “(” id list “)” “=” expr

stmt “;” stmt list | ()

assign | block | ... id “=” expr

cond | logical expr

“if” expr “then” expr “else” expr

rel | ...

expr “=” expr

E

...

E “+” T | E “-” T | T

T “\*” F | F “/” F | F

id | num | “(” expr “)” | id “(” expr list “)” | ... id [ “,” id list ] | ()

actual [ “,” expr list | ()

expr

*<*ident*>*

*<*integer*>*

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

=⇒

{

int y4;

int y5;

int y6 = y4 + y5; y4 = 5;

y5 = y4 + 5;

{

int y1 = y4 + 1;

int y2 = y1 + y5; int y3 = 4;

y1 = y5 + y2 \* y3;

};

y4 = y5 + y4;

}

{

int x1;

int x2;

int x3 = x1 + x2; x1 = 5;

x2 = x1 + 5;

{

int x1 = x1 + 1;

int z1 = x1 + x2; int z2 = 4;

x1 = x2 + z1 \* z2;

};

x1 = x2 + x1;

}

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

*restricted id*1 *id*2 :

*free id*1 *id*2

*dec*[[ *type*1 *id*1 = *expr*1 ]] → *dec*[[ *type*1 *id*2 = *expr*1 ]])

: *id*1 → *id*2

*gen rename*

:

if

*rename*

*var rename*

*dec*1 → *transient*((*restricted id*1 *id*2 ) *<*+ (*free id*1 *id*2 ))

*id*2 *new*∧

(*dec*1 *dec*[[ *type*1 *id*1 ]] ∨ *dec*1 *dec*[[ *type*1 *id*1 = *expr*1]])

: *block*1 → *TD BR*(*lcond tdl gen rename dec list*1 ) *block*1

if *block*1 *block*[[*dec list*1 *stmt list*1 ]]

: *prog*1 → *BUL rename prog*1

Fig. 16. The Strategies for renaming variables

In the TL implementation shown in Figure [16](#_bookmark22) 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 en- countered. 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 tran- sient 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 declara- tion. If this can be accomplished, then any traversal that continues on to the initialization expression will leave all occurrences of the declared variable un- changed. 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](#_bookmark29)]. 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 par- ticular, both *dec*1 and *dec list*1 will receive their own copy of *s*'. More specif- ically, 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 chil- dren. 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.

* 1. *Na¨ıve 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 pa- rameters associated with the call. An example of in-lining is shown in Figure

[17](#_bookmark23). In Figure [18](#_bookmark24), a TL implementation for performing na¨ıve function in-lining is given. The strategy *fun inline* is na¨ıve because it does not consider problems that may arise as a result of recursive and mutually recursive function defini- tions 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 pur- pose 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*

=⇒

{

int *z*1;

fun *f* 1(*x, y*)= *x* + *y*;

fun *f* 2(*x, y, z*)= *x* ∗ *y* + *z*; fun *f* 3(*x*)= *f* 1(*x, x*);

*z*1 = 20 + 30 + 40 + 50;

*z*1=2 + 3 + 22 ∗ 33 + 44;

*z*1 = 3 + 3+3+3+4+ 4;

}

{

int *z*1;

fun *f* 1(*x, y*)= *x* + *y*;

fun *f* 2(*x, y, z*)= *x* ∗ *y* + *z*; fun *f* 3(*x*)= *f* 1(*x, x*);

*z*1= *f* 1(*f* 1(20*,* 30)*,f* 1(40*,* 50));

*z*1= *f* 1(2*,* 3) + *f* 2(22*,* 33*,* 44);

*z*1= *f* 3(*f* 3(3)) + *f* 3(4);

}

Fig. 17. An example of function in-lining

*formal to actual*

: *id*1 → *transient*(*actual*[[*expr*1]] → *F* [[ *id*1 ]] → *F* [[ (*expr*1) ]])

*zip id list*1 *expr list*1 : *lcond tdl*(*lcond tdl formal to actual id list*1) *expr list*1

*inline id*1 *id list*1 *expr*1 : *F* [[ *id*1(*expr list*1) ]] → *F* [[ (*expr*2) ]]

if *expr*2 *T DL*(*zip id list*1 *expr list*1) *expr*1

*fun dec*

: *dec*[[*fun id*1(*id list*1) = *expr*1]] → *inline id*1 *id list*1 *expr*1

*remove parens*

: ...

*fun inline*

: *block*[[*dec list*1 *stmt list*1]] → *block*[[*dec list*1 *stmt list*3]]

if *dec list*2 *T DL*(*lcond tdl fun dec dec list*1) *dec list*1 *stmt list*2 *T DL*(*lcond tdl fun dec dec list*2) *stmt list*1 *stmt list*3 *T DL remove parens stmt list*2

Fig. 18. A TL implementation of na¨ıve 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 ]] → *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 col- lectively, 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.

# 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 cre- ated during such generic traversal is related to a morphism. Specifically, the one-layer generic traversal combinators that are used to construct full traver- sals are similar but not identical to hylomorphisms over rose trees found in functional programming frameworks [[8](#_bookmark34)][[9](#_bookmark35)]. Similar observations have been

made by others. For example, the catamorphism *fold b* ⊕ can be understood

of a list where the binary function ⊕ of the fold could be used to realize either in strategic terms as performing a bottom-up term traversal on the structure a type-preserving rewriting function or a type-unifying accumulating function.

This connection between catamorphisms and strategic driven term traversal is made in [[7](#_bookmark36)].

The *ρ*-calculus [[5](#_bookmark31)] is a failure-based rewriting framework in which match- ing 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 capa- bility of Stratego [[10](#_bookmark32)]. 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

can be explicitly constrained in strategy definitions by the scoping operator { driver behind the instantiation of rule variables. The lifetime of dynamic rules

| ... |}.

Primary differences between the higher-order strategies in TL and the

scoped dynamic rules described in [[10](#_bookmark32)] are the following:

*gic expressions* that are created dynamically. The ⊕ and *τ* combinators (i) TL higher-order strategies can be used as the basis of constructing *strate-* 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 com- binator and newly created rules are placed on the left-most end of the rule base). It would be interesting to extend the dynamic rule genera- tion 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](#_bookmark39)].

1. 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.
2. The incremental nature of Stratego’s rule bases is similar to the opera- tional 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](#_bookmark41)].
3. Though the transient combinator has no direct analogy within scoped dy- namic rewrite rules, its effects can be simulated in Stratego [[13](#_bookmark39)]. 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.

# 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 rewrit- ing. 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. How- ever, such rule bases can encounter difficulties with respect to confluence and termination. In order to address this concern we also lift the notion of strat- egy construction to the higher-order as well. That is, instantiations are struc- tured to form strategic expressions. Nevertheless, in many cases, simply lifting first-order control mechanisms to the higher-order does not permit the con- struction of strategic expressions that are sufficiently refined. This difficulty is alleviated though the introduction of the *transient* combinator. The inter- play 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. L¨ammel, 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).