

# The Transient Combinator, Higher-Order Strategies, and the Distributed Data Problem

Victor L. Winter<sup>1</sup> and Mahadevan Subramaniam

*Department of Computer Science*

*University of Nebraska at Omaha*

{ *vwinter@mail.unomaha.edu, msubramaniam@mail.unomaha.edu* }

---

## Abstract

The *distributed data problem* is characterized by the desire to bring together semantically related data from syntactically unrelated portions of a term. A strategic combinator called *transient* and a strategic constant called *skip* is introduced in the context of a higher-order strategic framework. The notion of traversal is lifted to the higher-order as well. The resulting framework allows the manipulation of data to be expressed directly in strategic terms. The impact of this dynamic approach to strategy creation is then explored on several instances of the distributed data problem. Problems considered include three strategic benchmarks as well as two transformations that arise within a class loader for the Java Virtual Machine.

*Key words:* distributed data problem, strategic programming, higher-order strategic programming, higher-order traversal, one-layer higher-order traversals, transformation, program transformation, rewriting, higher-order rewriting, transient combinator, skip strategy, TL, HATS, table normalization, Java class loader, strategic benchmarks, Java Virtual Machine, Sandia Secure Processor

---

## 1 The Distributed Data Problem

We introduce the term *distributed data problem* (DDP) to characterize the desire to bring together *semantically related* terms from *syntactically unre-*

---

<sup>1</sup> 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.

lated portions of a term. To see what we mean by this consider a simple term language  $\mathcal{L}$  consisting of sums involving symbolic constants and integers (e.g.,  $add(3, 1)$ ,  $add(add(7, c1), 4)$ , etc.). In this context, let us also consider a specification of a function called *swap* which states that the first two integers in a term should be swapped without otherwise disturbing the term structure. From the perspective of *swap*, the first two integers in a term are *semantically related*. However, we will shortly see that from the perspective of standard matching the first two integers in a term are *syntactically unrelated*. It is this discord that gives rise to the distributed data problem.

### 1.1 A Characterization of the DDP

Given a term  $t$ , let  $D_t$  denote the set of all subterms of  $t$ . The notion of being *semantically related* within the context of  $t$  can be abstractly defined as follows:

- (1) Define  $D_t^n = D_t \times D_t \times \dots \times D_t$  as the  $n$ -ary cross-product of  $D_t$  where the value of  $n$  is in part a function of  $t$  and in part a function of the specification under consideration.
- (2) Define a relation  $SR(t)$  such that an  $n$ -tuple of terms  $d_t^n \in D_t^n$  is semantically related iff  $d_t^n \in SR(t)$ .

Continuing on with our example, suppose we are asked to implement *swap* in a rewriting-based framework. For example, an implementation of *swap* should rewrite terms as shown below.

Initial Term	Result
$add(1, add(2, c))$	$\implies add(2, add(1, c))$
$add(1, add(2, add(3, 4)))$	$\implies add(2, add(1, add(3, 4)))$

A first attempt at capturing this type of rewriting in the form of a general rule might be:

$$add(x, add(y, z)) \rightarrow add(y, add(x, z))$$

where  $x$ ,  $y$ , and  $z$  are variables quantified over the terms in  $\mathcal{L}$ . Now, if we control the application of this rule using a suitable strategy, the result would be a strategic program that would correctly swap the first two integers of the terms shown in the previous table. Unfortunately, the proposed rule is overly general and a number of terms would be incorrectly rewritten.

Initial Term		Incorrect Result
$add(add(1, 2), add(3, 4))$	$==>$	$add(3, add(add(1, 2), 4))$
$add(1, add(c, 2))$	$==>$	$add(c, add(1, 2))$

The source of problem we are encountering is that within  $\mathcal{L}$  the syntactic distance between the first and second integers in a term may be arbitrarily large and therefore beyond the reach of matching or unification. It is this structural property of  $\mathcal{L}$  that makes the first and second integers in a term *syntactically unrelated*.

We define terms  $d_t^n \in D_t^n$  to be *syntactically unrelated* if an unbounded computational construct such as exhaustive rule application (to place the term into a normal form) or recursive traversal must be employed in order to construct  $d_t^n$ . In this context, the unbounded computational construct is seen as the mechanism by which the matching/unification capabilities within a strategic programming system are extended (e.g., a subterm can be retrieved from or carried to points arbitrarily deep within the term  $t$ ).

The reader should suspect that the notion of being syntactically unrelated is dependent upon the matching/unification capabilities of the strategic programming system  $\mathcal{W}$  as well as the term structures under consideration. For example, a system  $\mathcal{W}'$  in which AC unification is supported will consider a different set of terms to be syntactically unrelated when compared to a system  $\mathcal{W}''$  in which only first-order matching is supported. In response to this dependence, we define  $SU(\mathcal{W}, t)$  to be the relation consisting of all  $d_t^n \in D_t^n$  that are syntactically unrelated in  $t$  with respect to the matching/unification capabilities of the system  $\mathcal{W}$ .

Given these definitions, the distributed data problem for a given system  $\mathcal{W}$  and fixed term  $t$  can be defined as follows:

$$distributed\text{-}data\text{-}problem(\mathcal{W}, t) \stackrel{def}{=} SR(t) \cap SU(\mathcal{W}, t)$$

Instances of the distributed data problem arise in numerous settings including type-checking, program slicing, partial evaluation, variable renaming, function in-lining, constant propagation and constant folding, as well as a number of activities central to Java class loaders. In many cases, simple instantiated strategies needed to accomplish a desired task can easily be written (by hand) for a fixed term  $t$ . Unfortunately, the overly specific nature of such hand crafted solutions does not directly provide an automated solution to the more general problem. The transformational ideas presented in this paper provide higher-order mechanisms by which appropriate strategies can be automatically constructed for arbitrary  $t$ .

## 1.2 Contribution

In this paper, we develop a strategic programming language called *TL* (short for **T**ransformation **L**anguage) in which the combinators of a traditional first-order strategic programming system such as those described in [22] are lifted to a higher-order setting. Into the higher-order framework of *TL* we also introduce a combinator called a *transient*. The *transient* combinator restricts a strategy so that it can be applied only once. For example, let  $s$  denote an arbitrary strategy. In *TL* the expression  $\text{transient}(s)$  denotes a strategy in which  $s$  can only be applied once. Given a strategic mind-set, a *transient* strategy can be understood as a strategy that **transforms itself** into the strategy *skip* after its first successful application to a term. The strategy *skip* is similar to the identity strategy *id* in the sense that its application to any term will leave the term unchanged. The difference between *skip* and *id* is that *skip* never applies while *id* always applies. This distinction between *skip* and *id* impacts the semantic foundations upon which strategic systems are built. Systems such as Stratego [32][34] and ELAN [3][4] are *failure-based* in the sense that they are built upon the notion that if a strategy cannot be applied to a term, then the resulting value is *fail*. However, in order for *skip* to have the effect we desire, a more suitable semantic foundation is one that is *identity-based*. That is, where a term is left unchanged if a strategy cannot be applied to it.

While the notion of a transient may seem quite simple at first glance, a subtle interplay between higher-order strategies and transients lead to interesting and elegant solutions to problems involving term structures whose characteristics have been considered undesirable in the context of more traditional rewriting systems. To see what a transient is capable of, let us revisit the swap example discussed in Section 1. This will also allow us to begin to informally introduce the constructs of *TL*. Let  $1 \rightarrow 2$  and  $2 \rightarrow 1$  denote two rewrite rules that rewrite 1 to 2 and 2 to 1 respectively. Given two rules  $r_1$  and  $r_2$  let the expression  $r_1 \text{+>} r_2$  denote a *right-biased strategy* that when applied to a term  $t$  will apply rule  $r_2$  first and only attempt to apply  $r_1$  if the application of  $r_2$  fails. Furthermore, let us assume that when a rule application fails, it leaves the term it is applied to unchanged (i.e., the result is not the failure value as is typically the case in strategic programming). This is the framework of *TL*. In this framework, the term  $t = \text{add}(\text{add}(1, c), \text{add}(2, d))$  would be correctly rewritten by the following strategy

$$\text{transient}(2 \rightarrow 1) \text{+>} \text{transient}(1 \rightarrow 2)$$

assuming a generic traversal is used to apply this strategy to all the subterms of  $t$  in a left-to-right fashion.

In *TL* a generic traversal called *tdl* can be defined which performs a top-down

left-to-right traversal of the term to which it is applied. One would write  $tdl(s)t$  to denote a strategy that traverses the term  $t$  in a  $tdl$  fashion and attempts to apply the strategy  $s$  to every subterm of  $t$ . Thus, a strategy solving the instance of swap with respect to the term  $t$  would be:

$$tdl(transient(2 \rightarrow 1) +> transient(1 \rightarrow 2))$$

It is important to note that given any term a strategy similar to the one above can be written (by hand) that will correctly swap the first two integers of that term. The caveat being that the term must be inspected before the strategy can be constructed. It is to address this caveat that higher-order strategies are brought into play. For example, when applied to a term  $t$ , a second-order strategy could produce a first-order strategy of the kind shown above as its result. In particular, the second order strategy labeled *load2* can accomplish this:

$$replace : int_1 \rightarrow transient(int_2 \rightarrow int_1)$$

$$load2 : transient(replace) +> transient(replace)$$

Here  $replace : int_1 \rightarrow transient(int_2 \rightarrow int_1)$  is a labeled second-order rewrite rule that can only be successfully applied to terms of type integer (i.e.,  $int_1$  and  $int_2$  are variables quantified only over integer terms). For example, if this second-order rule is applied to the integer 1, the result will be the first-order rule  $transient(int_2 \rightarrow 1)$ . Similarly, the application to the integer 2 will yield  $transient(int_2 \rightarrow 2)$ . Given a suitable higher-order strategy and accompanying higher-order generic traversal incorporating a mechanism for strategy construction, the rule *load2* when applied to the term  $add(add(1, c), add(2, d))$  will produce the following first-order strategy:

$$transient(int_1 \rightarrow 1) +> transient(int_1 \rightarrow 2)$$

In *TL* a higher-order generic traversal called *rcond.tdl* can be defined that traverses a term  $t$  in a  $tdl$  fashion applying a higher-order strategy to every subterm in  $t$  combining the resulting strategies using the combinator  $+>$ . For example, the strategic expression  $(rcond.tdl \ load2 \ t)$  will traverse  $t$  in a  $tdl$  fashion applying *load2*. The evaluation of this strategic expression will yield a strategy that is appropriate for swapping the first two integers in the term  $t$ . The  $tdl$  application of this strategy to the term  $t$  then realizes the swap. This implementation of swap in *TL* is shown below.

---

## Implementation in $TL$

---

$replace : int_1 \rightarrow transient(int_2 \rightarrow int_1)$

$load2 : transient(replace) \rightarrow transient(replace)$

$swap : t \rightarrow tdl(rcond.tdl load2 t) t$

---

### 1.3 Outline

The remainder of the paper is organized as follows: Section 2 takes a general look at the activities involved in solving the DDP. Section 3 is a small example driven survey of the existing mechanisms used to implement solutions to the DDP. In Section 4 *table normalization* is introduced as an illustrative and somewhat classic example of the distributed data problem. Section 5 formally defines the syntax and semantics of  $TL$ . Section 6 gives several examples of how the transient combinator in conjunction with higher-order strategies can be used to solve distributed data benchmarks such as set union, intersection, and zip. Section 7 revisits the table normalization problem in a real world setting namely, *constant pool normalization* for the Java Virtual Machine (JVM). *Field distribution* for the JVM is also discussed. Section 8 gives a brief overview of the HATS transformation system and discusses the extent to which the ideas presented in this paper have been implemented. Section 9 discusses related work including Stratego, the  $\rho$ -calculus, ASF+SDF, the  $S'_\gamma$  calculus, ELAN, Strafunski, and Maude. Section 10 concludes.

## 2 A General Perspective of the Solution to the DDP

In general, a solution to the distributed data problem involves four distinct activities:

- (1) *The creation of data.* Data must be found or created and represented in a suitable form.
- (2) *The binding of data.* Data must be bound to variables.
- (3) *The distribution of data.* An unbounded computational construct such as recursive traversal must be employed or developed for transporting data to terms.
- (4) *The use of data.* The data is used at some point to manipulate some portion of the term under consideration.

In this article, we restrict our solution space to strategic programs whose input and output consists of a single term (e.g., a program). When applied to

an initial term  $t_0$ , a strategic program will incrementally modify  $t_0$  through the application of rewrite rules. Thus, during the course of computation, a sequence of terms:

$$t_0 \rightarrow t_1 \rightarrow \dots \rightarrow t_n$$

is produced where (1)  $t_0$  is the input to the strategic program, (2)  $t_n$  is the output of the strategic program, and (3) for all  $0 < i < n$ ,  $t_i$  denotes an intermediate form of  $t_0$  that is obtained from  $t_{i-1}$  through a rewriting step.

In order to accomplish a particular rewriting step (e.g., renaming a variable in a term), a rewrite rule may need to use information that, at least conceptually, may be stored in a structure separate from the term  $t_i$  being rewritten. We will refer to this kind of “external” information as *data*. Recall that the distributed data problem relates to the difficulties involved in making such *data* available at a particular position in a *term*. The focus of our work is on how higher-order strategies can be used to capture *data* at the strategy level rather than storing *data* in (term) structures separate from  $t_i$ .

### 2.1 Phase I: The Creation of Data

In the first phase, one or more values are created. Typically this is accomplished by the application of an accumulating strategy to an appropriate (sub)term. This accumulating strategy may collect a fixed number of values (e.g., a term or a tuple) or a varying number of values (e.g., a list).

It is worth noting that accumulated values are usually simple term structures such as tuples or flat lists. Furthermore, lists are usually homogeneous in the sense that all the elements of a list are of the same type. To our knowledge, there is not an example in the literature where an accumulated value is substantially more complex than a list (e.g., a heterogeneous list of lists).

In practice, the use of the auxiliary structures described in this section is widespread in strategic programming frameworks. Such structures occur in virtually every strategic program solving a non-trivial problem. As a result, auxiliary structures such as lists, even when implicitly defined by the system, are generally given first-class citizenship within the strategic framework. For example, strategies may be applied to such values directly and congruence relations may be defined using their structure.

## 2.2 Phase II: The Binding of Data

There are several approaches for binding a variable to an accumulated value, with strategy parameterization being the most common. Another possibility is to construct a strategy where a desired accumulated value is denoted by a free variable. This strategy can then be embedded in a scope (e.g., another strategy) in which this free variable is explicitly bound.

Term *normalization* is a technique that tackles the problem from an altogether different perspective. The goal of *normalization* is to alter the subterm relation through the exhaustive application of a collection of rewrite rules. For example, a term of the form

$$\text{add}(\text{add}(1, \text{add}(2, 3)), \text{add}(4, 5))$$

may be normalized to yield

$$\text{add}(1, \text{add}(2, \text{add}(3, \text{add}(4, 5))))$$

In the context of the distributed data problem, term normalization can be used to enable matching to simultaneously bind and distribute data. Notice that in this case the the first two integers will be at a fixed depth in the normalized term and thus within reach of standard matching. As a result of normalization, terms that were syntactically unrelated may thus become syntactically related.

## 2.3 Phase III: The Distribution of Data

Whether parameterization or the free variable technique for binding is used to bind distributed data, an auxiliary strategy must be employed for transferring data to appropriate subterms. Ultimately, this transfer of data will involve a recursive traversal which may be generic (e.g., top-down, bottom-up, etc.) or problem dependent. In the case where binding is achieved via parameterization, the problem is how to integrate/reconcile the parameterization of a strategy  $s$  with recursive traversals applying  $s$ .

## 2.4 Phase IV: The Use of Data

An extraction and/or conversion function (e.g., some form of lookup, zip, etc.) is typically developed to enable the data stored in auxiliary structures to be used to effect a change in a particular term. In some cases, the function is

trivial being little more than the identity function. However, if the accumulated value constitutes an aggregation (e.g., a list of values), then this structure will need to be traversed in some manner in order to extract the appropriate value.

### 3 A Small Survey of Mechanisms used to Solve the DDP

This section gives a brief overview and analysis of some examples of the distributed data problem and solutions that have been published in the literature.

#### 3.1 Type Checking

In [34], a strategic program is presented capable of type checking programs belonging to a small imperative language called Pico. In Pico, program blocks consist of a declaration list followed by a statement list. This is defined by the following abstract syntax.

$Block$	$: List(Decl) * Stat \rightarrow Program$
$Decl$	$: Id * Type \rightarrow Decl$
...	...
$Id$	$: String \rightarrow Id$

The basic idea of the type checker is to first rewrite all variables to their declared type (producing an intermediate program form) and then use basic type rules to simplify program constructs. For example, a type rule is given for simplifying an expression of the form  $Int + Int$  to  $Int$ . Similarly, a type rule is given for simplifying an assignment of the form  $Int := Int$  to  $Skip$ . In this approach, the statement list  $Stat$  will be type correct if it can be rewritten to  $Skip$ . The following table shows how a simple program might be rewritten by a strategic type checker.

Original Block	Distribute Type Info	Apply Type Rules	Etc.
Int x;	Int x;	Int x;	Int x;
Int y;	Int y;	Int y;	Int y;
x := 1;	Int := Int;	Skip;	Skip;
y := x + 2;	Int := Int + Int;	Int := Int;	Skip;

Before type rules can be applied, a preprocessing step must occur where variable occurrences within the statement list are rewritten to their declared types. The type of a variable can be found in the declaration list of a block (e.g.,  $List(Decl)$ ). We would like to point out the fact that the declaration list of a *Block* can have an arbitrary length as can the statement list. Thus the rewriting that occurs during this preprocessing step is a classic example of the distributed data problem which, in this case, is solved through the use of *contextual rules*. In particular, the following *contextual rule* is used to distribute the type data associated with variables in declarations over statements in  $s$  containing uses of the variable:

$$\text{InlTp: Block}(ds[Decl(Id(x),t)], s[Id(x)]) \rightarrow \text{Block}(ds, s[Tp(t)])$$

Here the context  $ds[Decl(Id(x),t)]$  denotes an occurrence of the declaration  $Decl(Id(x),t)$  within the declaration list  $ds$ . Similarly, the context  $s[Id(x)]$  denotes a use of the variable  $Id(x)$  within the statement sequence  $s$ .

Contexts provide an elegant abstraction for solving this type of distributed data problem. Furthermore, it turns out that *contexts* can be directly implemented using the primitive strategic constructs found in Stratego (e.g., match and build strategies, where clauses, and term traversals). In Stratego, the notation  $\langle s \rangle t$  denotes the application of the strategy  $s$  to the term  $t$ ,  $?t$  indicates that the term  $t$  should be matched,  $!t$  indicates that the term  $t$  should be built using the variable bindings in the current (match) environment, and  $onctd(s)t$  performs a top-down traversal on the term  $t$  searching for one subterm in  $t$  to which  $s$  can be successfully applied.

---

### Implementation in Stratego

---


$$\text{InlTp} : \text{Block}(ds, s) \rightarrow \text{Block}(ds, s')$$

$$\text{where } \langle onctd(?Decl(Id(x),t);$$

$$\text{where}(\langle onctd(!Id(x); !Tp(t)) \rangle s \Rightarrow s') \rangle ds$$


---

The implementation basically involves nested traversal in a fashion similar to how one might use nested for-loops to implement a bubble-sort in an imperative language. In this case, the outermost traversal (i.e., the first where-clause) walks across the declaration list  $ds$ . When a declaration  $Decl(Id(x),t)$  is encountered the values of  $x$  and  $t$  are bound and a traversal of the statement sequence  $s$  is initiated by the second where-clause with the goal of replacing a single occurrence of  $Id(x)$  with  $Tp(t)$ . If no occurrence of  $Id(x)$  is found, then the strategy fails at which point the next declaration in  $ds$  is tried. The exhaustive application of this strategy will replace all variables in  $s$  with their corresponding types. Adapting the syntax of *TL* to that of Stratego this type

in-lining strategy could be realized as follows:

---

**Implementation in pseudo-TL**

---

$replace : Decl(Id(x), t) \rightarrow Id(x) \rightarrow Tp(t)$

$InlTp : Block(ds, s) \rightarrow Block(ds, s')$  if  $s' \ll tdl(rcond\_tdl\ replace\ ds)s$

---

Here the higher-order rule *replace* captures the replacement of an identifier occurrence  $Id(x)$  with its type  $t$ . The expression  $(rcond\_tdl\ replace\ ds)$  denotes the application of the higher-order generic traversal *rcond\\_tdl* to the arguments *replace* and *ds*. The traversal *rcond\\_tdl* will perform a top-down left-to-right traversal of *ds* applying the rule *replace* to every declaration encountered. The resulting rules (i.e., the replacements) are then composed in a right-biased fashion using the combinator  $+>$  yielding a (dynamically created) first-order strategy. This first-order strategy is then applied to the term *s* using the traversal *tdl*, which performs a top-down left-to-right traversal. The resulting term is matched with the variable  $s'$  which replaces *s* in the resulting block.

In a variety of cases, the *contextual rules* of Stratego provide a capability that is similar to the higher-order strategies of *TL*. In fact, one could think of higher-order strategies as an extension of *contextual rules* and the *scoped dynamic rewrite rules* discussed in the next example.

### 3.2 Variable Renaming

In [32] the bound variable renaming problem is considered. A basic (conventional) algorithm is outlined in a functional style where a substitution list is used to keep track of the appropriate substitutions needed for renaming. When a construct binding the variable  $id_1$  is encountered, a new (i.e., fresh) variable  $id_2$  is generated and the tuple  $(id_1, id_2)$  is added to the substitution list. Then, when a variable use is encountered, that variable is looked up (using a lookup function) in the substitution list and the appropriate substitution is made. A pseudo-Stratego realization (taken from [32]) of the variable renaming algorithm for ML-style let-blocks is given below.

---

## Implementation in pseudo-Stratego using a Functional Style

---

$$\begin{aligned} \text{exprename}(\text{Let}([\text{VarDec}(x, t, e1)], e2), rn) = \\ & \text{Let}([\text{VarDec}(y, t, \text{exprename}(e1, rn))], \text{exprename}(e2, (x,y) : rn)) \\ & \text{where new} \Rightarrow y \\ \text{exprename}(\text{Var}(x), rn) = \\ & \text{Var}(\text{lookup}(x, rn)) \end{aligned}$$

---

Here the variable  $rn$  denotes the substitution list. When a declaration of a variable  $x$  is encountered in a let-block, a new variable  $y$  is generated and the tuple  $(x, y)$  is added to the substitution list  $rn$ . The body of the let-block  $e2$  is then renamed using the new substitution list, while the expression  $e1$  bound to  $x$  in the declaration is renamed using the original substitution list  $rn$ .

In [32], this basic algorithm is then adapted to a framework in which it is possible to dynamically create labeled rules. *Dynamic rules* are rules whose variables can be instantiated during the traversal of a term and whose instantiated forms can be added to the rule base during execution. Abstractly, this rule base can be seen as a set of labeled rule definitions whose cardinality and membership function can change dynamically. A *scoping* construct is then introduced in order to manage the rule definitions in this set (hence the title of the paper: Scoped Dynamic Rewrite Rules). Specifically, the scoping construct defines when rule definitions should be **removed** from the set (in contrast to the instantiations resulting from term traversal which indicate when a rule instance should be **added** to the set).

Given a framework in which scoped dynamic rewrite rules is supported, the variable renaming problem can be solved as follows. First, the labeled rule `RenameVar` is created defining variable renaming in general.

$$\text{RenameVar} : \text{Var}(x) \rightarrow \text{Var}(y)$$

It is important to note that `RenameVar`, when dynamically instantiated, stores the information needed in order to accomplish variable renaming and simultaneously obviates the need for the lookup function used in the basic algorithm discussed previously. The lookup function is no longer necessary because its effect is accomplished by *rule (base) application* – a primitive operation in the strategic framework. Thus, dynamic rule instantiations subsume the need for (1) tuple creation and (2) addition of tuples to substitution lists, while rule base application subsumes the need for (3) substitution list lookup.

In [32] the following strategy is given for renaming variables in ML style let-blocks using scoped dynamic rewrite rules.

---

## Implementation in Stratego

---

RenameVarDec:

$$\text{Let}([\text{VarDec}(x,t,e1)],e2) \rightarrow \text{Let}([\text{VarDec}(y,t,e1)],e2)$$

where new  $\Rightarrow$  y

$$; \text{rules}(\text{RenameVar} : \text{Var}(x) \rightarrow \text{Var}(y))$$

exprename =

$$\text{rec } r(\text{try}(\text{Let}([\text{VarDec}(\text{id},\text{id},r)],\text{id}));$$
$$\{| \text{RenameVar} :$$
$$\text{try}(\text{RenameVarDec} + \text{RenameVar});$$
$$(\text{Let}([\text{VarDec}(\text{id},\text{id},\text{id})],r) <+ \text{all}(r))$$
$$| \})$$

---

The idea of dynamically creating rule bases is similar to what we propose in this article. However, we take this idea one step further by enabling rule bases to be formed into strategies. Adapting the syntax of *TL* to that of Stratego the type variable renaming strategy could be realized as follows:

---

## Implementation in pseudo-TL

---

*RenameVarRule* :

$$\text{id1} \rightarrow \text{id2} \rightarrow \text{id1} \rightarrow \text{id2}$$

*aux\_rename* :

$$\text{Let}([\text{VarDec}(x,t,e1)],e2) \rightarrow \text{Let}([\text{VarDec}(y,t,e1)],e2')$$
$$\text{if } y \ll \text{new} \wedge e2' \ll \text{tdl}(\text{RenameVarRule } x \ y)e2$$

*exprename* :  $t \rightarrow \text{tdl}(\text{aux\_rename})$

---

Here the rule *RenameVarRule* is a third-order rule than when given two identifiers (in curried form) will yield a first-order rule capturing a specific renaming of variables. The strategy *aux\_rename* deals with the renaming associated with a single let-block. Within *aux\_rename*, the expression  $(\text{RenameVarRule } x \ y)$  will create an instance of a rule renaming  $x$  to  $y$ . This rule is then applied by *tdl* to the term  $e2$  yielding  $e2'$  which is then put in place of  $e2$  in the resulting let-block. Notice that the term  $e1$  is not effected by this rewrite, so the scoping rules for let-blocks are preserved. And finally, the strategy *exprename* uses *tdl* to traverse a term  $t$  applying the strategy *aux\_rename* to every let-block encountered. Thus, every variable declared in a let-block will be renamed.

### 3.3 General Replacements

In [5] *traversal functions* are presented as an extension of ASF+SDF rewrite rules. The capabilities of traversal functions is demonstrated by showing how various types of term replacements might be accomplished. We consider a replacement that involves incrementing each integer term by another term denoting a constant integer value. This integer increment is accomplished through a rewrite rule (i.e., a transformer) *incp* having two parameters. The first parameter is instantiated by the term  $t$  to which the rule is to be applied while the second parameter holds the constant integer value  $c$  which is to be used to increment every integer subterm in  $t$ . The traversal function *trafo* is used to transport  $c$  to every subterm in  $t$ . The *incp* equation applies (i.e., the increment occurs) only when the first argument of *incp* is a term of type integer. The solution presented here is taken directly from [5].

---

#### Implementation in ASF+SDF

---

**module** Tree-incp

**imports** Tree-syntax

**exports**

**context-free syntax**

        incp(TREE,NAT) -> TREE {traversal(trafo) }

**equations**

[1] incp(N1, N2) = N1 + N2

---

The following example, also taken from [5], operationally demonstrates how *incp* can be used to increment every integer subterm in  $f(g(1,2),3)$  by the constant 7:

```
incp(f(g(1,2),3),7) ->
f(incp(g(1,2),7), incp(3,7)) ->
f(g(incp(1,7),incp(2,7)), 10) ->
f(g(8,9),10)
```

Adapting the syntax of *TL* to that of ASF+SDF the *incp* could be implemented as follows:

---

## Implementation in pseudo-TL

---

$increment : N2 \rightarrow N1 \rightarrow N1 + N2$

$incp : t \rightarrow c \rightarrow tdl(incp\ c)\ t$

---

Here *increment* is a higher-order strategy that when applied to *c* will produce a rule of the form:  $N1 \rightarrow N1 + c$ . The traversal *tdl* will then apply this first-order rule everywhere in *t*.

### 3.4 Questions and Concerns

The examples above raise some interesting questions. For example, must the structure of data in an accumulated list always be simple? Should the manipulation of an accumulated value always be simple? Continuing on with this line of thought, does it make sense to consider the creation of lists whose elements are arbitrarily complex terms (e.g., a list of lists, a list of lists of lists)?

Strategic systems generally provide some sort of typing as a byproduct of their computational framework. For example, a rewrite rule of the form  $r : f(t_1, \dots, t_n) \rightarrow f(t'_1, \dots, t'_n)$  can only be successfully applied to terms of the form  $f(x_1, \dots, x_n)$ . Given this, the constructor *f* can be viewed as a type constraint and the rule *r* can be seen as being type preserving. In [5], traversal functions are classified as belonging to one of three possible types: (1) the sort-preserving *transformer*, (2) the *accumulator* that maps all types to a single type and in this sense can be thought of as being type unifying [21], and (3) the *accumulating transformer* that is a mixture of a transformer and an accumulator.

In a strategic setting, parameterization encourages the use of accumulators, especially when the goal of a strategy is to return an aggregation (e.g., a list) of values. Accumulators are type unifying not type preserving and typically rely on term-language extensions such as tuples or lists. Programming in such a framework may permit various kinds of errors that would otherwise not be possible or would be detected by a type system. The value of strong typing is recognized across the spectrum of computational frameworks from object oriented languages to functional languages. The strategic programming community also recognizes the value of strong typing. One of the contributions of the  $S'_\gamma$  calculus [21] is that it enables a strategic framework to use non-type preserving strategies such as accumulators while nevertheless reaping the benefits offered by type systems.

Accumulators, in the process of producing a value will typically strip constructors from a term with the goal of producing a term consisting only of

essential information. However, the constructors that are present in terms oftentimes can provide valuable information. The loss of information resulting from flattening these structures in to lists may present limitations for the use of accumulated values. Generally speaking, as the structural information in an aggregation diminishes, the sophistication of the extraction function will need to increase.

## 4 An Illustrative Example

The *table normalization* problem is a classic instance of the distributed data problem. The goal of table normalization is the removal of indirection from entries in a two-columned table. The solution to the table normalization problem has practical application to *constant pool normalization* within the Java Virtual Machine (JVM). In Section 7.1 the solution to the table normalization problem is re-visited in the context of the JVM.

### 4.1 The Table Normalization Problem

Suppose we are given a term representing a 2-column table whose entries are of the form  $(index, data)$  where *index* is an integer describing the position of the entry in the table and *data* is value that may either be an *index* or a *character*. Given an entry  $(i, d)$  if *d* is of type character, then we say that the entry  $(i, d)$  is *resolved*. Otherwise, *d* is of type index and the entry is *unresolved*.

**Definition 1** Given a table  $t$ , a **resolution step** for  $t$  involves two entries and is defined as follows: If  $(i, j)$  and  $(j, d)$  denote two entries in our table, then the entry  $(i, j)$  may be resolved to  $(i, d)$  yielding a new table  $t'$  such that  $(\forall k : k \neq i \rightarrow t[k] = t'[k]) \wedge t'[i] = (i, d)$

Note that the definition of a resolution step places no constraints on the ordering relationship between  $i$  and  $j$ . In particular,  $i$  may be positionally less-than or greater-than  $j$ .

**Definition 2** A table  $t$  is **normalized** by applying a sequence of resolution steps until no further resolution is possible.

A table having  $n$  entries is well-formed with respect to resolution step sequences if any entry can be (fully) resolved in fewer than  $n$  steps. In other words, a table is well-formed if all of its resolution step sequences are cycle-free. Given this constraint, we claim (without proof) that for well-formed tables resolution step sequences are convergent (i.e., terminating and confluent). This means that regardless of the order in which they are applied, a sequence

Index	Index or Data		Index	Data
1	5		1	a
2	b		2	b
3	a	⇒	3	a
4	2		4	b
5	3		5	a

Fig. 1. A Well-Formed Table and its Normal Form

$r_1 : 1 \rightarrow 5$
$r_2 : 2 \rightarrow b$
$r_3 : 3 \rightarrow a$
$r_4 : 4 \rightarrow 2$
$r_5 : 5 \rightarrow 3$

Fig. 2. A ground rule-set encoding resolution steps for the table in Figure 1

of resolution steps will (1) always terminate and (2) always reach the same normal form.

In Figure 1 we see two tables. The table on the left side of the figure has entries that can be further resolved. This table, when normalized, yields the table on the right side of the figure. Let us consider constructing a set of rewrite rules capable of normalizing the table shown on the left of Figure 1. The most direct solution would be obtained by simply creating the set of labeled rewrite rules shown in Figure 2.

An interesting characteristic of the rules given in Figure 2 is that they do not contain any variables. We will refer to rules that do not contain variables as *ground rules*. We will also use the term *resolution-set* to refer to a set of ground rewrite rules capable of normalizing a given table. For example, the ground rules in Figure 2, when considered collectively, describe a *resolution-set* for the table given in Figure 1.

Note that if the application of the resolution-set in Figure 2 can be controlled by a strategy so that rules are only applied to data values (i.e., the second element of a tuple), then the rule set will **correctly** normalize the left table given in Figure 1. Constructing a strategy that restricts the application of the above rules in this fashion is straightforward. The specific details of this type of strategy are unimportant in the context of this discussion and are therefore omitted. What is important, however, is the basic (strategic) approach taken to solve the table normalization problem – namely the generation of a set of

ground rules specifically tailored to resolve a given table.

A drawback of the approach described thus far is that it is highly problem specific. A resolution-set must be explicitly constructed by hand for each table under consideration. Of course, a more generic strategy would be one that could normalize an arbitrary well-formed table. In this case, our strategic line of thinking could be captured most directly if our framework has the ability to *dynamically* construct resolution-sets. In the higher-order framework of *TL* this can be achieved.

#### 4.1.1 A Higher-Order Solution

We first explore the dynamic construction of resolution-sets in a functional setting and then consider a generalization to a strategic framework. In a functional framework one could develop a higher-order function *make\_resolution\_set* that accepts a well-formed table  $t$  as its input parameter and produces a resolution-set  $rs$  as its output. Typically,  $rs$  would be a list of function values  $[r_1, r_2, \dots, r_n]$  where list elements are given in any order and where each function value  $r_i$  realizes a specific ground resolution rule as dictated/defined by  $t$ . Collectively, the list of functions in  $rs$  would be obtained by traversing the table  $t$ .

After constructing  $rs$ , a higher-order function such as *fold* could then be used together with an *apply* function  $\oplus$  enabling the rules in  $rs$  to be sequentially applied to  $t$ . The resulting expression would be:

$$foldl \oplus t \ rs \ \mathbf{where} \ \oplus : term * rule \rightarrow term$$

It should be noted that in order for *foldl* to have the desired effect, rule application should not result in failure. That is, given a rule  $r_i : lhs \rightarrow rhs$ , the rule application  $r_i(t)$  should either return  $t$  if  $r_i$  does not apply or a suitable instance of  $rhs$  if  $r_i$  does apply. If one makes this assumption about rules, then the fixed point of the sequential application resulting from *foldl* will yield the normal form of  $t$ . Since table normalization is confluent, the application of  $rs$  to the table  $t$  can proceed without giving much thought regarding the order in which rules are applied or even the order of the rules in the resolution-set list.

However, this approach encounters problems when dealing with non-confluent nonterminating systems. What we propose in this article is a way of dynamically constructing various types of strategies (e.g., strategies in which rules may be sequentially or conditionally composed and the order in which rules appear in the strategy is explicitly under the control of a traversal function). The resulting strategies can be seen as being similar to rule bases with the exception that the user has explicit control over their structure.

## 4.2 Generalization to a Higher-Order Strategic Framework

In the higher-order functional solution to the table normalization problem, a list  $rs = [r_1, r_2, \dots, r_n]$  of rules was created and applied to a term  $t$  using a fold operation on lists together with a function  $\oplus$  that applies rules to terms. In a strategic framework, such a computation sequence could most directly be described by the following strategy in which non-failure resulting rules  $r_i$  are sequentially composed.

$$rS_{\text{sequential}} = r_1; r_2; \dots; r_n$$

However, since the rule set is confluent and terminating, the computation could alternately be expressed as a strategy in which the rules  $r_i$  are composed using a nondeterministic choice combinator. This second strategy would be appropriate in a failure-based strategic framework.

$$rS_{\text{nondeterministic}} = r_1 + r_2 + \dots + r_n$$

Here the application of  $r_1 + r_2 + \dots + r_n$  to a term  $t$  will nondeterministically select an  $r_i$  which can be successfully applied to  $t$ . One could also express the rule set as strategy in which the rules  $r_i$  are composed using deterministic left-biased choice.

$$rS_{\text{deterministic}} = r_1 <+ r_2 <+ \dots <+ r_n$$

Here the application of  $r_1 <+ r_2 <+ \dots <+ r_n$  to a term  $t$  will select the leftmost  $r_i$  which can be successfully applied to  $t$ . The point here is that there is a design space for strategy construction.

Though interesting, the utility obtained by providing a strategic framework with the ability to dynamically construct strategies like the variations of  $rs$  shown above is somewhat limited. The reason for this is that strategic systems typically deal with non-confluent non-terminating systems and dynamic aggregations like  $rs$  oftentimes do not provide enough control over the application of the rules contained within them. However, such dynamically constructed aggregations become significantly more interesting if one can exercise just a little more control over their composition. In particular, suppose that the following is permitted:

- (1) The full power of traversal (e.g., bottom-up left-to-right, top-down right-to-left, or a selective traversal, etc.) may be used to construct the aggregation.
- (2) It is possible to specify which binary combinator (e.g., a sequential composition, deterministic choice, user defined) should be used to compose the individual strategies in the aggregation.

- (3) One can uniformly apply a strategic combinator to each rule in the aggregation. In particular, we introduce a combinator called *transient* that restricts any strategy  $s$  to which it is applied so that  $s$  can only be applied at most once to a term during the lifetime of the strategy.

At first glance, the proposed extensions may not appear significant. However, the control provided by the *transient* combinator in this framework should not be underestimated (see Section 6).

As a conclusion of this section, we present the *TL* solution to the table normalization problem. This solution introduces two constructs that have not been seen in previous examples. Typically, meta-programming systems such as strategic programming systems define term structures

First, in contrast to most strategic programming systems, *TL* represents terms as parse trees and not as abstract syntax trees (see Section 5.1). This shift is accompanied by a corresponding change in syntax. For example, where a system like Stratego would write  $entry(int, d)$  in *TL* this would be expressed as  $entry[[ (int, d) ]]$ . Second, the first-order traversal  $fix\_tdl(s)$  is introduced that will exhaustively apply the strategy  $s$  in a *tdl* fashion.

---

### Implementation in TL

---


$$resolution\_step : entry[[ (int_1, d_1) ]] \rightarrow entry[[ (int_2, int_1) ]] \rightarrow entry[[ (int_2, d_1) ]]$$

$$normalize : t \rightarrow fix\_tdl(rcond\_tdl\ resolution\_step\ t)\ t$$


---

Here the second-order strategy *resolution\_step* captures the general notion of a resolution step. The strategic expression  $(rcond\_tdl\ resolution\_step\ t)$  will traverse the table  $t$  in a *tdl* fashion constructing a *resolution-set* for  $t$ . This resolution-set is then applied to  $t$  in a fixed point fashion using *fix\_tdl*.

## 5 The Syntax and Semantics of *TL*

In this section we formally define the strategic programming language *TL*. Our focus is primarily on semantic and theoretical considerations and not necessarily on practical concerns such as the efficient implementation of constructs.

### 5.1 Trees

We are interested in the manipulation of terms corresponding to derivation sequences defined with respect to a given context-free grammar. Let  $G =$

$E$	$::=$	$E + T \mid T$
$T$	$::=$	$T * F \mid F$
$F$	$::=$	$int$

Fig. 3. A BNF describing a restricted set of mathematical expressions

$(N, T, P, S)$  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  $\alpha = X_1X_2\dots X_m$  where for all  $1 \leq i \leq m : X_i \in N \cup T$ , we say  $B$  derives  $\alpha$  iff the productions in  $P$  can be used to expand  $B$  to  $\alpha$ . Traditionally, the expression  $B \xrightarrow{*} \alpha$  is used to denote that  $B$  can derive  $\alpha$  in zero or more expansion steps. Similarly, one can write  $B \xrightarrow{\pm} \alpha$  to denote a derivation consisting of one or more expansion steps.

In our strategic framework, we write  $B[[\alpha']]$  to denote an *instance* of the derivation  $B \xrightarrow{\pm} \alpha$  whose resulting value is a parse tree having  $B$  as its *dominating symbol*. We refer to expressions of the form  $B[[\alpha']]$  as *parse expressions*. In the parse expression  $B[[\alpha']]$  the string  $\alpha'$  is an *instance* of  $\alpha$  because nonterminal symbols in  $\alpha'$  are constrained through the use of subscripts. We call subscripted nonterminal symbols *schema variables* or simply *variables* for short. We also consider a schema variable (e.g.,  $B_i$ ) to be a parse expression in its own right. An important thing to note about schema variables is that they are typed variables and as such many only be bound to parse trees resulting from proper derivations obtained from corresponding nonterminal symbols.

Within a given scope all occurrences of schema variables having the same subscript denote the same variable. The purpose of 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  $B$ .

Consider a BNF grammar shown in Figure 3 describing a restricted set of mathematical expressions. Given this grammar, the parse expression  $E[[T_1 + T_1]]$  denotes the set of all mathematical expressions  $e$  where  $e$  contains a single occurrence of the terminal symbol  $+$  and where the expressions on the left and right-hand side of the  $+$  operator are syntactically equal. Contrast this to the syntactic category  $[[T + T]]$  which imposes no such equality constraint on the derivations associated with either occurrence of  $T$ . In practice, equality constraints can easily be removed from a parse expression by requiring that all schema variables have unique subscripts. For example, the parse expression  $E[[T_1 + T_2]]$  is equivalent to the syntactic category  $[[T + T]]$ .

$\sigma(e_1 \wedge e_2)$	$\stackrel{def}{=}$	$\sigma(e_1) \wedge \sigma(e_2)$
$\sigma(e_1 \vee e_2)$	$\stackrel{def}{=}$	$\sigma(e_1) \vee \sigma(e_2)$
$\sigma(\neg e_1)$	$\stackrel{def}{=}$	$\neg(\sigma(e_1))$
$\sigma(t_1 \ll t_2)$	$\stackrel{def}{=}$	$\sigma(t_1) = t_2$

Fig. 4. The semantics of sigma distribution

When the dominating symbol and specific structure of a parse expression is unimportant the parse expression will be denoted by variables of the form  $t, t_1, \dots$  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.

## 5.2 Match Equations

Matching is a fundamental operation in our framework. We will use the symbol  $\ll$  adapted from the  $\rho$ -calculus [6] 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 substitution  $\sigma$  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  $\sigma$  is a solution to a match expression  $m$  iff  $\sigma(m)$  evaluates to true using the standard semantics for boolean operators in conjunction with the semantics defined in Figure 4.

## 5.3 Conditional Rewrite Rules

We assume a first-order conditional rewrite rule to be a *scoped directed equality* having the form:

$$lhs \rightarrow rhs \text{ if } E$$

where  $lhs$  and  $rhs$  are parse expressions and  $E$  is a *match expression*. In this framework,  $E$  plays a role similar to the *local evaluation* construct found in ELAN [3] and the *where* construct found in Stratego [32]. We restrict the free (schema) variables in  $rhs$  to be a subset of the free (schema) variables occurring in  $lhs$  and  $E$ .

The directed equality  $lhs \rightarrow rhs$  if  $E$  is *scoped* because in this context (and not beyond), identical variables must be bound to the same value.

For notational convenience, we will use the term  $lhs'$  to denote the portion of a conditional rewrite rule consisting of  $lhs$  together with the condition  $E$ . More specifically one can think of  $lhs'$  as a tuple of the form  $(lhs, E)$ . Thus we will write  $lhs' \rightarrow rhs$  as a shorthand for  $lhs \rightarrow rhs$  if  $E$ .

#### 5.4 The Syntax of TL Strategies

In this section, we define a term language for strategic expressions. In the definition below, we use some of the combinators introduced in [22] with some slight modifications and we also add a few combinators of our own.

- (1) A parse expression  $A[[\alpha']]$  is a strategy of order 0. Conceptually, we distinguish a parse expression as a *trivial* or *constant* strategy. All other strategies are non-trivial.
- (2) Let  $s^n$ ,  $s_1^n$  and  $s_2^n$  denote strategies of order  $n$ , then:
  - (a)  $skip^n$  is a strategy of order  $n$ , provided  $n > 0$ .
  - (b)  $transient(s^n)$  is a strategy of order  $n$ , provided  $n > 0$ .
  - (c)  $I(s^n)$  is a strategy of order  $n$  for any  $n$  provided  $n > 0$ . The unary combinator  $I$  is essentially a no-op.
  - (d)  $lhs' \rightarrow s^n$  is a (non-trivial) strategy of order  $n + 1$ , provided  $n \geq 0$ .
  - (e) Sequential Composition:  $s_1^n; s_2^n$  is a strategy of order  $n$ , provided  $n > 0$ .
  - (f) Left-biased Choice:  $s_1^n <+ s_2^n$  is a strategy of order  $n$ , provided  $n > 0$ .
  - (g) Right-biased Choice:  $s_1^n +> s_2^n$  is a strategy of order  $n$ , provided  $n > 0$ .
  - (h) Fixed point application:  $fix(s^n)$  is a strategy of order  $n$ , provided  $n = 1$ .
  - (i) One-layer First-Order Generic Traversal Combinators:
    - (i)  $all\_thread(s^n)$  is a strategy provided  $n = 1$ .
    - (ii)  $all\_broadcast(s^n)$  is a strategy provided  $n = 1$ .
  - (j) One-layer Higher-Order Generic Traversal Combinators:
    - (i)  $all\_thread\_left(s^n, \tau, \oplus)$  is a strategy where  $\tau$  is a unary strategy combinator (e.g., transient or  $I$ ) and  $\oplus$  is a binary strategy combinator (e.g., sequential composition, conditional composition),

provided  $n > 1$ .

- (ii)  $all\_thread\_right(s^n, \tau, \oplus)$  is a strategy where  $\tau$  is a unary strategy combinator (e.g., transient or  $I$ ) and  $\oplus$  is a binary strategy combinator (e.g., sequential composition, conditional composition), provided  $n > 1$ .
  - (iii)  $all\_broadcast\_left(s^n, \tau, \oplus)$  is a strategy where  $\tau$  is a unary strategy combinator (e.g., transient or  $I$ ) and  $\oplus$  is a binary strategy combinator, provided  $n > 1$ .
  - (iv)  $all\_broadcast\_right(s^n, \tau, \oplus)$  is a strategy where  $\tau$  is a unary strategy combinator (e.g., transient or  $I$ ) and  $\oplus$  is a binary strategy combinator, provided  $n > 1$ .
- (3) No other expressions are strategies.

### 5.5 The Semantics of TL

In the previous sections we have defined the following:

- (1) The syntax and semantics of match equations and match expressions.
- (2) The syntax of conditional first-order rewrite rules.
- (3) The syntax of strategic expressions in general.

We are now in a position to define what it means to apply a strategy to a term. We will do this in two stages. First we define the semantics of the conditional rewrite rule application in an identity-based manner. Then we define the semantics of strategy application in general (i.e., the application of composite strategies), including the application of higher-order strategies to terms.

#### 5.5.1 Basis: The Application of Conditional Rewrite Rules

The application of a conditional rewrite rule  $r$  to a tree  $t$  is expressed as  $r(t)$  where  $r$  is either an abstraction of a rewrite rule (i.e., a name) 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  $r(t)$ .

Let us consider the application  $(lhs' \rightarrow s^n) t$  where  $lhs'$  is  $(lhs, E)$ . We say that  $lhs' \rightarrow s^n$  *applies* to the term  $t$  if  $lhs \ll t \wedge E$  holds. The notion of “ $lhs \ll t \wedge E$  holds” is so central to our framework, that we define this concept explicitly.

**Definition 3**  $eval(lhs', t, \sigma)$  is a predicate that when given an  $lhs'$  whose value is  $(lhs, E)$  and a tree  $t$ , will evaluate to true iff  $\sigma(t \ll lhs \wedge E)$  evaluates to

*true*.

With this definition we are now in a position to define the application of a higher-order conditional rewrite rule.

$$\textbf{Definition 4} \quad (lhs' \rightarrow s^n) t = \begin{cases} \sigma(s^n) & \text{if } \exists \sigma : eval(lhs', t, \sigma) \\ t & \text{if } \neg \exists \sigma : eval(lhs', t, \sigma) \wedge n = 0 \\ skip^n & \text{if } \neg \exists \sigma : eval(lhs', t, \sigma) \wedge n > 0 \end{cases}$$

It should be noted that the definition given for rule application is identity-based. This means that if a first-order rule fails to apply to a term  $t$  then  $t$  will be returned unchanged. If a higher-order rule fails to apply then  $skip^n$  will be returned where  $n$  is the order of the right-hand side of the rule. This is in sharp contrast to systems such as Stratego, ELAN, and the  $S'_\gamma$  calculus where the failure of a rule to apply will yield a distinguished value *fail*.

Furthermore, even though  $TL$  is a higher-order language, alpha-conversion, as it is defined in the lambda-calculus, is not required. All schema variables within a higher-order strategy fall within a single scope and must be (statically) distinguished accordingly within the definition. When applying strategies, the name capture problem is avoided 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.

### 5.5.2 Choice and Observing the Application of a Strategy

The notion of choosing a strategy from a collection of strategies is central to any strategic programming framework. ELAN [3] provides the operators *dc* and *dk* which respectively denote *don't care choose* and *don't know choose* and enables strategies to be created in which the choice of which strategy to apply is left unspecified. A biased choice combinator is also common in the literature. Stratego [37][37] defines a left-biased choice operator in an operational fashion. The strategy  $s_1 <+ s_2$  will first try to apply  $s_1$  and if that fails, the application of the strategy  $s_2$  is attempted. The  $S'_\gamma$  calculus [21], defines biased choice in terms of a non-deterministic choice combinator, a negation-by-failure combinator, and a sequential composition combinator. For example, let the expression  $s_1 + s_2$  denote a strategy that will non-deterministically apply either  $s_1$  or  $s_2$ . Let  $s_1; s_2$  denote the sequential composition of  $s_1$  and  $s_2$  (apply  $s_1$  followed by  $s_2$ ), and let  $\neg s_1$  denote a strategy that succeeds if and only if  $s_1$  fails. Given these combinators, left-biased choice (first try  $s_1$  and if that fails try  $s_2$ ) and right-biased choice (first try  $s_2$  and if that fails try  $s_1$ )

can be defined as follows<sup>2</sup>:

$$s_1 <+s_2 \stackrel{def}{=} s_1 + (\neg s_1; s_2)$$

$$s_1 +> s_2 \stackrel{def}{=} (\neg s_2; s_1) + s_2$$

An issue that every strategic framework supporting a choice combinator must address is how to “observe” when a strategy has been successfully applied. Such an observation is essential in order to effectively navigate strategies involving choice combinators. In a failure-based framework, the implementation of such an observation is straightforward since the value *fail* explicitly indicates when a rule application has failed. However, in an identity-based framework such as ours, the implementation of observation becomes a bit more involved. One way to solve the problem is to implement an observer predicate  $observe(s, t)$  that evaluates to *true* if and only if the strategy  $s$  applies to the term  $t$ . Note that in addition to being computationally expensive, simply performing an equality comparison on the terms  $t$  and  $s(t)$  is not correct (e.g., if  $t \neq s(t)$  then  $observe(s, t)$  is true otherwise it is false). In particular, such a test would not be able to distinguish between the failure or success of applications involving identity-like rules (e.g., the application of  $(b \rightarrow b)$  to the term  $b$ ).

In our framework, the presence of the transient combinator requires the notion of observation to be further refined. The nature of this refinement is dependent upon the semantics given to the transient combinator. There are several possible definitions from which one could choose. Informally stated, we have chosen to define the transient combinator in the following way:

Given a strategic expression of the form  $transient(s)$  we refer to  $s$  as the *contents* of the transient. A transient is a strategic combinator that restricts the application of its *contents* so that it may be applied **at most once**. The only exception to this rule is that a transient **may not** observe the application of the contents of another (nested) transient.

The “at most once” property characterizes the *transient* combinator and motivates the introduction of *skip* into our strategic framework. We define *skip* as a strategy whose application never succeeds but whose application nevertheless does not yield a failure value. For example, in a first-order setting, *skip* will always leave any term it is applied to unchanged.

Figure 5 gives some relationships between two abstract strategic constants  $\epsilon$  and  $\delta$  and the combinators  $<+$  and  $;$ . These relationships are considered from the perspective of a failure-based framework as well as an identity-based

---

<sup>2</sup> From the perspective of implementation, it is more efficient for a strategic system to directly support the right and left choice operators as primitives of the system.

Strategy	Failure-Based Semantics	Identity-based Semantics
$\epsilon \lt+ s$	$\epsilon$	$\epsilon$
$s \lt+ \epsilon$	$s \lt+ \epsilon$	$s$
$\delta \lt+ s$	$s$	$s$
$s \lt+ \delta$	$s$	$s$
$\epsilon ; s$	$s$	$s$
$s ; \epsilon$	$s$	$s$
$\delta ; s$	$\delta$	$s$
$s ; \delta$	$\delta$	$s$

Fig. 5. The semantics of *id*, *skip*, and *fail*

framework. In failure-based systems such as Stratego and ELAN,  $\epsilon$  is typically called *id* or *identity* and  $\delta$  is typically called *fail*. In the identity-based framework of TL,  $\epsilon$  is called *id* and  $\delta$  is called *skip*.

Operationally, we define a strategy of the form *transient*(*s*) as a strategy that reduces to the strategy *skip* if the application of its contents (i.e., the strategy *s*) can be observed.

From an implementation perspective, the definition of *transient* introduces the need for two distinct internally maintained observer predicates. The first predicate *observe<sub>choice</sub>* defines the semantics of applies from the perspective of the choice combinator. The second predicate *observe<sub>transient</sub>* defines the semantics of applies from the perspective of the transient combinator. The following example illustrates the difference between the observer predicates:

$$\mathit{transient}(\mathit{transient}(s_1) \lt+ s_2) t$$

For the purposes of this discussion, let us assume that  $s_1$  and  $s_2$  are first-order strategies and that  $s_1$  can be applied to the term  $t$  yielding  $t'$ . The choice combinator  $\lt+$  must be able to observe that  $s_1$  has been successfully applied to  $t$  in order to prevent an attempt to apply  $s_2$  to  $t$ . In addition, we would like the successful application of  $s_1$  to  $t$  to reduce  $\mathit{transient}(\mathit{transient}(s_1) \lt+ s_2)$  to  $\mathit{transient}(\mathit{skip} \lt+ s_2)$  which can be reduced to  $\mathit{transient}(s_2)$ . In particular, we do not want to permit the observation of the successful application of  $s_1$  to  $t$  to reach the outermost transient, since then the entire strategy  $\mathit{transient}(\mathit{transient}(s_1) \lt+ s_2)$  would reduce to *skip*. Therefore, in order to prevent the cascading effect described, it is essential that the outermost *transient* not be permitted to observe the application of  $s_1$  to  $t$ .

$observe_X(skip^n, t)$	$\stackrel{def}{=} false$
$observe_X(id, t)$	$\stackrel{def}{=} true$
$observe_{choice}(transient(s^n), t)$	$\stackrel{def}{=} observe_{choice}(s^n, t)$
$observe_{transient}(transient(s^n), t)$	$\stackrel{def}{=} false$
$observe_X(lhs' \rightarrow s^n, t)$	$\stackrel{def}{=} \exists \sigma : eval(lhs', t, \sigma)$
$observe_X(s_1^n; s_2^n, t)$	$\stackrel{def}{=} observe_X(s_1^n, t) \vee observe_X(s_2^n, t)$
$observe_X(s_1^n <+ s_2^n, t)$	$\stackrel{def}{=} observe_X(s_1^n, t) \vee observe_X(s_2^n, t)$
$observe_X(s_1^n +> s_2^n, t)$	$\stackrel{def}{=} observe_X(s_1^n, t) \vee observe_X(s_2^n, t)$
$observe_X(fix(s^1), t)$	$\stackrel{def}{=} observe_X(s^1, t)$
$observe_X(all\_thread\_left(s^1), tree)$	$\stackrel{def}{=} \bigvee_{i=1}^m observe_X(s^1, t_i)$ where $tree = t(t_1, t_2, \dots, t_m)$
$observe_X(all\_thread\_right(s^1), tree)$	$\stackrel{def}{=} \bigvee_{i=1}^m observe_X(s^1, t_i)$ where $tree = t(t_1, t_2, \dots, t_m)$
$observe_X(all\_thread\_left(s^n, \tau, \oplus), tree)$	$\stackrel{def}{=} \bigvee_{i=1}^m observe_X(s^n, t_i)$ where $tree = t(t_1, t_2, \dots, t_m)$
$observe_X(all\_thread\_right(s^n, \tau, \oplus), tree)$	$\stackrel{def}{=} \bigvee_{i=1}^m observe_X(s^n, t_i)$ where $tree = t(t_1, t_2, \dots, t_m)$
$observe_X(all\_broadcast\_left(s^1), tree)$	$\stackrel{def}{=} \bigvee_{i=1}^m observe_X(s^1, t_i)$ where $tree = t(t_1, t_2, \dots, t_m)$
$observe_X(all\_broadcast\_right(s^1), tree)$	$\stackrel{def}{=} \bigvee_{i=1}^m observe_X(s^1, t_i)$ where $tree = t(t_1, t_2, \dots, t_m)$
$observe_X(all\_broadcast\_left(s^n, \tau, \oplus), tree)$	$\stackrel{def}{=} \bigvee_{i=1}^m observe_X(s^n, t_i)$ where $tree = t(t_1, t_2, \dots, t_m)$
$observe_X(all\_broadcast\_right(s^n, \tau, \oplus), tree)$	$\stackrel{def}{=} \bigvee_{i=1}^m observe_X(s^n, t_i)$ where $tree = t(t_1, t_2, \dots, t_m)$

Fig. 6. The semantics of observation

Operationally, it turns out that the definitions of  $observe_{choice}$  and  $observe_{transient}$  are identical for all strategic combinators except for the definition of the *transient* combinator. Thus in order to avoid duplication, Figure 6 formally presents the semantics of a single predicate called  $observe_X$  rather than presenting separate tables for the predicates  $observe_{choice}$  and  $observe_{transient}$ . As a result, the definition of  $observe_X$  can be viewed as being somewhat overloaded. The definition of  $observe_{choice}$  can be obtained (i.e., extracted) from the definition of  $observe_X$  by instantiating the subscript  $X$  with the value *choice*. Similarly, the definition of  $observe_{transient}$  can be obtained by instantiating the subscript  $X$  with the value *transient*.

Note that in an identity-based framework it is sufficient to conclude that a strategy has applied if **at least** one of its sub-strategies applies. For example, in order to conclude that  $s_1; s_2; \dots; s_n$  applies to  $t$ , it is sufficient to find a single

strategy  $s_i$  that applies to  $t$ .

### 5.5.3 The Semantics of Basic Strategic Combinators

In  $TL$ , strategies are subject to the following restrictions:

- (1) All strategies must be homogeneous with respect to order. That is, an order  $n$  strategy may not have components that are order  $m$  where  $m \neq n$ .
- (2) The application of a strategy is restricted to ground terms (i.e., order 0 strategies).

In the semantic definitions below a distinction is made between first-order strategies and higher-order strategies. This is done for the following reason: For  $n > 1$ , the strategy  $skip^n$  will return  $skip^{n-1}$  when applied to a tree. In contrast,  $skip^1$  will return  $t$  when applied to  $t$ .

In a higher-order framework without the transient combinator the application  $s^n t = s^{n-1}$ . When  $n = 1$  we have the degenerate case where  $s^{n-1} = s^0 = t'$  is a ground term. Introducing the *transient* combinator into this framework enables the application of a strategy to a term to change both the strategy as well as the term. For example, consider a first-order strategy  $s^1$  containing no transients. Suppose that  $s^1$  can successfully be applied to  $t$  yielding  $t'$ . The application  $transient(s^1)t$  will yield the strategy  $skip^1$  as well as the term  $t'$ . From an operational perspective this tuple  $(skip^1, t')$  denotes the result obtained from the evaluation of  $transient(s^1)t$ . In general then, the application of a strategy  $s^n$  to a term  $t$  will yield a tuple of the form  $(\hat{s}^n, s^{n-1})$  where  $\hat{s}^n$  is the strategy resulting from the application and  $s^{n-1}$  is the result of the application. In the higher-order case (i.e., when  $n > 1$ ) the *result of the application* will be a strategy. In the first-order case (i.e., when  $n = 1$ ) the *result of the application* will be a term.

We would like to mention that the tuples resulting from the application of a strategy to a term are a semantic artifact and represent a level of detail that must be dealt with in an implementation of  $TL$ . Under normal circumstances the typical  $TL$  programmer will not have access to nor be required to manipulate such tuples. As we have seen in the example  $TL$  programs so far, the application of a strategy to a term in the typical  $TL$  program appears similar to any other strategic programming framework.

$$skip^n t \stackrel{def}{=} \begin{cases} (skip^1, t) & \text{if } n = 1 \\ (skip^n, skip^{n-1}) & \text{if } n > 1 \end{cases}$$

$$\begin{aligned}
(lhs' \rightarrow s^n) t &\stackrel{def}{=} \begin{cases} ((lhs' \rightarrow s^n), \sigma(s^n)) & \text{if } observe_{choice}(lhs', t) \wedge eval(lhs', t, \sigma) \\ ((lhs' \rightarrow s^n), t) & \text{if } \neg observe_{choice}(lhs', t) \wedge n = 0 \\ ((lhs' \rightarrow s^n), skip^n) & \text{if } \neg observe_{choice}(lhs', t) \wedge n > 0 \end{cases} \\
transient(s^n) t &\stackrel{def}{=} \begin{cases} (skip^n, s^n t) & \text{if } observe_{transient}(s^n, t) \\ (\hat{s}^n, s^{n-1}) & \text{if } \neg observe_{transient}(s^n, t) \wedge (\hat{s}^n, s^{n-1}) = s^n t \end{cases} \\
I(s^n) \quad t &\stackrel{def}{=} \begin{cases} s^n t \end{cases} \\
(s_1^n; s_2^n) t &\stackrel{def}{=} \begin{cases} ((\hat{s}_1^1; \hat{s}_2^1), t'') & \text{if } n = 1 \wedge (\hat{s}_1^1, t') = s_1^n t \wedge (\hat{s}_2^1, t'') = s_2^n t' \\ ((\hat{s}_1^n; \hat{s}_2^n), (s_1^{n-1}; s_2^{n-1})) & \text{if } n > 1 \wedge (\hat{s}_1^n, s_1^{n-1}) = s_1^n t \wedge (\hat{s}_2^n, s_2^{n-1}) = s_2^n t \end{cases} \\
(s_1^n <+ s_2^n) t &\stackrel{def}{=} \begin{cases} ((\hat{s}_1^n <+ s_2^n), s^{n-1}) & \text{if } observe_{choice}(s_1^n, t) \wedge (\hat{s}_1^n, s^{n-1}) = s_1^n t \\ ((s_1^n <+ \hat{s}_2^n), s^{n-1}) & \text{if } \neg observe_{choice}(s_1^n, t) \wedge (\hat{s}_2^n, s^{n-1}) = s_2^n t \end{cases} \\
(s_1^n >+ s_2^n) t &\stackrel{def}{=} \begin{cases} ((s_1^n >+ \hat{s}_2^n), s^{n-1}) & \text{if } observe_{choice}(s_2^n, t) \wedge (\hat{s}_2^n, s^{n-1}) = s_2^n t \\ ((\hat{s}_1^n >+ s_2^n), s^{n-1}) & \text{if } \neg observe_{choice}(s_2^n, t) \wedge (\hat{s}_1^n, s^{n-1}) = s_1^n t \end{cases} \\
fix(s_0^1) t &\stackrel{def}{=} \begin{cases} (\hat{s}^1, t'') & \text{if } (observe_{transient}(s_0^1, t) \vee observe_{choice}(s_0^1, t)) \\ & \wedge (s_0^1, t') = (s_0^1 t) \wedge (\hat{s}^1, t'') = fix(s_0^1) t' \\ (s_0^1, t) & \text{if } \neg (observe_{transient}(s_0^1, t) \vee observe_{choice}(s_0^1, t)) \end{cases}
\end{aligned}$$

#### 5.5.4 The Semantics of First-Order Generic Traversal Combinators

The ability to control term traversal is central to strategic programming frameworks. Three approaches for specifying term traversals are possible: manual,

fixed, and user defined. In a manual approach, recursive rules need to be written to account for every term constructor that may be encountered during the traversal. This form of traversal construction is supported by rewriting systems in general and by ELAN [12] in particular. A second approach is to provide a fixed set of generic traversals (e.g., top-down, bottom-up, etc.) which can then be used to define various rewriting strategies. This approach has been taken in an extension to ASF+SDF [5].

In the third approach, a set of primitive generic *one-layer* traversal combinators are provided by the language from which the user may construct custom traversals. A *one-layer* traversal is a combinator that applies a given strategy to a subset of the immediate children of a term – and goes no further. One-layer traversals can be used in recursive equations to describe a number of useful strategies capable of traversing entire term structures as well as selective portions of terms. This is the approach taken by Stratego [33], the  $S'_\gamma$  calculus [21], and *TL*.

Recall that the application of a *TL* strategy to a term can change both the term as well as the strategy. This raises some questions regarding how strategies should be applied within traversals. Two possibilities come to mind: *threading* and *broadcasting*. In threading, when a strategy  $s$  is applied to a term  $t$ , the resulting strategy  $s'$  becomes the strategy that is applied to the next term encountered in the traversal, and so on. Threading creates a need to distinguish between left-to-right and right-to-left traversals. Broadcasting on the other hand, involves making copies of the strategy under consideration and is insensitive to left/right traversal orientation. As a result, *TL* provides three basic first-order generic combinators: *all\_thread\_left*, *all\_thread\_right*, and *all\_broadcast*. As the name suggests these primitives are variations of the generic traversal combinator *all* which arises in various guises in the literature. Informally stated,  $all(s)t$  applies the strategy  $s$  to all of the immediate subterms (i.e., the children) of  $t$ .

$$\begin{aligned}
all\_thread\_left(s_0^1) tree &\stackrel{def}{=} (s_m^1, tree') \text{ where } tree = t(t_1, t_2, \dots, t_m) \\
&\text{and } (s_1^1, t'_1) = s_0^1 t_1 \\
&\quad (s_2^1, t'_2) = s_1^1 t_2 \\
&\quad \dots \\
&\quad (s_m^1, t'_m) = s_{m-1}^1 t_m \\
&\text{and } tree' = t(t'_1, t'_2, \dots, t'_m)
\end{aligned}$$

$$\begin{aligned}
\text{all\_thread\_right}(s_0^1) \text{ tree} &\stackrel{\text{def}}{=} (s_m^1, \text{tree}') \text{ where } \text{tree} = t(t_1, t_2, \dots, t_m) \\
&\text{and } (s_1^1, t'_m) = s_0^1 t_m \\
&\quad (s_2^1, t'_{m-1}) = s_1^1 t_{m-1} \\
&\quad \dots \\
&\quad (s_m^1, t'_1) = s_{m-1}^1 t_1 \\
&\text{and } \text{tree}' = t(t'_1, t'_2, \dots, t'_m)
\end{aligned}$$

$$\begin{aligned}
\text{all\_broadcast}(s^1) \text{ tree} &\stackrel{\text{def}}{=} (s^1, \text{tree}') \text{ where } \text{tree} = t(t_1, t_2, \dots, t_m) \\
&\text{and } (s_1^1, t'_1) = s^1 t_1 \\
&\quad (s_2^1, t'_2) = s^1 t_2 \\
&\quad \dots \\
&\quad (s_m^1, t'_m) = s^1 t_m \\
&\text{and } \text{tree}' = t(t'_1, t'_2, \dots, t'_m)
\end{aligned}$$

### 5.5.5 The Semantics of Higher-Order Generic Traversal Combinators

The higher-order generic one-layer traversals described here are unique to  $TL$ . Abstractly, they can be seen as a morphism from term structures to strategic structures (i.e., strategies). Given this perspective they can be seen as being similar but not identical to hylomorphisms over rose<sup>3</sup> trees found in functional programming frameworks [25][26]. The primary difference between our higher-order traversals and hylomorphisms is that in our framework, the strategy  $s_i^n$  is itself changing as it is being applied to the term  $t_i$  while in a hylomorphism  $s_i^n$  would need to remain the same.

In the definitions that follow the symbol  $\oplus$  denotes a binary infix strategic combinator such as  $;$ ,  $\langle + \rangle$ , or  $+ \rangle$ , and  $\tau$  denotes a unary combinator such as *transient* or  $I$ .

---

<sup>3</sup> A rose tree is a multi-way branching tree.

$$\begin{aligned}
all\_thread\_left(s_0^n, \tau, \oplus) \text{ tree} &\stackrel{def}{=} (s_m^n, s_m^{n-1}) \text{ where } tree = t(t_1, t_2, \dots, t_m) \\
&\text{and } (s_1^n, s_1^{n-1}) = s_0^n t_1 \\
&\quad (s_2^n, s_2^{n-1}) = s_1^n t_2 \\
&\quad \dots \\
&\quad (s_m^n, s_m^{n-1}) = s_{m-1}^n t_m \\
&\text{and} \\
s^{n-1} &= \tau(s_1^{n-1}) \oplus \tau(s_2^{n-1}) \oplus \dots \oplus \tau(s_m^{n-1})
\end{aligned}$$

$$\begin{aligned}
all\_thread\_right(s_0^n, \tau, \oplus) \text{ tree} &\stackrel{def}{=} (s_m^n, s_m^{n-1}) \text{ where } tree = t(t_1, t_2, \dots, t_m) \\
&\text{and } (s_1^n, s_1^{n-1}) = s_0^n t_m \\
&\quad (s_2^n, s_2^{n-1}) = s_1^n t_{m-1} \\
&\quad \dots \\
&\quad (s_m^n, s_m^{n-1}) = s_{m-1}^n t_1 \\
&\text{and} \\
s^{n-1} &= \tau(s_1^{n-1}) \oplus \tau(s_2^{n-1}) \oplus \dots \oplus \tau(s_m^{n-1})
\end{aligned}$$

$$\begin{aligned}
all\_broadcast\_left(s^n, \tau, \oplus) \text{ tree} &\stackrel{def}{=} (s^n, s^{n-1}) \text{ where } tree = t(t_1, t_2, \dots, t_m) \\
&\text{and } (-, s_1^{n-1}) = s^n t_1 \\
&\quad (-, s_2^{n-1}) = s^n t_2 \\
&\quad \dots \\
&\quad (-, s_m^{n-1}) = s^n t_m \\
&\text{and} \\
s^{n-1} &= \tau(s_1^{n-1}) \oplus \tau(s_2^{n-1}) \oplus \dots \oplus \tau(s_m^{n-1})
\end{aligned}$$

$$\begin{aligned}
all\_broadcast\_right(s^n, \tau, \oplus) \text{ tree} &\stackrel{def}{=} (s^n, s^{n-1}) \text{ where } tree = t(t_1, t_2, \dots, t_m) \\
&\text{and } (-, s_1^{n-1}) = s^n t_m \\
&\quad (-, s_2^{n-1}) = s^n t_{m-1} \\
&\quad \dots \\
&\quad (-, s_m^{n-1}) = s^n t_1 \\
&\text{and} \\
s^{n-1} &= \tau(s_1^{n-1}) \oplus \tau(s_2^{n-1}) \oplus \dots \oplus \tau(s_m^{n-1})
\end{aligned}$$

definition	::=	non_recursive_def   recursive_def
non_recursive_def	::=	id : expression   id ( arg_list ) : expression
recursive_def	::=	id = body   id ( arg_list ) = body
body	::=	id   expression   $\lambda$ t. let binding_list in strategy_application end
binding_list	::=	binding binding list   binding
binding	::=	( id , id ) = strategy_application   definition
expression	::=	strategy_application   strategy_expression
strategy_application	::=	the application of a strategy to a term
strategy_expression	::=	defined in Section 5.4

Fig. 7. A partial BNF for strategy definitions

### 5.5.6 Coda

In the definitions above, we have glossed over some low-level details regarding type consistency. For example, the equations described in Section 5.2 may involve expressions in which strategies are applied to terms. Given the above definitions, a strategy application, will yield the atypical tuple rather than a single value which is standard in strategic frameworks. This problem can be resolved by extending the definition of match equations so they can handle tuples. For example,  $e \ll (s, t) \stackrel{def}{=} e \ll t$ . In practice, these issues do not pose a problem. Furthermore, the details are uninteresting with respect to the theme of this paper and are therefore not discussed in further detail.

### 5.5.7 Non-recursive and Recursive Strategy Definitions

*TL* makes a distinction between non-recursive and recursive strategy definitions. The colon and equality symbols are used as the mechanisms for defining non-recursive and recursive strategies respectively. A partial BNF syntax for strategy definitions is given in Figure 7.

In *TL*, non-recursive strategy definitions are called *labeled strategies*. Due to their non-recursive nature, labeled strategies are little more than syntactic sugar when seen from a semantic perspective. They provide a mechanism for

abstracting strategy expressions. Their purpose is to increase readability, and they can be statically removed through a fixed number of unfold operations. Because of this, labeled strategies do not enhance the capabilities of a system with respect to the *distributed data problem* as defined in Section 1.

On the other hand, combining parameter passing with recursive definitions enhances the capabilities of a system with respect to the distributed data problem. Recursive strategies have the ability to transport values to points arbitrarily deep within a term structure. Thus, when writing *TL* programs, we discourage the use of recursive definitions involving parameters except for defining strategies that are completely generic (e.g., top-down, bottom-up, etc.).

*TL* provides a library of predefined generic traversals that should be sufficient for most situations. However, an expert user has the option of defining their own library of traversals. Such users must be aware of the fact that the application of a strategy to a term yields a tuple. In Sections 5.5.8 and 5.5.9 the general purpose traversals for the *TL* library are defined. All of these definitions manipulate the tuple resulting from the application of a strategy to a term.

### 5.5.8 Some Generic First-Order *TL* Traversals

In this section we present some of the generic first-order traversals that form part of the *TL* strategy library.

```

tdl_thread(s01) = λ t . let
    (s11, t') = s01 t
in
    all_thread_left(tdl_thread(s11)) t'
end

```

When applied to a term *t*, the strategy *t**dl\_thread*(*s*<sub>0</sub><sup>1</sup>) will perform a top-down left-to-right traversal of *t* and apply the current value of *s*<sub>0</sub><sup>1</sup> to every term encountered. Note that every application of *s*<sub>0</sub><sup>1</sup> to a term in *t* may potentially alter *s*<sub>0</sub><sup>1</sup>.

```

tdr_thread( $s_0^1$ ) =  $\lambda t$  . let
    ( $s_1^1, t'$ ) =  $s_0^1 t$ 
  in
    all_thread_right(tdr_thread( $s_1^1$ ))  $t'$ 
  end

```

The strategy *tdr\_thread*( $s_0^1$ ) is similar to *tdl\_thread*( $s_0^1$ ). The only difference is that when applied to a term  $t$  the strategy *tdr\_thread*( $s_0^1$ ) will perform a top-down right-to-left traversal of  $t$ .

```

bul_thread( $s_0^1$ ) =  $\lambda t$  . let
    ( $s_1^1, t'$ ) = all_thread_left(bul_thread( $s_0^1$ ))  $t$ 
  in
     $s_1^1 t'$ 
  end

```

When applied to a term  $t$ , the strategy *bul\_thread*( $s_0^1$ ) will perform a bottom-up left-to-right traversal of  $t$  and apply the current value of  $s_0^1$  to every term encountered. Note that every application of  $s_0^1$  to a term in  $t$  may potentially alter  $s_0^1$ .

```

bur_thread( $s_0^1$ ) =  $\lambda t$  . let
    ( $s_1^1, t'$ ) = all_thread_right(bur_thread( $s_0^1$ ))  $t$ 
  in
     $s_1^1 t'$ 
  end

```

The strategy *bur\_thread*( $s_0^1$ ) is similar to *bul\_thread*( $s_0^1$ ). The only difference is that when applied to a term  $t$  the strategy *bur\_thread*( $s_0^1$ ) will perform a bottom-up right-to-left traversal of  $t$ .

```

td_broadcast( $s_0^1$ ) =  $\lambda t$  . let
    ( $s_1^1, t'$ ) =  $s_0^1 t$ 
  in
    all_broadcast(td_broadcast( $s_1^1$ ))  $t'$ 
  end

```

When applied to a term  $t$ , the strategy *td\_broadcast*( $s_0^1$ ) will traverse  $t$  in a

top-down breadth-first fashion. If a term  $t_i$  causes  $s_0^1$  to be changed to  $s_1^1$ , then  $s_1^1$  will be uniformly applied to all the children of  $t_i$ .

In addition to the generic traversals defined here, traversals involving the fixed point application of a strategy to a term are also useful (e.g., *fix\_tdl*). Informally speaking, the most reasonable semantics for the evaluation of a strategic expression like *fix\_tdl*( $s$ ) $t$  would be to continue to traverse and apply the current value of the strategy  $s$  until an entire traversal can be performed in which neither  $s$  nor  $t$  change. Such a semantics requires access to the observe predicate and can be realized by an implementer of *TL*. However, traversals based on *fix* cannot be defined at the user-level since the observe predicate is not available to them.

### 5.5.9 Some Generic Higher-Order Traversals

In this section, we present some of the higher-order generic traversals that form part of the *TL* strategy library. There are essentially three degrees of freedom in the traversal strategies presented. First, there are four possible traversals (1) top-down left-to-right, (2) top-down right-to-left, (3) bottom-up left-to-right, and (4) bottom-up right-to-left. Three possibilities for binary combinators are considered (1) sequential composition, (2) left-biased choice, and (3) right-biased choice. And finally, two unary combinators are considered (1) *I*, and (2) *transient*.

Below we give the definitions of several strategies. We leave it to the reader to construct the remaining strategic variations.

```

seq_tdl( $s_0^n$ ) =  $\lambda t$ .  let
    ( $s_1^n, s^{n-1}$ ) =  $s_0^n t$ 
in
     $s^{n-1}$  ; (all_thread_left(seq_tdl( $s_1^n$ ), I, ; )  $t$ )
end

```

When applied to a term  $t$ , the strategy *seq\_tdl*( $s_0^n$ ) will traverse  $t$  in a top-down left-to-right fashion applying the current instance of  $s_0^n$  to every term encountered. The results of these applications will then be sequentially composed to form a strategy of order  $n - 1$ .

```

seq_bul( $s_0^n$ ) =  $\lambda t.$  let
    ( $s_1^n, s_1^{n-1}$ ) = all_thread_left(seq_bul( $s_0^n$ ),  $I, ;$ )  $t$ 
    ( $s_2^n, s_2^{n-1}$ ) =  $s_1^n t$ 
in
     $s_1^{n-1} ; s_2^{n-1}$ 
end

```

The strategy  $\textit{seq\_bul}(s_0^n)$  is similar to  $\textit{seq\_tdl}(s_0^n)$ . The only difference is that when applied to a term  $t$ ,  $\textit{seq\_bul}(s_0^n)$  will traverse  $t$  in a bottom-up left-to-right fashion.

```

lcond_tdl( $s_0^n$ ) =  $\lambda t.$  let
    ( $s_1^n, s_1^{n-1}$ ) =  $s_0^n t$ 
in
     $s_1^{n-1} <+ (\textit{all\_thread\_left}(\textit{lcond\_tdl}(s_1^n), I, <+) t)$ 
end

```

When applied to a term  $t$ , the strategy  $\textit{lcond\_tdl}(s_0^n)$  will traverse  $t$  in a top-down left-to-right fashion an applying the current instance of  $s_0^n$  to every term encountered. The results of these applications will then be composed using the left-biased choice combinator  $<+$  to form a strategy of order  $n - 1$ .

```

rcond_tdl( $s_0^n$ ) =  $\lambda t.$  let
    ( $s_1^n, s_1^{n-1}$ ) =  $s_0^n t$ 
in
     $s_1^{n-1} <+ (\textit{all\_thread\_left}(\textit{rcond\_tdl}(s_1^n), I, <+) t)$ 
end

```

The strategy  $\textit{rcond\_tdl}(s_0^n)$  is similar to  $\textit{lcond\_tdl}(s_0^n)$ . The only difference is that when applied to a term  $t$ ,  $\textit{rcond\_tdl}(s_0^n)$  will traverse  $t$  in a top-down right-to-left fashion and compose the results using the right-biased choice combinator  $+>$  to form a strategy of order  $n - 1$ .

```

lcond_bul( $s_0^n$ ) =  $\lambda t$ .  let
    ( $s_1^n, s_1^{n-1}$ ) = all_thread_left(lcond_bul( $s_0^n$ ),  $I, <+$ )  $t$ 
    ( $s_2^n, s_2^{n-1}$ ) =  $s_1^n t$ 
in
     $s_1^{n-1} <+ s_2^{n-1}$ 
end

```

The strategy  $lcond\_bul(s_0^n)$  is similar to  $lcond\_tdl(s_0^n)$ . The only difference is that when applied to a term  $t$ ,  $lcond\_bul(s_0^n)$  will traverse  $t$  in a bottom-up left-to-right fashion.

```

rcond_bul( $s_0^n$ ) =  $\lambda t$ .  let
    ( $s_1^n, s_1^{n-1}$ ) = all_thread_right(rcond_bul( $s_0^n$ ),  $I, <+$ )  $t$ 
    ( $s_2^n, s_2^{n-1}$ ) =  $s_1^n t$ 
in
     $s_1^{n-1} <+ s_2^{n-1}$ 
end

```

The strategy  $rcond\_bul(s_0^n)$  is similar to  $rcond\_tdl(s_0^n)$ . The only difference is that when applied to a term  $t$ ,  $rcond\_bul(s_0^n)$  will traverse  $t$  in a bottom-up right-to-left fashion.

#### 5.5.10 Some Standard First-Order Generic Strategies in TL

In the literature there are a number of generic traversals that are widely recognized. In this section, we provide some definitions of these standard traversals in terms of the traversals we have defined thus far. In the definitions given below, we assume that  $s^1$  is a first-order strategy that is transient-free (i.e., contains no transient combinators).

Strategy	Comment
$TD(s^1)$ : $tdl\_thread(s^1)$	Apply $s^1$ top-down left-to-right
$BU(s^1)$ : $bul\_thread(s^1)$	Apply $s^1$ bottom-up left-to-right
$onceTD(s^1)$ : $TD(transient(s^1))$	Apply $s^1$ at most once
$onceBU(s^1)$ : $BU(transient(s^1))$	Apply $s^1$ at most once
$stopTD(s^1)$ : $tdl\_broadcast(transient(s^1))$	Apply $s^1$ at most once on any path from a subterm to its root.

From a theoretical perspective, the application of the strategy  $TD(transient(s^1))$  to a term  $t$  will proceed as follows. The first time that  $s^1$  can be successfully applied to a term in  $t$  the strategy  $transient(s^1)$  will be reduced to *skip*. The *skip* strategy will then be the strategy applied to the remaining terms encountered in the traversal. Such traversals could be short-circuited, but this is an implementation detail. Similar arguments hold for the definitions of *BU* and *tdl.broadcast*.

## 6 Benchmarks: Union, Intersection, and Zip

We believe that union, intersection, and zip have characteristics similar to a number of common transformational activities. For example, variations of set union can be used as the basis for constant propagation, variable renaming, data flow analysis, control flow analysis, Java constant pool normalization, as well as field distribution and method method table construction in Java class files. Thus, because of their wide range of applicability, we consider union, intersection, and zip to be benchmarks for the distributed data problem.

In this section we show how these benchmarks 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 an element into a union (i.e., a set) only if the element does not already occur in the union. In *TL* the construction of these types of problem specific first-order strategies can be accomplished through higher-order strategies.

In Figure 8 is a BNF grammar describing a language of set/sequence expressions. The meta-symbols of the grammar are  $::=$ ,  $()$ ,  $|$ ,  $<$ ,  $>$ , “, and ”. The term  $()$  is used to denote the epsilon production, domain variables are enclosed in pointy brackets and terminal symbols are enclosed in quotes.

In Figure 9, *keep*, *add*, *remove*, and *tuple* are strategies realizing primitive operations on sets such as adding an element to an empty set (i.e., empty list) or removing an element from a set. The strategies *union\_s*, *intersect\_s*, and *zip\_s* are higher-order and respectively define a single computational step (e.g., “union” one element to a set) for union, intersection, and zip. These computational steps are defined in terms of strategic expressions involving the primitive strategies mentioned previously. And finally, the strategies *make\_union*, *make\_intersection*, and *make\_zip* perform their respective set/sequence operations by first properly instantiating *union\_s*, *intersect\_s*, and *zip\_s* with respect to a  $set_1$  and then applying the resulting strategy to a  $set_2$ . We now look at the implementation of these benchmarks in greater detail.

set_expr	::= set set_op set   set
set	::= “{” es “}”
es	::= e es   ()
e	::= <id>   “(” <id> <id> “)”
set_op	::= “union”   “intersect”   “zip”

Fig. 8. A BNF describing set/sequence expressions

<i>keep</i> ( $e_1$ )	: $es[[e_1 es_2]] \rightarrow es[[e_1 es_2]]$
<i>add</i> ( $e_1$ )	: $es[[ ]] \rightarrow es[[e_1]]$
<i>remove</i>	: $es[[e_1 es_2]] \rightarrow es_2$
<i>tuple</i> ( $e_1$ )	: $es[[e_2 es_2]] \rightarrow es[[ (e_1 e_2) es_2 ]]$
<i>union_s</i>	: $es[[ e_1 es_1 ]] \rightarrow transient(keep(e_1) <+ add(e_1))$
<i>intersect_s</i>	: $es[[ e_1 es_1 ]] \rightarrow transient(keep(e_1))$
<i>zip_s</i>	: $es[[ e_1 es_1 ]] \rightarrow transient(tuple(e_1))$
<i>make_union</i>	: $set\_expr[[set_1 union set_2]]$ $\rightarrow TD(lcond\_tdl\ union\_s\ set_1)\ set_2$
<i>make_intersect</i>	: $set\_expr[[set_1 intersect set_2]]$ $\rightarrow BU((lcond\_tdl\ intersect\_s\ set_1) <+ remove)\ set_2$
<i>make_zip</i>	: $set\_expr[[set_1 zip set_2]]$ $\rightarrow TD(lcond\_tdl\ zip\_s\ set_1)\ set_2$

Fig. 9. Instantiation and application of second-order strategies to terms

### 6.1 Union

The strategic theme here is to decompose a set expression  $\{a_1, a_2, \dots, a_n\} \cup \{e_1, e_2, \dots, e_m\}$  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 sequential composition of instances of *union\_s*;

$$\begin{array}{l}
\text{transient}(es[[x1\ es_2]] \rightarrow es[[x1\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x1]]) \\
\leftarrow \text{transient}(es[[x2\ es_2]] \rightarrow es[[x2\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x2]]) \\
\leftarrow \text{transient}(es[[x3\ es_2]] \rightarrow es[[x3\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x3]]) \\
\leftarrow \text{transient}(es[[x4\ es_2]] \rightarrow es[[x4\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x4]])
\end{array}
\begin{array}{l}
\{y_1 \downarrow x_2\ x_3\ y_2\} \\
\{y_1 \downarrow x_2\ x_3\ y_2\}
\end{array}$$

Fig. 10. Union with TD traversal: The term  $y_1$  is unaffected

$$\begin{array}{l}
\text{transient}(es[[x1\ es_2]] \rightarrow es[[x1\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x1]]) \\
\leftarrow \text{transient}(es[[x2\ es_2]] \rightarrow es[[x2\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x2]]) \\
\leftarrow \text{transient}(es[[x3\ es_2]] \rightarrow es[[x3\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x3]]) \\
\leftarrow \text{transient}(es[[x4\ es_2]] \rightarrow es[[x4\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x4]])
\end{array}
\begin{array}{l}
\{y_1\ x_2 \downarrow x_3\ y_2\} \\
\{y_1\ x_2 \downarrow x_3\ y_2\}
\end{array}$$

Fig. 11. Union with TD traversal: The term  $x_2$  changes the strategy

$$\begin{array}{l}
\text{transient}(es[[x1\ es_2]] \rightarrow es[[x1\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x1]]) \\
\leftarrow \text{transient}(es[[x2\ es_2]] \rightarrow es[[x2\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x2]]) \\
\leftarrow \text{transient}(es[[x3\ es_2]] \rightarrow es[[x3\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x3]]) \\
\leftarrow \text{transient}(es[[x4\ es_2]] \rightarrow es[[x4\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x4]])
\end{array}
\begin{array}{l}
\{y_1\ x_2\ x_3 \downarrow y_2\} \\
\{y_1\ x_2\ x_3 \downarrow y_2\}
\end{array}$$

Fig. 12. Union with TD traversal: The term  $x_3$  changes the strategy

$$\begin{array}{l}
\text{transient}(es[[x1\ es_2]] \rightarrow es[[x1\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x1]]) \\
\leftarrow \text{transient}(es[[x2\ es_2]] \rightarrow es[[x2\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x2]]) \\
\leftarrow \text{transient}(es[[x3\ es_2]] \rightarrow es[[x3\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x3]]) \\
\leftarrow \text{transient}(es[[x4\ es_2]] \rightarrow es[[x4\ es_2]] \leftarrow es[[\ ]] \rightarrow es[[x4]])
\end{array}
\begin{array}{l}
\{y_1\ x_2\ x_3\ y_2 \downarrow x_1\} \\
\{y_1\ x_2\ x_3\ y_2\ x_1\}
\end{array}$$

Fig. 13. Union with TD traversal: The term  $x_1$  is added to the union

one instance for each element in  $set_1$ . The resulting strategy is then applied to  $set_2$  using the traversal TD. 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 Figure 10. Similarly, Figures 11 and 12 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 (similar to Figure 10) and is not shown. And finally, 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.

transient( $es[[x1\ es_2]] \rightarrow es[[x1\ es_2]]$ )	$\{y1\ x2\ x3\ \overset{\downarrow}{y2}\}$
<+ transient( $es[[x2\ es_2]] \rightarrow es[[x2\ es_2]]$ )	
<+ transient( $es[[x3\ es_2]] \rightarrow es[[x3\ es_2]]$ )	
<+ transient( $es[[x4\ es_2]] \rightarrow es[[x4\ es_2]]$ )	
<+ <i>remove</i>	$\{y1\ x2\ x3\ \overset{\downarrow}{\cancel{y2}}\}$

Fig. 14. Intersection with BU Traversal: Remove the term  $y_2$

transient( $es[[x1\ es_2]] \rightarrow es[[x1\ es_2]]$ )	
<+ transient( $es[[x2\ es_2]] \rightarrow es[[x2\ es_2]]$ )	
<+ transient( $es[[x3\ es_2]] \rightarrow es[[x3\ es_2]]$ )	$\{y1\ x2\ \overset{\downarrow}{x3}\ \cancel{y2}\}$
<+ transient( $es[[x4\ es_2]] \rightarrow es[[x4\ es_2]]$ )	
<+ <i>remove</i>	

Fig. 15. Intersection with BU Traversal: Keep the term  $x_3$

## 6.2 Intersection

The strategic idea here is to use a set  $S_1$  to construct a strategy that when applied to a set  $S_2$  will keep only those elements that are also in  $S_1$  and remove the rest. The higher-order strategy *intersect\_s* generates strategies that can be used to “keep” elements. Specifically, when given the context  $es[[ e_1\ es_1 ]]$ , *intersect\_s* will extract the element  $e_1$  and produce a transient strategy *keep*( $e_1$ ).

Building on *intersect\_s* is the strategic expression  $(lcond\_tdl\ intersect\_s\ set_1)$  which creates a conditionally composed sequence of “keep” strategies, one for each element in  $set_1$ . The remove strategy is then appended to the very end of this strategy yielding:  $(lcond\_tdl\ intersect\_s\ set_1)\ <+ remove$ . When applied to an element  $e_2$  in  $set_2$ , this strategy will leave  $e_2$  untouched if there is a corresponding keep strategy in the strategy expression (i.e.,  $e_2$  is also in  $set_1$ ). Otherwise, the remove strategy will apply and  $e_2$  will be removed from  $set_2$ . Keeping this in mind let us examine the bottom up application of the strategy  $(lcond\_tdl\ intersect\_s\ set_1)$  to the term  $set_2$  where  $set_1 = \{x_1\ x_2\ x_3\ x_4\}$  and  $set_2 = \{y_1\ x_2\ x_3\ y_2\}$ .

In Figure 14 shows that the element  $y_2$  will be removed from  $set_2$  since no “keep” strategy applies. Figures 15 and 16 show that the third and second “keep” strategies apply to  $x_3$  and  $x_2$  respectively and therefore  $x_3$  and  $x_2$  remain in  $set_2$ . And finally, in Figure 17 the element  $y_1$  is removed from  $set_2$ . Thus the only elements remaining in  $set_2$  are  $x_2$  and  $x_3$  which is the intersection of  $set_1$  and  $set_2$ .

transient( $es[[x1\ es_2]] \rightarrow es[[x1\ es_2]]$ )	
<+ transient( $es[[x2\ es_2]] \rightarrow es[[x2\ es_2]]$ )	$\{y_1\ \overset{\downarrow}{x_2}\ x_3\ \psi_2\}$
<+ transient( $es[[x3\ es_2]] \rightarrow es[[x3\ es_2]]$ )	
<+ transient( $es[[x4\ es_2]] \rightarrow es[[x4\ es_2]]$ )	
<+ <i>remove</i>	

Fig. 16. Intersection with BU Traversal: Keep the term  $x_2$

transient( $es[[x1\ es_2]] \rightarrow es[[x1\ es_2]]$ )	$\{y_1\ \overset{\downarrow}{x_2}\ x_3\ \psi_2\}$
<+ transient( $es[[x2\ es_2]] \rightarrow es[[x2\ es_2]]$ )	
<+ transient( $es[[x3\ es_2]] \rightarrow es[[x3\ es_2]]$ )	
<+ transient( $es[[x4\ es_2]] \rightarrow es[[x4\ es_2]]$ )	
<+ <i>remove</i>	$\{\overset{\downarrow}{\psi_1}\ x_2\ x_3\ \psi_2\}$

Fig. 17. Intersection with BU Traversal: Remove the term  $y_1$

<del>transient(<math>es[[e_2\ es_2]] \rightarrow es[[x1\ e_2]\ es_2]]</math>)</del>	$\{y_1\ \overset{\downarrow}{y_2}\ y_3\ y_4\}$
<+ transient( $es[[e_2\ es_2]] \rightarrow es[[x2\ e_2]\ es_2]]$ )	$\{(x1\ y_1)\ y_2\ y_3\ y_4\}$
<+ transient( $es[[e_2\ es_2]] \rightarrow es[[x3\ e_2]\ es_2]]$ )	
<+ transient( $es[[e_2\ es_2]] \rightarrow es[[x4\ e_2]\ es_2]]$ )	

Fig. 18. Zip with TD traversal: Creating the tuple  $(x_1, y_1)$

<del>transient(<math>es[[e_2\ es_2]] \rightarrow es[[x2\ e_2]\ es_2]]</math>)</del>	$\{(x1\ y_1)\ \overset{\downarrow}{y_2}\ y_3\ y_4\}$
<+ transient( $es[[e_2\ es_2]] \rightarrow es[[x3\ e_2]\ es_2]]$ )	$\{(x1\ y_1)\ (x2\ y_2)\ y_3\ y_4\}$
<+ transient( $es[[e_2\ es_2]] \rightarrow es[[x4\ e_2]\ es_2]]$ )	

Fig. 19. Zip with TD traversal: Creating the tuple  $(x_2, y_2)$

### 6.3 Zip

The *zip-s* strategy is a higher-order strategy that when given an element as described by the pattern  $es[[e_1\ es_1]]$  will produce a transient instance of the strategy  $tuple(e_1)$ . The expression  $(lcond.tdl\ zip-s\ set_1)$  is used to traverse  $set_1$  and produce a conditional composition consisting of an appropriately instantiated transient strategy for every element in  $set_1$ . Figures 18 and 19 show the first two tuples that get created when  $(lcond.tdl\ zip-s\ set_1)$  is applied to  $set_2$  where  $set_1 = \{x_1\ x_2\ x_3\ x_4\}$  and  $set_2 = \{y_1\ y_2\ y_3\ y_4\}$ . The remaining tuples are created in a similar manner and therefore not shown.

## 7 A Class Loader for Java

At Sandia National Laboratories, a subset of the Java Virtual Machine (JVM) has been developed in hardware for use in high-consequence embedded applications. The implementation is called the *Sandia Secure Processor* (SSP) [24]. An application program for the SSP is called a *ROM image* and consists of a collection of structures similar to class files that have been stored on a read-only memory. The SSP is a *closed system* in the sense that the execution of an application program may only access the structures in the ROM (e.g., no dynamic loading of class files across a network). The closed nature of the SSP's execution environment enables the class loading activities of the JVM to be performed statically. Under these conditions, the functionality of the class loader is well-suited to a strategic implementation.

In the discussion that follows, we assume that an *application* consists of one or more Java *class files* and that Java class files have the structure defined in [23] subject to some minor restructuring to facilitate strategic objectives. For the purposes of this discussion the important things to know about class files is that they contain:

- (1) A *class* entry whose value denotes the name of the class.
- (2) A *constant pool* whose entries contain various forms of data such as a full description of the fields that are explicitly used within the class.
- (3) A *fields section* containing all of the fields, both static and instance, declared within the class.

Activities that can be performed statically include (1) *constant pool normalization* – which consists of removing indirection from constant pool entries and (2) *field distribution* – which consists of distributing field address information across all constant pool entries within an application.

### 7.1 Constant Pool Normalization

In this section we will look at how higher-order strategies can be used to remove indirection from constant pool entries. This problem, which we call *constant pool normalization*, is a real-world instance of the *table normalization* problem presented in Section 4.1.

Figure 20 gives an example, in human readable form, of the kind of information found in the constant pool of a Java class file. In particular, the contents of a constant pool entry may be a utf8 value (i.e., a string) or one or two indexes to other constant pool entries. For example, the fourth entry in Figure 20 describes a field whose class (name) index can be found at entry 2 and whose

Index	Original Type	Contents
1	<code>constant_utf8_info</code>	<code>animal</code>
2	<code>constant_class_info</code>	<code>name_index = 1</code>
3	<code>constant_name_and_type_info</code>	<code>name_index = 5</code> <code>descriptor_index = 6</code>
4	<code>constant_fieldref_info</code>	<code>class_index = 2</code> <code>name_and_type_index = 3</code>
5	<code>constant_utf8_info</code>	<code>x</code>
6	<code>constant_utf8_info</code>	<code>I</code>

Fig. 20. Unresolved constant pool entries

Unresolved	Partially Resolved	Resolved
<code>class_index = 2</code>	<code>→ name_index = 1</code>	<code>→ animal</code>
<code>name_and_type_index = 3</code>	<code>→ name_index = 5</code> <code>descriptor_index = 6</code>	<code>→ x</code> <code>→ I</code>

Fig. 21. Normalization sequence for entry 4

(field) name and type indexes can be found at entry 3. Similarly, entry 2 contains an index to a utf8 entry whose value denotes the name of the class.

A *resolution step* of a constant pool entry is performed by replacing an index to an entry with the data contained in the entry. Resolution steps should be repeated until all indirection has been removed, at which time the constant pool is *normalized*. The resolution steps leading to the normalization of entry 4 is shown in Figure 21.

Our approach to solving the index resolution problem begins with a language redesign phase. Shown in Figure 22 is a redesigned grammar fragment describing the structure of the constant pool within a Java class file. Symbols enclosed in square brackets [ ] denote optional portions of a production. The naming conventions have been taken from the JVM specification [23] to the extent possible. The primary grammar redesign has been to group constant pool index and utf8 entries under the nonterminal symbol `data`. The reason for this is that we want to minimize the number of strategies needed to rewrite indexes to utf8's.

Analysis shows that the resolution of any entry in the constant pool can be achieved in two steps or less. In the *TL* solution developed, the first of these resolution steps is performed by the strategy *resolve\_data* whose goal is to resolve all indexes that refer directly to `constant_utf8_info` entries. We will

constant_pool	::= cp_info_list
cp_info_list	::= cp_info cp_info_list   ()
cp_info	::= [ access ] base_entry
access	::= [ <offset> ] <index>
base_entry	::= constant_class_info   constant_utf8_info   constant_fieldref_info   constant_methodref_info   constant_name_and_type_info   constant_integer_info
constant_name_and_type_info	::= name descriptor
constant_fieldref_info	::= class name_and_type
constant_methodref_info	::= class name_and_type
constant_class_info	::= name
constant_integer_info	::= <bytes>
constant_utf8_info	::= <utf8>
class	::= name
name	::= data
name_and_type	::= data
descriptor	::= data
data	::= <index>   <utf8>   name descriptor

Fig. 22. A redesigned grammar fragment for the Java class file structure

use the term *data-index* to denote indexes of this type. In the *TL* fragment shown below the strategy *resolve\_data* uses an auxiliary strategy *resolve\_index* to perform a resolution step on a data-index. We would like to mention that the abstraction provided by the auxiliary strategy *resolve\_index* serves no purpose other than to enhance readability.

---


$$\begin{aligned}
& \text{resolve\_index}(index_1, utf8_1) : \text{data}[[index_1]] \rightarrow \text{data}[[utf8_1]] \\
& \text{resolve\_data} : \text{cp\_info}[[index_1 \text{ constant\_utf8\_info}_1]] \rightarrow \text{resolve\_index}(index_1, utf8_1) \\
& \text{if } \text{constant\_utf8\_info}_1 \gg \text{constant\_utf8\_info}[[ \text{utf8}_1]]
\end{aligned}$$


---

In the strategy *resolve\_data*, the term *cp\_info*[[*index*<sub>1</sub> *constant\_utf8\_info*<sub>1</sub>]] is used to match constant pool elements consisting of an indexed utf8 entry. When such a match succeeds, the resulting information is passed on to the strategy *resolve\_index* which generates a single rule capable of rewriting any **data** element containing the *index*<sub>1</sub> into its corresponding value *utf8*<sub>1</sub>.

In general, a constant pool will contain a number of utf8 entries. For each such entry a rewrite rule of the kind previously described will need to be generated. The generation of these rules is accomplished by the second-order strategic expression (*lcond\_tdl resolve\_data ClassFile*<sub>0</sub>). The evaluation of this expression yields a first-order strategy capable of resolving all data-indexes in *ClassFile*<sub>0</sub>. Thus, the resolution of all data-indexes in *ClassFile*<sub>0</sub> can be accomplished by the strategic expression:

$$TD(\text{lcond\_tdl } \text{resolve\_data } \text{ClassFile}_0) \text{ClassFile}_0$$

In Figure 23, this expression is used in the first match equation of the if-condition in the *resolve* strategy.

The second resolution step is performed by two strategies *resolve\_class* and *resolve\_nt*. In the constant pool, some indexes refer to **constant\_class\_info** entries rather than **utf8** entries. We will use the term *class-index* to denote indexes of this type. In a fashion similar to *resolve\_data*, the *resolve\_class* strategy defines how class-indexes can be resolved. In the implementation shown below the strategy *resolve\_class* also uses *resolve\_index* to perform a resolution step on a class-index.

---


$$\begin{aligned}
& \text{resolve\_class} : \text{cp\_info}[[index_1 \text{ constant\_class\_info}_1]] \rightarrow \text{resolve\_index}(index_1, utf8_1) \\
& \text{if } \text{constant\_class\_info}_1 \gg \text{constant\_class\_info}[[ \text{utf8}_1]]
\end{aligned}$$


---

Note that after the *resolve\_data* strategy has been used to partially resolve **data**, **constant\_class\_info** entries will contain a utf8 value rather than a class-index. The *resolve\_class* strategy is based on the assumption that entries will have this intermediate form.

The last type of index refers to **constant\_name\_and\_type\_info**. We will use the term *nt-index* to denote indexes of this type. The resolution of nt-indexes is accomplished by the strategy *resolve\_nt*. As was the case with the previous strategies, *resolve\_nt* makes use of an auxiliary strategy (called *resolve\_index*)

which is included to enhance readability.

---


$$\begin{aligned}
& \text{resolve\_nt\_index}(index_1, utf8_1, utf8_2) : \text{name\_and\_type}[[index_1]] \\
& \qquad \qquad \qquad \rightarrow \text{name\_and\_type}[[utf8_1 \text{ } utf8_2]] \\
& \text{resolve\_nt} : \text{cp\_info}[[index_1 \text{ } \text{constant\_name\_and\_type\_info}_1]] \\
& \qquad \qquad \rightarrow \text{resolve\_nt\_index}(index_1, utf8_1, utf8_2) \\
& \qquad \text{if } \text{constant\_name\_and\_type\_info}_1 \\
& \qquad \qquad \gg \text{constant\_name\_and\_type\_info}[[utf8_1 \text{ } utf8_2]]
\end{aligned}$$


---

After the *resolve\_data* strategy has been used to partially resolve **data**, all entries of type **constant\_name\_and\_type\_info** will contain two utf8 values rather than two data-indexes. The *resolve\_nt* strategy is based on the assumption that entries will have this intermediate form.

In general, a constant pool will contain a number of **constant\_class\_info** and **constant\_name\_and\_type\_info** entries. For each such entry an appropriate rewrite rule of the kind previously described will need to be generated. The generation of these rules is accomplished by the second-order strategic expression:

$$lcond\_tdl \text{ (resolve\_class; resolve\_nt) } \text{ClassFile}_1$$

The assumption here being that *ClassFile*<sub>1</sub> is an intermediate form of a class file in which all data-indexes have been resolved.

And finally, the top-down application of the resulting first-order strategy to the term *ClassFile*<sub>1</sub> will resolve all class-indexes and nt-indexes in *ClassFile*<sub>1</sub>. This is accomplished in the second part of if-condition the *resolve* strategy shown in Figure 23 which gives the entire *TL* implementation for solving the constant pool normalization problem. For a more detailed discussion of both the SSP project and transformation in this setting see [42].

## 7.2 Field Distribution

A Java application consists of a collection of class files. In this section we look at the problem of distributing field offset information among all the constant pool entries in a Java application. We refer to this activity as *field distribution*. Field distribution assumes that offset and absolute addresses have been computed for all fields in the application. By this we mean that for all classes in the application, every instance field in its fields section has been annotated with an appropriate offset address and every static field has been annotated

$resolve\_index(index_1, utf8_1)$	:	$data[[index_1]] \rightarrow data[[utf8_1]]$
$resolve\_nt\_index(index_1, utf8_1, utf8_2)$	:	$name\_and\_type[[index_1]]$ $\rightarrow name\_and\_type[[utf8_1\ utf8_2]]$
$resolve\_data$	:	$cp\_info[[index_1\ constant\_utf8\_info_1]] \rightarrow resolve\_index(index_1, utf8_1)$ if $constant\_utf8\_info_1 \gg constant\_utf8\_info[[\ utf8_1]]$
$resolve\_class$	:	$cp\_info[[index_1\ constant\_class\_info_1]] \rightarrow resolve\_index(index_1, utf8_1)$ if $constant\_class\_info_1 \gg constant\_class\_info[[\ utf8_1]]$
$resolve\_nt$	:	$cp\_info[[index_1\ constant\_name\_and\_type\_info_1]]$ $\rightarrow resolve\_nt\_index(index_1, utf8_1, utf8_2)$ if $constant\_name\_and\_type\_info_1$ $\gg constant\_name\_and\_type\_info[[utf8_1\ utf8_2]]$
$resolve$	:	$ClassFile_0 \rightarrow ClassFile_2$ if $ClassFile_1 \ll TD(lcond\_tdl\ resolve\_data\ ClassFile_0)\ ClassFile_0$ $\wedge\ ClassFile_2 \ll$ $TD(lcond\_tdl\ (resolve\_class; resolve\_nt)\ ClassFile_1)\ ClassFile_1$

Fig. 23. Strategies for constant pool normalization

with an appropriate absolute address. Though this article does not discuss the strategies needed to compute such addresses, we would like to mention that they involve transient strategies.

In our discussion here, we restrict the field distribution problem to instance fields (i.e., fields having offset addresses). The strategic objective at this point is to distribute field offset information to all appropriate constant pool entries within the application. For example, let *animal x int* denote an entry occurring in the normalized constant pools of one or more classes in the application and suppose that :0004 is the offset address that has been calculated for the field *animal x*. Field distribution would require the following rewrite rule:

$$animal\ x\ int \rightarrow :0004\ int$$

to be applied to every constant pool entry in the application containing the value *animal x int*. Note that an occurrence of *animal x int* need not be restricted to the class *animal* in which the field *x* is declared; it can occur almost anywhere within the class hierarchy of the application.

Though we have implemented a solution to full field distribution problem, for the sake of brevity, in this article we consider the field distribution problem

application	::=	classfile   classfile application
classfile	::=	“{” class constant_pool field_section “}”
constant_pool	::=	“{” cp_list “}”
cp_list	::=	cp_entry [ cp_list ]
cp_entry	::=	class field type   offset type
field_section	::=	“{” field_list ”}”
field_list	::=	f_entry [ field_list ]
f_entry	::=	field offset
class	::=	id
field	::=	id
offset	::=	<HEX>
id	::=	<ident>
type	::=	“int”   “long”   “byte”

Fig. 24. A simplified Java grammar

class	constant pool	instance fields
{ A	{ C x int B y byte A z long }	{ x:0004 y:000C z:0014 } }
{ B	{ A y long B y byte C z int }	{ x:0004 y:0005 z:0006 } }
{ C	{ A x long C y int B z byte }	{ x:0004 y:0008 z:000C } }

Fig. 25. Class files prior to distribution

class	constant pool	instance fields
{ A	{ :0004 int :0005 byte :0014 long }	{ x:0004 y:000C z:0014 } }
{ B	{ :000C long :0005 byte :000C int }	{ x:0004 y:0005 z:0006 } }
{ C	{ :0004 long :0008 int :0006 byte }	{ x:0004 y:0008 z:000C } }

Fig. 26. Class files after distribution

in the context of the simplified Java grammar given in Figure 24. Given this grammar, the an application consisting of the class files A, B, and C could be expressed as shown in Figure 25. Here { A { C x int B y byte A z long } { x:0004 y:000C z:0014 } } denotes the class A whose constant pool contains the fields C x int, B y byte, and A z long. The class A declares the instance fields x, y, and z whose offsets are :0004, :000C, and :0014 respectively. The remaining entries in the example can be described in a similar fashion. Figure 26 shows the result after field distribution.

$Field(class_1)$	: $f\_entry[[field_1\ offset_1]]$ $\rightarrow cp\_entry[[class_1\ field_1\ type_1]] \rightarrow cp\_entry[[offset_1\ type_1]]$
$Class$	: $classfile[[\{class_1\ constant\_pool_1\ field\_section_1\}]]$ $\rightarrow lcond\_tdl (Field\ class_1)\ field\_section_1$
$Field\_Distribution$	: $application_1 \rightarrow TD(lcond\_tdl\ Class\ application_1)\ application_1$

Fig. 27. A higher-order strategic solution to the field distribution problem

The solution in *TL* to the field distribution problem is shown in Figure 27. Let us take a closer look at strategies involved in this higher-order solution. Recall that according to the BNF in Figure 24, an application consists of a list of *classfiles* and each *classfile* term in turn contains a *field\_section* consisting of a list of *f\_entry* elements. Our high-level strategic design is as follows: From a top-down perspective, we first create an instance of the *Class* strategy for every *classfile* in the application. The strategic expression ( $lcond\_tdl\ Class\ application_1$ ) accomplishes this and yields a strategy that, abstractly speaking, has the form:

$$Class_1 \leftarrow Class_2 \leftarrow \dots \leftarrow Class_n$$

In turn, each  $Class_i$  produces an instance of the *Field* strategy for every *f\_entry* in the *field\_section* of its corresponding class file. The strategic expression ( $lcond\_tdl (Field\ class_i)\ field\_section_i$ ) accomplishes this and yields a strategy of the form:

$$Field_{i,1} \leftarrow Field_{i,2} \leftarrow \dots \leftarrow Field_{i,m}$$

These rewrite rules are the ones that ultimately perform field resolution. For example, with respect to the class A given in Figure 25 the evaluation of the strategic expression ( $lcond\_tdl (Field\ class_1)\ field\_section_1$ ) will produce:

---


$$\begin{aligned} & cp\_entry[[ A\ x\ int\ \ ]] \rightarrow cp\_entry[[ :0004\ int\ \ ]] \\ \leftarrow & cp\_entry[[ A\ y\ byte\ \ ]] \rightarrow cp\_entry[[ :000C\ byte\ \ ]] \\ \leftarrow & cp\_entry[[ A\ z\ long\ \ ]] \rightarrow cp\_entry[[ :0014\ long\ \ ]] \end{aligned}$$


---

Similar strategies will result from the evaluation of other classes. When considered in their totality, the resulting first-order strategies contain the information needed to resolve every instance field in  $application_1$ . A full traversal that applies these strategies to the term  $application_1$  solves the field distribution problem.

## 8 HATS: A Restricted Implementation of $TL$

HATS [40][41] is an integrated development environment (IDE) for strategic programming in a restricted dialect of  $TL$ . The IDE consists of an interface written in Java and an execution engine written in ML. The interface supports file management, provides specialized editors for various file types including an editor that highlights  $TL$  keywords and terms. The interface also supports the graphical display of term structures. The execution engine consists of three components: a parser, an interpreter, and an abstract pretty-printer. All of the examples discussed in this article have been implemented in HATS. HATS runs on Windows NT/2000/XP and Unix-based platforms and is freely available [13].

## 9 Related Work

In this section we take a look at various frameworks (e.g., systems and calculi) which support a strategic perspective of computation. Our discussion centers around (1) identification of the type of matching/unification supported, (2) whether the framework is failure-based or identity-based, (3) whether the framework is first-order or higher-order, and (4) some thoughts concerning the extent to which ideas similar to the ones presented in this paper might be incorporated within the framework.

### 9.1 The $\rho$ -calculus

The  $\rho$ -calculus [6] is a failure-based rewriting framework in which matching modulo an equational theory provides the mechanism for the syntactic comparison of terms. In the  $\rho$ -calculus the distinction between a rule and a term to which a rule is applied is blurred. Both rules and terms are considered  $\rho$ -terms. This uniform treatment is reminiscent of the relationship between functions and terms in the  $\lambda$ -calculus. And, similar to the  $\lambda$ -calculus, in the  $\rho$ -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. While it should theoretically be possible to simulate the constructs described in this paper within the  $\rho$ -calculus, the solution does not appear obvious.

## 9.2 ELAN

ELAN [3] is a first-order failure-based rewrite system in which an AC matching algorithm [11] can be used as the mechanism for the syntactic comparison of terms. ELAN is a strategic system whose semantic foundation rests upon the  $\rho$ -calculus. Rewrite rules can be labeled and one or more rules may share the same label. Thus labels are bound to rule bases. The consequence of AC matching and labeled rule bases is that the application of a rule (base) to a specific term may yield multiple results. This form of non-determinism surrounding rule base application is central to ELAN and gives the system a deductive/declarative flavor. ELAN provides a variety of choice combinators together with a backtracking capability as mechanisms for dealing with the non-determinism.

It appears that the constructs discussed in this article would be difficult to implement directly in ELAN. However, ELAN is a modular language so it should be possible to extend ELAN with a module supporting an appropriate variation of the higher-order strategy construction mechanisms of *TL*. Such an extension may have non-trivial implications since it would raise ELAN to a higher-order system. Furthermore, since ELAN has a failure-based semantics, it is unclear how the transient combinator could be incorporated.

## 9.3 Stratego

Stratego is a first-order failure-based strategic programming system in which matching provides the mechanism for the syntactic comparison of terms. Stratego has two constructs related to the higher-order strategies presented in the paper: *contextual rules* and *scoped dynamic rewrite rules*. In [34], contextual rules are used to distribute data within a term structure and can be seen as a first-order cousin of the higher-order rules presented in this paper. Operationally, the term association in a contextual rule can be implemented using a nested traversal to search for a set of terms satisfying the rule.

In [32], an approach to the distributed data problem is taken that is similar to what we have described. Here the distributed data problem is viewed from a context-free/context-sensitive perspective. In particular, semantic relationships between portions of a term are seen as representing context-sensitive relationships. Dynamic rewrite rules are developed as a mechanism for capturing these relationships. 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 program 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  $\{ | \dots | \}$ .

Primary differences between our approach and the scoped dynamic rules described in [32] are the following:

- (1) In our approach, we view the rule base as a *strategy* that is created dynamically. The  $\oplus$  combinator provides the user explicit control over the structure of this strategy.
- (2) Though the transient combinator has no direct analogy within scoped dynamic rewrite rules, its effects can be simulated in Stratego [35]. 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.

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 [35].

#### 9.4 ASF+SDF

ASF+SDF [1] is a first-order identity-based rewriting framework in which an extended form of matching provides the mechanism for the syntactic comparison of terms. The extension to matching permits associative matching on lists structures. In [5] ASD+SDF is further developed so that one can combine parameterized rewrite rules with a fixed set of generic traversals. The result of such a combination is a *traversal function* – which is essentially a rewrite rule annotated with an appropriated predefined traversal. One of the goals in [5] is to provide primitives so that the resulting traversal functions can be used in a type-safe manner.

The fact that ASF+SDF is identity-based should make it possible to incorporate the transient combinator described in this article. However, without the ability to dynamically generate rule (equation) instances the usefulness of the transient is unclear.

#### 9.5 The $S'_\gamma$ Calculus

The  $S'_\gamma$  calculus [21] is a first-order failure-based strategic programming framework in which matching provides the mechanism for the syntactic comparison

of terms. The  $S'_\gamma$  calculus is a strongly typed cousin of system  $S$  supporting a variety of combinators for generic one-layer traversal which can be recursively composed to produce typed generic traversals. Fundamental to the  $S'_\gamma$  calculus is the strategy extension combinator  $\triangleleft$  which lifts a many-sorted strategy  $s$  to a generic type  $\gamma$ . A type inferencing system supporting a restricted form of parametric polymorphism is developed in which strategies fall into one of two categories: type-preserving strategies and type-unifying strategies. Tuples, lists, strategy parameterization, and the implicit binding of free variables in strategies by embedding them in scopes in which the variables are bound (e.g., via *where* clauses) are the primary mechanisms used for addressing the distributed data problem.

In [21] a combinator  $\bigcirc(\cdot)$  is introduced where  $\cdot$  denotes the placeholders for arguments. A strategy expression of the form  $\bigcirc^{s_0}(s)$  will process all the children of a term using the strategy  $s$  and will then compose the result using the strategy  $s_0$ . This combinator could be defined in the framework of  $TL$  as follows:

$$\bigcirc^{s_0}(s) = \text{all\_broadcast}(\mathcal{I}, s, s_0) \text{ where } \mathcal{I} \text{ is the identity strategy}$$

The combinator  $\bigcirc(\cdot)$  is used as the basis for defining a strategy  $CF$  which has a semantics can be understood in terms of the catamorphism *fold*.

The failure-based semantics of  $S'_\gamma$  make the incorporation of a transient-like combinator problematic. The incorporation of higher-order strategy instantiation as presented in this article would seem to be a natural lifting of combinators like  $\bigcirc(\cdot)$  to a higher-order. However, this would considerably complicate the type system of the  $S'_\gamma$  calculus.

## 9.6 Functional Strategies

Historically, the functional and strategic programming communities have had different research interests. Within the functional community, type systems [27][29], polytypic programming [14], morphisms [25], and monads [39][26] are being extensively investigated. In contrast, the strategic community has looked deeply into (1) term structure recognition [11], (2) controlling of term traversal [34], and (3) the development of a clean (i.e., generic) separation between generic control and rule definition [34][5]. However, as the complexity and size of strategic programs increases, so too does the appreciation of the benefits offered by strong (static) typing. As a result, an area of current research focuses on bringing strategic programming concepts into a functional framework<sup>4</sup>

<sup>4</sup> In contrast, the  $S'_\gamma$  calculus is an effort to bring strong typing into a strategic framework.

with the result being a *functional strategy* [16][18].

The notions of functional strategies originated from the observation that catamorphisms (i.e., what [25] refers to as “bananas”) such as  $\text{fold } b \oplus$  could 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 [19].

The ideas discussed above can be further lifted into the realm of monads. Monads enable information to be propagated throughout the strategic computation. One piece of information that is interesting from a strategic perspective is the success or failure of the application of a strategy. Such information, together with backtracking enable monadic algebras to express the *choice* combinator. Strafinski [17][18][20] is a Haskell-based system implementing the ideas discussed here.

Functional strategies as described here go beyond first-order strategies in the sense that the application of a traversal may return a function. It would be interesting to explore whether the crush combinator could be used to implement the *TL* the higher-order strategy construction mechanisms of *TL* as well as the transient combinator. The plumbing resulting from the application of transients could also be captured within a monad.

### 9.7 Maude

Maude [10] is an equational and rewriting system in which matching modulo equational theories provides the mechanism for the syntactic comparison of terms. Maude is based on a refined form of algebraic specification built on top of a *membership equational logic*. The two fundamental constructs in Maude are the  $\Sigma$ -equation and the conditional  $\Sigma$ -equation. While conditional  $\Sigma$ -equations syntactically might appear similar to our conditional rewrites, their semantics is quite different. In particular, an equation  $u = v$  belonging to a condition holds only if, under the given substitution  $\sigma$ , the (irreducible) normal form of the left-hand side of the equation is equivalent to its right-hand side. That is,  $\sigma(u) \downarrow_E \equiv \sigma(v) \downarrow_E$ .

While the notions of dynamics and transients are not primitive operations in Maude, its reflective capabilities [7][8][9] should easily support their implementation as well as the rest of the framework described in this paper.

## 10 Conclusion

Rewriting offers an attractive paradigm for describing number of computational systems. The intimate relationship between an equational theory and set of rewrite rules together with the implicit nature of rule application provide an elegant computational paradigm when considered from an analytical perspective based on equational reasoning. However, in areas such as meta-programming pure rewriting has met with limited success. A primary reason for this is that rule bases describing meta-programming goals are frequently neither confluent nor terminating. To cope with this difficulty, rewriting frameworks have been extended so that *strategies* for rule application can be explicitly described. Strategies typically have two axes of control: (1) mechanisms for describing **how** a collection of rules should be applied to a term, and (2) mechanisms for selecting **where** rules should be applied within a term. These two aspects of control are typically made explicit (i.e., under user control) in a strategic programming system.

The control mechanisms offered by strategic programming has been successfully used to address a variety of problems relating to confluence and termination. However, the application of strategic programming to problems of increasing complexity has raised another issue, namely how auxiliary data fits within a strategic framework. The *distributed data problem* characterizes the problem posed by auxiliary data. This problem arises from a discord between the semantic association of terms within a specification and the structural association of terms resulting from a term language definition.

Rule parameterization is one approach for addressing the distributed data problem. In this approach, terms to be used as auxiliary data are typically translated into an intermediate form such as a list or tuple. This value is then added as a parameter to appropriate rules. In general, lookup functions as well as term reconstruction functions will need to be employed to reintroduce intermediate data back into the term structure at the point of use. We feel this approach is a departure from the spirit of strategic programming as well as general rewriting.

Our research is based on the premise that higher-order rewriting provides a mechanism for dealing with auxiliary data conforming to the tenets of rewriting. In a higher-order framework, the use of auxiliary 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 will again 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] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.
- [2] P. Borovansky, C. Kirchner, and H. Kirchner. *Controlling Rewriting by Rewriting*. In J. Meseguer, editor, Proceedings of the First International Workshop on Rewriting Logic and its Applications, volume 4 of Electronic Notes in Theoretical Computer Science, Asilomar, Pacific Grove, CA, September 1996. Elsevier.
- [3] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. *An Overview of ELAN*. In C. Kirchner and H. Kirchner, eds., International Workshop on Rewriting Logic and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science, France, 1998. Elsevier Science.
- [4] P. Borovansky, C. Kirchner, H. Kirchner, and C. Ringeissen. *Rewriting with strategies in ELAN: A Functional Semantics*. International Journal of Foundations of Computer Science, March 2001.
- [5] M.G.J. van den Brand, P. Klint, and J.J. Vinju. *Term Rewriting with Traversal Functions*. ACM Transactions on Software Engineering and Methodology (TOSEM), 12:2, pp 152-190, 2003.
- [6] H. Cirstea and C. Kirchner. *Intoduction to the rewriting calculus*. INRIA Research Report RR-3818, December 1999.
- [7] M. Clavel and J. Meseguer. *Reflection and strategies in rewriting logic*. In J. Meseguer, editor, Electronic Notes in Theoretical Computer Science, vol. 4. Elsevier Science Publishers, 1996. Proceedings of the First International Workshop on Rewriting Logic and its Applications.
- [8] M. Clavel. *Reflection in Rewriting Logic*. CSLI Publications, 2000.
- [9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer . *Metalevel Computation in Maude*. In 2<sup>nd</sup> International Workshop on Rewriting Logic and its Applications (WRLA'98), Vol. 15, Electronic Notes in Theoretical Computer Science, Elsevier, 1998.

- [10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Theoretical Computer Science, 2001.
- [11] S. Eker. *Associative-commutative matching via bipartite graph matching*. Computer Journal, 38(5):381-399, 1995.
- [12] ELAN – User Manual.
- [13] HATS.  
*<http://faculty.ist.unomaha.edu/winter/hats-uno/HATSWEB/index.html>*
- [14] P. Jansson and J. Jeuring. *Polyp - a polytypic programming language extension*. In Conference record of POPL'97, pages 470-482. ACM Press, 1997.
- [15] P. Klint. *A meta-environment for generating programming environments*. ACM Transactions of Software Engineering and Methodology, 2(2):176–201, 1993.
- [16] J. Kort and R. Lämmel and J. Visser. *Functional Transformation Systems*. Proceedings of the 9<sup>th</sup> International Workshop on Functional and Logic Programming, July 2000.
- [17] R. Lämmel and J. Visser. *A Strafunski Application Letter*. In Proceedings of Practical Aspects of Declarative Programming (PADL), Springer-Verlag, LNCS Vol 2562:357–375, Jan 2003.
- [18] R. Lämmel and J. Visser. *Typed Combinators for Generic Traversal*. Proc. Practical Aspects of Declarative Programming PADL 2002, Springer-Verlag, LNCS Vol 2257:137–154, Jan 2002.
- [19] R. Lämmel, J. Visser, and J. Kort. *Dealing with Large Bananas*. In Johan Jeuring, editor, Workshop on Generic Programming, Ponte de Lima, July 2000. Technical Report, Universiteit Utrecht.
- [20] R. Lämmel. *The Sketch of a Polymorphic Symphony*. Electronic Notes in Theoretical Computer Science, Vol. 70:6, B. Gramlich and S. Lucas (editors), Elsevier, 2002.
- [21] R. Lämmel. *Typed Generic Traversal With Term Rewriting Strategies*. Journal of Logic and Algebraic Programming, Vol 54, pp 1–64, 2003.
- [22] R. Lämmel, E. Visser, and J. Visser. *The Essence of Strategic Programming*. Draft.
- [23] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification 2<sup>nd</sup> Edition*. Addison-Wesley, Reading, Massachusetts, 1999.
- [24] J. A. McCoy. *An Embedded System For Safe, Secure And Reliable Execution Of High Consequence Software*. Proceedings of the 5<sup>th</sup> IEEE International High-Assurance Systems Engineering Symposium, Nov. 2000.
- [25] E. Meijer, M.M. Fokkinga, and R. Paterson. *Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire*. In J. Hughes, editor, FPCA'91: Functional Programming Languages and Computer Architecture, volume 523 of LNCS, pages 124-144. Springer-Verlag, 1991.

- [26] E. Meijer and J. Jeuring. *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, volume 925 of Lecture Notes in Computer Science, pages 228–266. Springer-Verlag, Berlin, 1995.
- [27] R. Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences, 17(3):348-375, December 1978.
- [28] K. Olmos and E. Visser. *Strategies for Source-to-Source Constant Propagation*. In B. Gramlich and S. Lucas (editors) Second International Workshop on Reduction Strategies in Rewriting and Programming(WRS'02). Electronic Notes in Theoretical Computer Science vol. 70 No. 6. Elsevier Science Publishers, Copenhagen, Denmark. July 2002.
- [29] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [30] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1998.
- [31] E. Visser. *Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5*. In A. Middeldorp, editor, Rewriting Techniques and Applications (RTA'01), volume 2051 of Lecture Notes in Computer Science, pages 357–361. Springer-Verlag, May 2001.
- [32] E. Visser. *Scoped dynamic rewrite rules*. In M. van den Brand and R. Verma, editors, Rule Based Programming (RULE'01), volume 59/4 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, September 2001.
- [33] E. Visser. *Language Independent Traversals for Program Transformation*. In Johan Jeuring, editor, Workshop on Generic Programming (WGP'00), Ponte de Lima, Portugal, July 2000.
- [34] E. Visser. *Strategic Pattern Matching*. In: Rewriting Techniques and Applications (RTA '99), Trento, Lecture Notes in Computer Science (1999).
- [35] E. Visser. Personal communication, Feb. 18, 2004.
- [36] E. Visser, Z. Benaissa, and A. Tolmach. *Building Program Optimizers with Rewriting Strategies*. Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98).
- [37] E. Visser and Z. Benaissa. *A Core Language for Rewriting*. Eds. C. Kirchner and H. Kirchner Electronic Notes in Theoretical Computer Science Vol. 15, Elsevier Science Publishers, 1998.
- [38] E. Visser. *Meta-Programming with Concrete Object Syntax*. In Lecture Notes in Computer Science, Generative Programming and Component Engineering (GPCE'02), pp 299–315, 2002.
- [39] P. Wadler. *Monads for functional programming*. In J. Jeuring and E. Meijer, editors, Advanced Functional Programming, Springer Verlag, LNCS 925, 1995.

- [40] V. L. Winter. *Program Transformation in HATS*. Proceedings of the Software Transformation Systems (STS) Workshop (part of ICSE '99), May 17 1999.
- [41] V.L. Winter. *An Overview of HATS: A Language Independent High Assurance Transformation System*. Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET), March 24-27, 1999.
- [42] V.L. Winter, S. Roach, G. Wickstrom. *Transformation-Oriented Programming: A Development Methodology for High Assurance Software*. Advances in Computers vol 58.