SANDIA REPORT
SAND2003-4587
Unlimited Release
Printed December 2003

Higher-order Transformation and the Distributed Data Problem

Victor L. Winter
Sandia Contract No. 5137
University of Nebraska at Omaha
Department of Computer Science

Mahadevan Subramaniam
University of Nebraska at Omaha
Department of Computer Science

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2003-4587
Unlimited Release
Printed December 2003

Higher-order Transformation and the Distributed Data Problem

Victor Winter
Sandia Contract No. 5137
University of Nebraska at Omaha
Department of Computer Science
vwinter@ist.unomaha.edu

Mahadevan Subramaniam
University of Nebraska at Omaha
Department of Computer Science
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. Two strategic combinators, *dynamic* and *transient*, are introduced in the context of a classical strategic programming framework. The impact of the resulting system on instances of the distributed data problem is then explored.

Contents

List of Figures

Nomenclature

ASF	Algebraic Specification Framework
BNF	Backus-Naur Form
HATS	High Assurance Transformation System
JVM	Java Virtual Machine
ROM	Read Only Memory
SDF	Syntax Definition Formalism
SSP	Sandia Secure Processor

1 Introduction

Existing transformation techniques have encountered stumbling blocks when dealing with a certain class of manipulations that frequently arise in the context of program transformation. We introduce the term *distributed data problem* to characterize the desire to bring together *semantically related* terms from *syntactically unrelated* portions of a term.

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 now be abstractly defined as follows:

1. Define $D_t^n = D_t \times D_t \times ... \times D_t$ as the n-ary cross-product of D_t.
2. Define a relation $SR(t) \subseteq D_t^n$ such that an n-tuple of terms $d_t^n \in D_t^n$ is semantically related iff $d_t^n \in SR(t)$.

While the idea of semantically related terms motivates the research in this paper, a more detailed definition of the relation is not essential to the concepts presented. Therefore, we use a few examples to give the reader an informal idea of what we mean by *semantically related.*

1. When performing type checking, the type information found at the point of declaration of a variable and the occurrences of that variable within a program are semantically related.
2. When performing function inlining, the bindings that exist between the formal and actual parameters of a function are semantically related.
3. When performing resolution of Java classfiles, the entries in the constant pool are semantically related to the bytecodes which index them.

We define terms $d_t^n \in D_t^n$ to be *syntactically unrelated* from the perspective of the structure of t if one or more **recursive traversals** of t or one of its subterms are required in order to construct d_t^n. That is, d_t^n cannot be directly captured and manipulated within a *primitive* conditional rewrite rule. A conditional rewrite rule is considered to be primitive if none of its components (e.g., *where* clauses, *local evaluations*, etc.) perform any recursive traversals. In this context, the *recursive traversal* 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 definition of what is syntactically unrelated is dependent upon the matching/unification capabilities of the strategic programming system $\mathcal{S}$ as well as the class of term structures under consideration. For example, a system $\mathcal{S}'$ in which AC unification [16][15] is supported will consider a different set of terms to be syntactically unrelated when compared to a system $\mathcal{S}''$ in which only first-order unification is supported. We define $SU(\mathcal{S}, t)$ 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 $\mathcal{S}$.

Given the above definitions of semantically related and syntactically unrelated, we define the distributed data problem for a given system $\mathcal{S}$ and term t as the following relation:

$$\textit{distributed-data-problem}(\mathcal{S}, t) \equiv SR(t) \cap SU(\mathcal{S}, t)$$

Term *normalization*, *strategy instantiation*, and *recursive traversal* are the primary mechanisms used to solve the distributed data problem in a strategic framework. Abstractly speaking, *normalization* alters the subterm relation. Is generally accomplished by the application of a collection of rewrite rules whose goal is to place the term into some canonical form. For example, a term of the form add(add(1,add(2,3)),add(4,5)) may be normalized to yield add(1,add(2,add(3,add(4,5)))). Notice that in the normalized form the first two constants (e.g., in this case 1 and 2) will always be at a fixed depth in the term. Canonical forms may be naturally occuring within the term structure, or the term structure can be extended so they may arise. The restructuring of terms may bring appropriate subterms within reach of unification/matching. Terms that were syntactically unrelated may thus become syntactically related after normalization.

Strategy instantiation, on the other hand, provides the means for binding specific values (e.g., terms) to variables within a stragegy which then can be passed to a traversal. There are two mechanisms by which external values can flow into a strategy: parameterization and free-variables. In parameterization values flow into the strategy through formal parameters. In the free-variable approach, a strategy is defined (possibly anonymous) containing free-variables which are bound in an enclosing scope (e.g., another strategy).

Recursive traversal provides the mechanism for transferring data in an instantiated strategy to a particular term or set of terms. Success is achieved when normalization makes it possible to construct a term traversal in which a parameterized strategy is capable of bringing together semantically related terms. When this has been accomplished, the particular instance of the distributed data problem is considered solved.

Instances of the distributed data problem arise in numerous settings including type-checking, program slicing, partial evaluation, variable renaming, function inlining, as well as constant pool normalization in Java classfiles. In many cases, simple instantiated strategies needed to accomplish a desired task can easily be written (by hand) for a fixed context or problem. 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 a mechanism by which appropriate strategies can be automatically generated.

1.1 Contribution

In this paper, we explore how the combinators of a traditional first-order strategic programming system such as those described in [23] can be lifted to a higher-order setting. We refer to higher-order strategies as *dynamics* because of their ability to dynamically create first-order strategies which are then applied to terms.

In this higher-order framework we also introduce a combinator called a *transient*. The *transient* combinator restricts a strategy so that it may be applied at most once. For example, let s denote an arbitrary strategy. The expression *transient(s)* denotes a strategy in which s can only be applied once. Given a strategic mindset, a *transient strategy* can be understood as a strategy that **transforms itself** into the strategy *skip* after its first successful application to a term. Here 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[33][35] and Elan[2][3] 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 might be one that is non-failure 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 dynamics and transients lead to interesting and elegant solutions to problems involving term structures whose characteristics have been considered undesirable in the context of a more traditional rewriting system.

This article formally integrates dynamics and transients in a strategic programming framework. The resulting language is called TL. The expressiveness of TL is demonstrated by solving various instances of the *distributed data problem*.

1.2 Outline

Section 2 looks at the distributed data problem in more detail and gives a small survey of the existing mechanisms used for solving the distributed data problem. In Section 3 *table normalization* is introduced as a running example. Section 4 describes TL, a higher-order strategic programming langauge featuring dynamic and transient combinators. Section 5 gives several examples of how dynamic and transient combinators can be used to solve general instances of the distributed data problem such as set union and intersection. Section 6 revisits the table normalization problem in a real world setting namely, *constant pool normalization* for the Java Virtual Machine (JVM). *Field distribution* and *method table construction* for the JVM is also discussed. Section 7 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 8 discusses related work including Stratego, the ρ-calculus, ASF+SDF, the S'_γ calculus, Elan, Strafunski[21], and Maude. Section 9 indicates some areas for future work, and Section 10 concludes.

2 Motivation

Existing strategic paradigms generally support tuples, lists, and the ability to bring one or more terms within the scope of a recursive traversal as the primary mechanisms for solving

the distributed data problem. Conceptually speaking, the recursive distribution of data 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.* Recursive traversals must be employed or developed for transporting data to terms.
4. *The use of data.* The data is used at some point to extend, alter, or create one or more terms.

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 (i.e., an accumulator) to an appropriate (sub)term. An accumulator may collect a fixed number of values (e.g., a term or a tuple) or a varying number of values (e.g., a list). When using accumulators, it is generally useful to have the ability to produce initial values such as the empty tuple or the empty list. For example, the S'_γ calculus [22] provides a combinator specifically for building the empty tuple.

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). The relatively simple top-level structure of accumulated values distinguishes them from classes of terms that are more structurally complex such as abstract syntax trees or parse trees which are typically used to describe program structures. Another noteworthy characteristic of strategies playing the role of an accumulator is that they are not type preserving. For instance, an accumulator typically yields an output that is of type list or tuple regardless of the type of the term to which it is applied.

In many strategic programming frameworks the types of accumulated values are extra-grammatical in the sense that they are not native to the abstract syntax or context-free grammar describing the domain of discourse. Instead, the definitions of these *auxiliary structures* can be seen as extensions to the term-language which are either imported, as is the case in ELAN [2], or provided as universal primitives by the underlying strategic system, as is the case in Stratego [34]. The S'_γ calculus [22] as well as ASF+SDF [4] implicitly support tuples as an auxiliary structure. However in both the S'_γ calculus as well as ASF+SDF, users must explicitly define list constructors and constants (e.g., cons and nil).

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, 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. In one approach, a strategy is constructed where a desired accumulated value is denoted by a free variable. This strategy is then embedded in a scope in which this free variable is explicitly bound to the desired accumulated value. In Stratego [34], this can be accomplished by (1) binding an accumulated value to a variable using a *where* clause, (2) defining an anonymous strategy within the scope of the *where* clause, and (3) then passing this anonymous strategy to a recursive traversal function. Stratego also provides a special kind of strategy called a *match strategy* that has the form $?t$. Match strategies can be quite effectively used as mechanism for binding certain types of accumulated values. In [35], *contextual rules* are introduced as a notation for describing distributed data. Here a context can be specified by an expression of the form $t[x]$ which denotes a single occurrence of the term x within the term t. We will look more closely at an example involving contextual rules in Section 2.5.1.

Strategy parameterization is also a mechanism that can be used for binding variables to accumulated values. The problem introduced here is how to integrate/reconcile the parameterization of s with recursive traversals applying s. An extension to ASF+SDF [4] requires that parameters be explicitly passed to *traversal functions*. Here traversal functions are essentially a set of rewrite rules annotated with an appropriate predefined traversal. The traversal mechanism takes care of properly passing the parameter(s) in rules applying to subterms. In this framework, *conditions* can be used in a fashion similar to the *where* clauses in Stratego to bind accumulated values to variables. One drawback of explicit rule parameterization is that general purpose (unparameterized) form of the rules is not readily accessible.

In ELAN[2][13], *local evaluations* can be used to bind accumulated values. Rules can then be specifically parameterized (a non-generic approach) and an evaluation mechanism (e.g., nondeterministic choice, backtracking, normalization, etc.) can be used to bring the parameterized data to the appropriate (sub)term.

2.3 Phase III: The Distribution of Data

In this phase a strategy must be developed that enables a strategy containing a variable bound to an accumulated value to be transfered to appropriate subterms. Ultimately, this transfer of data will involve a recursive traversal which may be generic (e.g., topdown, botttomup, etc.) or problem dependent.

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 one or more terms. 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.

2.5 Examples

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.

2.5.1 Type Checking

In [35], a strategic program is developed for the purpose of type checking a small imperative language named Pico. In Pico, program blocks consist of a declaration list followed by a statement list.

$$Block : List(Decl) * Stat \rightarrow Program$$

The basic idea of the type checker is to rewrite all variables with their type and then use basic type rules to simplify program constructs. For example, a type rule is given for simplifying an expression of the form *Integer* + *Integer* to *Integer*. Similarly, a type rule is given for simplifying an assignment of the form *Integer* := *Integer* to *Skip*. In this approach, the statement list *Stat* will be type correct if it can be rewritten to *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 this preprocessing step is a classic example of the distributed data problem which is solved in this example through the use of *contextual rules.*

In particular, the following *contextual rule* is used to distribute the type data associated with variables found in declarations over statements in s that use the variable:

InlTp: Block(ds[Decl(Id(x),t)], s[Id(x)]) → 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 implemented by the primitive strategic constructs found in Stratego (e.g., match and build strategies, where clauses, and term traversals). 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 walks across the declaration list ds. When a declaration $Decl(Id(x), t)$ is encountered a traversal of the statement sequence s is initiated 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.

The given implementation of contextual rules in Stratego seem to be well suited for specifying a *one-to-many* rerwriting relationship. The outer traversal defines the *one* and the inner traversal defines the *many*. However, in order for this to work it should only be possible to transform each of the terms in the set of the *many* a finite number of times (e.g., one time). If this is not the case, the exhaustive application of such nested of traversals will result in an infinite rewriting sequence. It is worth noting that the nesting of traversals works because the single/finite application property realizes a counter or marker of sorts at the term-level. The *transient* combinator presented in this article lifts this concept to the strategy-level.

The notion of contexts provides a capability that is similar to the dynamic combinator introduced in this article. In fact, one could think of dynamic combinators as some sort of cross between contexts and the scoped dynamic rewrite rules discussed in the next example.

2.5.2 Variable Renaming

In [33] bound variable renaming problem is considered. A basic (conventional) algorithm is outlined 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.

This basic algorithm is adapted to a generic 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 rulebase during execution. Abstractly, this rulebase 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). The idea of dynamically creating rule sets is similar to what we propose in this article.

Given a framework in which scoped dynamic rewrite rules is supported, the variable renaming problem can be solved as follows. First, a labeled rule is created defining variable renaming in general. The body of this rule is essentially of the form: $id_1 \rightarrow id_2$. It is important to note that such a rule, 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. And lastly, what may be even more important to note that in this context, dynamically created rules obviate the need of term-language extensions such as tuples and lists!

2.5.3 General Replacements

In [4] *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. One replacement involves incrementing each integer term by a term denoting a constant integer value. A second example has a higher-order flavor and involves replacing all occurrences of the function symbol g by the function symbol i within a term t.

The 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 is then 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 following example, taken from the paper, 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)
```

The second example involves the substitution of one function symbol for another. In this example, the new symbol is not stored as a parameter to an equation, but rather embedded in the equation itself. The transformer *frepl* is used to define the replacement as follows:

```
frepl(g(T1,T2)) = i(T1,T2)
```

Notice that *frepl* takes only one argument, the term t to which it is applied, while *incp* takes two arguments.

2.5.4 Closure Conversion

In [34] an example is given of a closure conversion for a functional language. The goal of closure conversion is to turn the free variables of a nested function into explicit parameters

of that function. Within a function definition, a variable x is considered free if (1) x is used within the definition, and (2) x is not declared locally.

In the solution presented, an accumulating strategy is used to traverse a term denoting a function definition and collect all free variables. The free variables encountered are placed into a list and this list is then used to extend the formal and actual parameter lists of the function.

In the abstract syntax for the functional language given, a variable use (free or otherwise) is denoted by a term of the form *Var(id,type)*. In contrast, the elements of the accumulated free variable list are tuples of the form *(id,type)*, a term structure not explicitly defined by the abstract syntax. When appending the free variable list to the actual parameter lists of the function the free variable list is first converted into a list of tuples of the form *Var(id,type)*.

2.6 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)? When should data defining a desired change be passed as a parameter (*incp*) and when should it be embedded within a rule or equation ($frepl$)?

Strategic systems generally provide some sort of typing as a byproduct of their computational framework. For example, a rewrite rule of the form $r : f(t_1, ..., t_n) \rightarrow f(t'_1, ..., t'_n)$ can only be successfully applied to terms of the form $f(x_1, ..., x_n)$. Given this, the constructor f can be viewed as a type constraint and the rule r can be seen as being type preserving. In [4], 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[22], 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 – extensions which are essentially typeless. 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'_γ calculus [22] 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 consiting 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.

3 Running Example

This article developes a strategic framework where the bindings possible within higher-order strategies are used place of the parameter passing found in first-order strategic frameworks. This provides a different framework for describing strategic computations in which solutions to a number of strategic problems can be elegantly expressed.

In this section, we consider a basic unlabeled first-order rewrite rule to have the form:

$$lhs \rightarrow rhs$$

where lhs and rhs denote terms taken from a suitable term language $T(F, X)$ in which terms are constructed using the function symbols in F and variables in X. A rewrite rule can be labeled by prefixing it with a label followed by a colon as follows:

$$r : lhs \rightarrow rhs$$

In a strategic setting, a rewrite rule may also be referred to as a strategy.

The expression $[lhs \rightarrow rhs](t)$ denotes the application of the strategy $lhs \rightarrow rhs$ to the term t. We will only consider the application of strategies to ground terms (i.e., terms containing no variable symbols). Given this restriction, rule application can be accomplished by matching lhs with t and then using the bindings resulting from the match in order to construct an instance of the term rhs which then replaces t. An interesting question arises concerning the result of rule application in the case where lhs and t cannot be matched. In a pure rewriting framework, the term t is left unchanged. In a strategic framework the result is typically a value denoting failure.

We consider a second-order rewrite rule to have the form:

$$lhs_2 \rightarrow lhs_1 \rightarrow rhs_1$$

where the $\rightarrow$ symbol is right-associative. The application of the second order strategy $lhs_2 \rightarrow lhs_1 \rightarrow rhs_1$ to a term t will yield a first-order strategy of the form $lhs'_1 \rightarrow rhs'_1$ where lhs'_1 and rhs'_1 are instances respectively of lhs_1 and rhs_1 as determined by variable bindings resulting from the (successful) match between lhs_2 and t.

Given this notation, let us consider the second-order rewrite rule s^2 shown below:

$$s^2 : g(i,k) \rightarrow g(j,i) \rightarrow g(j,k)$$

Informally speaking, the rule s^2 can be seen as a template (of sorts) for relating information between the terms k and i in a specific context. The application $[s^2](g(1,b))$ yields the first-order strategy $g(j,1) \rightarrow g(j,b)$, and the application $[g(j,1) \rightarrow g(j,b)](g(2,1))$ yields $g(2,b)$. In this instance, s^2 provides a vehicle for transferring data from $g(1,b)$ to $g(2,1)$.

3.1 The Table Normalization Problem

The *table normalization* problem involves the removal of indirection within the entries of a two-columned table. In this context, normalization is interesting because *constant pool normalization* within the Java Virtual Machine (JVM) can be seen as an instance of this problem (mod some real-world details). In Section 6.1 the table normalization problem is re-visited in the context of the JVM.

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 appling 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 confluent. That is, regardless of the order in which they are applied, a sequence of resolution steps will (1) always terminate and (2) always reach the same normal form.

Let us consider constructing a set of rewrite rules capable of normalizing the table shown in Figure 3.1. The most direct solution would be obtained by simply writing the following set of labeled rewrite rules:

An interesting characteristic of the rules given in Figure 3.1 is that they do not contain any variables. We will refer to rules that do not contain variables as *ground rules*. We

Index	Index or Character
1	5
2	b
3	a
4	2
5	3

Figure 1: An example of a well-formed table

$$
\begin{array}{|l|}
\hline
r_1 : 1 \rightarrow 5 \\
r_2 : 2 \rightarrow b \\
r_3 : 3 \rightarrow a \\
r_4 : 4 \rightarrow 2 \\
r_5 : 5 \rightarrow 3 \\
\hline
\end{array}
$$

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

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 3.1, when considered collectively, describe a *resolution-set* for the table given in Figure 3.1.

Note that if the application of the resolution-set in Figure 3.1 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 table given in Figure 3.1. 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.

The fixpoint application of the *resolution-set* shown in Figure 3.1 to the table in Figure 3.1 yields the table in Figure 3.1.

A drawback of the approach described thus far is that it is highly problem specific. A

Position	Position or Data
1	a
2	b
3	a
4	b
5	a

Figure 3: The normal form of the table in Figure 1

resolution-set must be explicitly constructed by hand for each table under consideration. Of course, a more generic strategy would be 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 a higher-order framework this can be achieved. This type of solution should also be expressible using the scoped dynamic rules of Stratgeo. Otherwise, in a first-order framework resolution-sets must be simulated through the techniques described in Section 2.

3.1.1 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 $[r_1, r_2, ..., r_n]$ 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:

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

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 a suitable instance of rhs if r_i does apply. If one makes this assumption about rules, then the fixpoint 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.

It is worth noting that the dynamic scoping construct of Stratego [33] could be used to realize this type of a solution. Though we have not explicitly programmed such a solution in Stratego, the sketch of such a strategy would be as follows. First develop a named (generic) rule $r : index \rightarrow data$. Then enclose this rule in a dynamic scope in which the table t is traversed and r is applied to appropriate points in t. The dynamic aspect of the dynamic scoping construct will cause instantiations of r (e.g., what we have referred to as $r_1, r_2, ..., r_5$ in Figure 3.1) to be added to the rulebase. The resulting rulebase can then be used to resolve the table after which the dynamic scope is exited and the rules are removed from the rulebase. The limitation here is the structure and application semantics of the rulebase neither of which is under the control of the user. A reasonable model of the rulebase is that it is a strategy having a nondeterministic (or possibly ordered) application semantics. For example, a dynamically created rulebase might be viewed as a strategy of the form $r_1 + r_2 + ... + r_n$ where r_i are instantiations of r. This type of a rulebase is most

appropriate for constructing a set of rewrite rules for confluent and terminating systems such as table normalization. This approach encounters problems when dealing with non-confluent nonterminating systems . What we propose in this paper 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 rulebases with the exception that the user has explicit control over their structure and application semantics.

3.1.2 A First-Order Solution

In a functional framework, table normalization could also be achieved without resorting to higher-order functions. A first-order approach would bypass the creation of the (intermediate) resolution-set altogether, favoring instead a more direct resolution of table entries. Because resolution steps may involve table entries both above and below the entry being resolved, a value (e.g., either the table itself or an accumulated values such as list of tuples) would need to be introduced for the purpose of providing information about the entries of the entire table. This value could then be passed as a parameter or embedded within a strategy and a lookup function could then be used to access the information needed to perform a resolution step on a table entry. The contextual rules of Stratego discussed in Section 2.5.1 could also be employed as a realization of this kind of idea.

Whether a first-order or higher-order approach is taken, every solution to the table resolution problem must ultimately deal with the problem of (1) finding a term d of type data, (2) using the information in the table t in order to perform a resolution step on d, and (3) computing the fixpoint of the individual resolution steps. An interesting characteristic of the higher-order approach described is that the lookup function is not explicit, but in some sense embedded in the semantics of application (i.e., the fold operation).

3.2 Generalization to a Higher-Order Strategic Framework

In the higher-order functional solution to the table normalization problem, a list $rs = [r_1, r_2, ..., r_n]$ 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 the rules r_i are sequentially composed.

$$rs_{\text{sequential}} = r_1; r_2; ...; r_n$$

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.

$$rs_{\text{nondeterministic}} = r_1 + r_2 + ... + r_n$$

Here the application of $r_1 + r_2 + ... + r_n$ 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.

$$rs_{\text{deterministic}} = r_1 +> r_2 +> ... +> r_n$$

Here the application of $r_1 +> r_2 +> ... +> r_n$ to a term t will select the leftmost r_i which can be successfully applied to t.

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.

4 Model

In this section we develop a language for strategic programming called TL. Here our focus is primarily on semantic and theoretical considerations and not necessarily on practical concerns such as the efficient implementation of constructs.

4.1 Trees

We are interested in the manipulation of terms corresponding to derivation sequences defined with respect to a given context-free grammar. Let $G = (N, T, P, S)$ 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

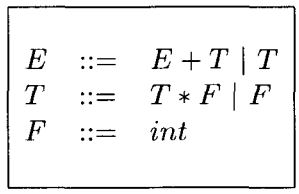


Figure 4: A BNF describing a restricted set of mathematical expressions

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

In our strategic framework, we write $B[[\alpha']]$ to denote an *instance* of the derivation $B \stackrel{+}{\Rightarrow} \alpha$ whose resulting value is a parse tree having B as its *dominating symbol.* We refer to expressions of the form $B[[\alpha']]$ as *parse expressions.* In the parse expression $B[[\alpha']]$ the string α' is an *instance* of α because nonterminal symbols in α' are constrained through the use of subscripts. We call subscripted nonterminal symbols *schema variables* or simply *variables* for short. We also consider a lone schema variable to be a parse expression (e.g., B_i). An important thing to note about schema variables is that they are typed variables and as such many only be bound to values (i.e., parse trees) derivable 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 defined by the nonterminal B. Consider a BNF grammar shown in Figure 4.1 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 $[[T + T]]$.

When the dominating symbol and specific structure of a parse expression is unimportant it will be denoted by variables of the form $t, t_1, ...$ or variables of the form $tree, tree_1, tree_2$, and so on. Parse expressions containing no schema variables are called *ground* and parse expressions containing one or more schema variables are called *non-ground.*

Within the context of rewriting or strategic programming, *trees* as described here can

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

Figure 5: The semantics of sigma distribution

and generally are viewed as *terms*. When the distinction is unimportant, we will refer to *trees* and *terms* interchangeably. However, we would like to point out that our reason for defining trees rather than terms stems from the fact that our conceptual framework as well as our tools are based on tree representations as defined above and not the term representations that are more commonly found in the literature.

At some level of abstraction the distinction between trees and terms blurs. However, there are also important distinctions between them. For example, the internal structure of a tree can be automatically completed using a parser while the internal structure of a term, in its purest sense, cannot. Automatic completion of trees assures that trees will always be well-formed entities as defined by a given grammar. In contrast, strategic frameworks based on terms typically require the internal structure of terms to be made explicit (by the user) within strategy and rule definitions.

We feel that term completion is a significant enough distinction to justify our departure from the traditional term nomenclature and representation. Tree representations have other advantages over terms, and the reverse is also true, but such discussion lies beyond the scope of this paper.

4.2 Match Equations

Matching is a fundamental operation in our framework. We will used the symbol $\ll$ adapted from the ρ-calculus [8] 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 σ binding schema variables to ground parse expressions is a solution to $t_1 \ll t_2$ if $\sigma(t_1) = t_2$ with $=$ denoting a boolean valued test for syntactic equality. Our matching framework is strictly first-order in the sense that we do not consider parse expressions that make use of variables whose values are quantified over arbitrary sets of dominating symbols (e.g., $X \in \{A, B, C\}$ in an expression of the form $X[[\alpha']]$ etc.).

A *match expression* is a boolean expression involving one or more match equations. Match expressions may be constructed using the standard boolean operators: $\wedge, \vee, \neg$. A substitution σ is a solution to a match expression m iff $\sigma(m)$ evaluates to true using the standard semantics for boolean operators in conjunction with the semantics defined in Figure 4.2.

4.3 Conditional Rewrite Rules

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

$$lhs \rightarrow rhs \ if \ E$$

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 [2] and the *where* construct found in Stratego [33]. 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 from (lhs, E). Thus we will write $lhs' \rightarrow rhs$ as a shorthand for $lhs \rightarrow rhs \ if \ E$.

The main points to note regarding the definition of conditional rewrite rules are:

1. A conditional rewrite rule defines a scope boundary.
2. Within E, match equations make matching operations explicit.
3. Rewrite conditions (e.g., normal form constraints, etc.) can be expressed in E.
4. $lhs' \rightarrow rhs$ is a shorthand for $lhs \rightarrow rhs \ if \ E$ provided $lhs' = (lhs, E)$.

4.4 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 [23] 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) $lhs' \rightarrow s^n$ is a (non-trivial) strategy of order $n + 1$, provided $n \geq 0$.
   (d) Sequential Composition: $s_1^n; s_2^n$ is a strategy of order n, provided $n > 0$.
   (e) Conditional Composition: $s_1^n | s_2^n$ is a strategy of order n, provided $n > 0$.
   (f) 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 $n = 1$.

(g) One-layer Higher-Order Generic Traversal Combinators:

i. $all\_thread\_left(s^n, \tau, \oplus)$ is a strategy where τ is a unary strategy combinator (e.g., transient) and $\oplus$ is a left-associative binary strategy combinator (e.g., sequential composition, conditional composition), provided $n > 0$.

ii. $all\_thread\_right(s^n, \tau, \oplus)$ is a strategy where τ is a unary strategy combinator (e.g., transient) and $\oplus$ is a right-associative binary strategy combinator (e.g., sequential composition, conditional composition), provided $n > 0$.

iii. $all\_broadcast(s^n, \tau, \oplus)$ is a strategy where τ is a unary strategy combinator (e.g., transient) and $\oplus$ is a binary strategy combinator, provided $n > 0$.

3. No other expressions are strategies.

4.5 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 application semantics of conditional rewrite rules in a non-failure 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.

4.5.1 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., $lhs' \rightarrow s^n$. When expressing rule and strategy application, we adopt a curried notation in the style of ML where application is a left-associative implicit operator and parenthesis 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 non-failure based application of a higher-order conditional rewrite rule.

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

It should be noted that the definition given for rule application is non-failure 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'_γ calculus where the failure of a rule to apply will yield a distinguished value *fail*.

4.5.2 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 [2] 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[34] and the S'_γ calculus [22], define 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 only if s_1 fails. Given these combinators, left-biased choice (first try s_1 and if that fails try s_2) can be defined by the strategy $s_1 + (\neg s_1; s_2)$ and right-biased choice can be defined by the strategy $(\neg s_2; s_1) + s_2$.

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 observation is straightforward since the value *fail* explicitly indicates when a rule application has failed. However, in a non-failure based framework such as ours, this observation becomes a bit more involved. One way to solve the problem is to introduce an observer predicate $observe(s, t)$ that evaluates to *true* iff 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 the application succeeded) since such a test would not be able to distinguish between the failure or success of an application of an identity-like rule (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 choosen 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 stratgegy whose application never succeeds[1]. 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.

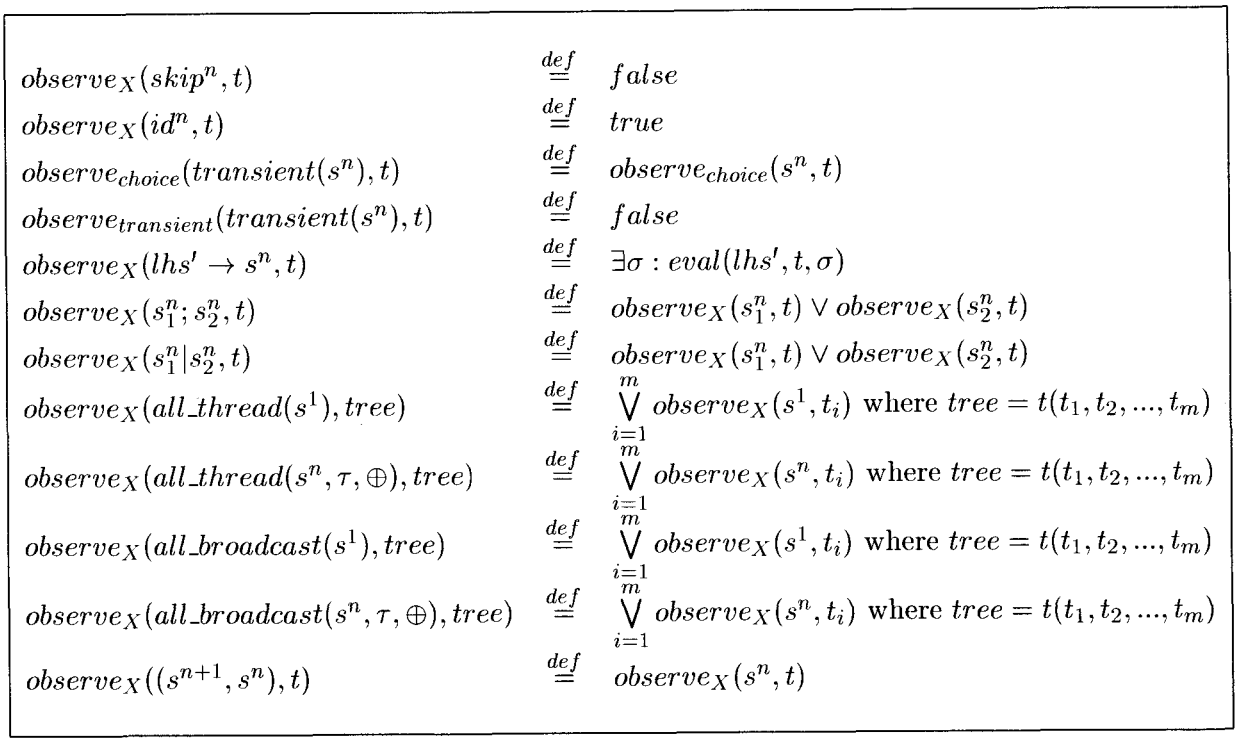
The definition of *transient* also introduces the need for two distinct observer predicates. The first predicate $observe_{choice}$ defines the semantics of applies from the perspective of the choice combinator. The second predicate $observe_{transient}$ defines the semantics of applies from the perspective of the transient combinator.

The following example illustrates the difference between the observer predicates:

$$transient(transient(s_1)|s_2)\ t$$

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 | 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. Furthermore, we would like the successful application of s_1 to t to reduce $transient(transient(s_1)|s_2)$ to $transient(skip|s_2)$ which can further be reduced to $transient(s_2)$. 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 the entire strategy $transient(transient(s_1)|s_2)$ would then reduce to *skip*. Therefore, it is essential that the outermost *transient* not be permitted to observe the application of s_1 to t.

It turns out that the definitions of $observe_{choice}$ and $observe_{transient}$ are identical for all strategies and strategy combinators except for the definition of the *transient* combinator. Thus in order to avoid duplication, Figure 4.5.2 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 definitinon 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*.



$$
\begin{array}{lcl}
observe_X(skip^n, t) & \stackrel{def}{=} & false \\
observe_X(id^n, t) & \stackrel{def}{=} & true \\
observe_{choice}(transient(s^n), t) & \stackrel{def}{=} & observe_{choice}(s^n, t) \\
observe_{transient}(transient(s^n), t) & \stackrel{def}{=} & false \\
observe_X(lhs' \rightarrow s^n, t) & \stackrel{def}{=} & \exists \sigma : eval(lhs', t, \sigma) \\
observe_X(s_1^n; s_2^n, t) & \stackrel{def}{=} & observe_X(s_1^n, t) \vee observe_X(s_2^n, t) \\
observe_X(s_1^n | s_2^n, t) & \stackrel{def}{=} & observe_X(s_1^n, t) \vee observe_X(s_2^n, t) \\
observe_X(all\_thread(s^1), tree) & \stackrel{def}{=} & \bigvee_{i=1}^{m} observe_X(s^1, t_i) \text{ where } tree = t(t_1, t_2, ..., t_m) \\
observe_X(all\_thread(s^n, \tau, \oplus), tree) & \stackrel{def}{=} & \bigvee_{i=1}^{m} observe_X(s^n, t_i) \text{ where } tree = t(t_1, t_2, ..., t_m) \\
observe_X(all\_broadcast(s^1), tree) & \stackrel{def}{=} & \bigvee_{i=1}^{m} observe_X(s^1, t_i) \text{ where } tree = t(t_1, t_2, ..., t_m) \\
observe_X(all\_broadcast(s^n, \tau, \oplus), tree) & \stackrel{def}{=} & \bigvee_{i=1}^{m} observe_X(s^n, t_i) \text{ where } tree = t(t_1, t_2, ..., t_m) \\
observe_X((s^{n+1}, s^n), t) & \stackrel{def}{=} & observe_X(s^n, t)
\end{array}
$$

Figure 6: The semantics of observation

Note that in a non-failure 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 $s_1; s_2; ...; s_n$ applies to t, it is sufficient to find a single strategy s_i that applies to t. Furthermore, $transient(s^n)$ is a strategy whose application can never be observed by $observe_{transient}$ but whose application can be observed by $observe_{choice}$.

4.5.3 The Semantics of Basic Strategic Combinators

In TL, the application of all strategies 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 reasons:

1. 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.
2. All strategies must be homogeneous with respect to order (e.g., an order n strategy may not have components that are order m where $m \neq n$).

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 operation to change the value of the strategy being applied. 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' which we will denote as follows: $transient(s^1)t = (skip^1, t')$. 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.

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

$$(lhs' \rightarrow s^n)\ t \quad \stackrel{def}{=} \begin{cases} ((lhs' \rightarrow s^n), \sigma(s^n)) & \text{if } observe_{choice}(lhs', t) \wedge eval(lhs', t, \sigma) \\ ((lhs' \rightarrow s^n), t) & \text{if } \neg observe_{choice}(lhs', t) \wedge n = 0 \\ ((lhs' \rightarrow s^n), skip^n) & \text{if } \neg observe_{choice}(lhs', t) \wedge n > 0 \end{cases}$$

$$transient(s^n)\ t \stackrel{def}{=} \begin{cases} (skip^n, s^n\ t) & \text{if } observe_{transient}(s^n, t) \\ (\hat{s}^n, s^{n-1}) & \text{if } \neg observe_{transient}(s^n, t) \wedge (\hat{s}^n, s^{n-1}) = s^n\ t \end{cases}$$

[1]Note that *skip*, the strategy which never applies, is the dual of *id*, the strategy which always applies.

$$(s_1^n; s_2^n)\ t \quad \stackrel{def}{=} \begin{cases} ((\hat{s}_1^1; \hat{s}_2^1), t'') & \text{if } n = 1 \wedge (\hat{s}_1^1, t') = s_1^n\ t \wedge (\hat{s}_2^1, t'') = s_2^n\ t' \\ ((\hat{s}_1^n; \hat{s}_2^n), (s_1^{n-1}; s_2^{n-1})) & \text{if } n > 1 \wedge (\hat{s}_1^n, s_1^{n-1}) = s_1^n\ t \wedge (\hat{s}_2^n, s_2^{n-1}) = s_2^n\ t \end{cases}$$

$$(s_1^n | s_2^n)\ t \quad \stackrel{def}{=} \begin{cases} ((\hat{s}_1^n | s_2^n), s^{n-1}) & \text{if } observe_{choice}(s_1^n, t) \wedge (\hat{s}_1^n, s^{n-1}) = s_1^n\ t \\ ((s_1^n | \hat{s}_2^n), s^{n-1}) & \text{if } \neg observe_{choice}(s_1^n, t) \wedge (\hat{s}_2^n, s^{n-1}) = s_2^n\ t \end{cases}$$

$$fix(s_0^1)\ t \quad \stackrel{def}{=} \begin{cases} (\hat{s}^1, t'') & \text{if } (observe_{transient}(s_0^1, t) \vee observe_{choice}(s_0^1, t)) \\ & \wedge (s_1^1, t') = (s_0^1\ t) \wedge (\hat{s}^1, t'') = fix(s_1^1)\ t' \\ (s_0^1, t) & \text{if } \neg(observe_{transient}(s_0^1, t) \vee observe_{choice}(s_0^1, t)) \end{cases}$$

4.5.4 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[13] in particular. A second approach is to provide a fixed set of generic traversals (e.g., topdown, bottomup, etc.) which can then be used to define various rewriting strategies. This approach has been taken in a system in which ASF+SDF has been extended with a fixed set of generic term traversals[4], and also by *Transformation Factories*, a similar ASF+SDF based system whose purpose is to generate components for software renovation factories[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[34], the S'_γ calculus[22], 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. In Stratego, this combinator is called *all*. In the S'_γ calculus it is denoted by the symbol $\Box$.

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

4.5.5 The Semantics of Higher-Order Generic Traversal Combinators

The higher-order generic one-layer traversals described here are unique to TL. They are similar but not identical to hylomorphisms over rose[2] trees found in functional programming frameworks [26][27]. 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.

[2]A rose tree is a multiway branching tree.

$all\_thread\_left(s_0^n, \tau, \oplus)\ tree \stackrel{def}{=}$

(s_m^n, s^{n-1})
where $tree = t(t_1, t_2, ..., t_m)$
and $(s_1^n, s_1^{n-1}) = s_0^n\ t_1$
$(s_2^n, s_2^{n-1}) = s_1^n\ t_2$
...
$(s_m^n, s_m^{n-1}) = s_{m-1}^n\ t_m$
and
$s^{n-1} = \tau(s_1^{n-1}) \oplus \tau(s_2^{n-1}) \oplus ... \oplus \tau(s_m^{n-1})$
where $\oplus$ is a left-associative binary infix combinator such as ; or | and τ is a strategy such as *transient*.

$all\_thread\_right(s_0^n, \tau, \oplus)\ tree \stackrel{def}{=}$

(s_m^n, s^{n-1})
where $tree = t(t_1, t_2, ..., t_m)$
and $(s_1^n, s_1^{n-1}) = s_0^n\ t_m$
$(s_2^n, s_2^{n-1}) = s_1^n\ t_{m-1}$
...
$(s_m^n, s_m^{n-1}) = s_{m-1}^n\ t_1$
and
$s^{n-1} = \tau(s_1^{n-1}) \oplus \tau(s_2^{n-1}) \oplus ... \oplus \tau(s_m^{n-1})$
where $\oplus$ is a right-associative binary infix combinator such as ; or | and τ is a strategy such as *transient*.

$all\_broadcast(s^n, \tau, \oplus)\ tree \stackrel{def}{=}$

(s^n, s^{n-1})
where $tree = t(t_1, t_2, ..., t_m)$
and $(\_, s_1^{n-1}) = s^n\ t_1$
$(\_, s_2^{n-1}) = s^n\ t_2$
...
$(\_, s_m^{n-1}) = s^n\ t_m$
and
$s^{n-1} = \tau(s_1^{n-1}) \oplus \tau(s_2^{n-1}) \oplus ... \oplus \tau(s_m^{n-1})$
where $\oplus$ is a binary infix combinator such as ; or | and τ is a strategy.

4.5.6 Coda

In the definitions above, we have glossed over some low-level details regarding type consistency. For example, the equations described in Section 4.2 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.

We now make some basic observations in the form of theorems. The purpose of these theorems is to help the reader gain a better understanding of TL by explicitly articulating some basic properties many of which the reader may already be aware. We leave the proofs of these theorems to the reader.

Theorem 1 $skip^n \mid s^n = s^n$

Theorem 2 $s^n \mid skip^n = s^n$

Theorem 3 $skip^n \; ; \; s^n = s^n$

Theorem 4 $s^n \; ; \; skip^n = s^n$

Theorem 5 $transient(skip^n) = skip^n$

Theorem 6 *For any strategy* s^1 *that is free from transients:*

$$all\_thread\_left(s^1) = all\_thread\_right(s^1) = all\_broadcast(s^1)$$

Theorem 7 *For any strategy* s^n *that is free from transients:*

$$all\_thread\_left(s^n, \tau, \oplus) = \; all\_thread\_right(s^n, \tau, \oplus) = all\_broadcast(s^n, \tau, \oplus)$$

4.5.7 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 4.5.7.

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.

When writing TL programs, we discourage the use of recursive definitions involving parameters except for defining strategies that are completely generic (e.g., topdown, bottomup, etc.).

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 | λ 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 4.4

Figure 7: A partial BNF for strategy definitions

4.5.8 Some Generic Recursive TL Strategies

Strategy			Comment
$\mathcal{I}(s)$	$=$	s	The identity strategy.
$tdl\_thread(s_0^1)$	$=$	$\lambda\ t.$ let $(s_1^1, t') = s_0^1\ t$ in $all\_thread\_left(tdl\_thread(s_1^1))\ t'$ end	A complete threaded top-down left-to-right traversal
$tdr\_thread(s^1)$	$=$	$\lambda\ t.$ let $(s_1^1, t') = s_0^1\ t$ in $all\_thread\_right(tdr\_thread(s_1^1))\ t'$ end	A complete threaded top-down right-to-left traversal
$td\_broadcast(s^1)$	$=$	$\lambda\ t.$ let $(s_1^1, t') = s_0^1\ t$ in $all\_broadcast(td\_broadcast(s_1^1))\ t'$ end	A complete broadcast top-down traversal
$bul\_thread(s^1)$	$=$	$\lambda\ t.$ let $(s_1^1, t') = all\_thread\_left(bul\_thread(s^1))\ t$ in $s_1^1\ t'$ end	A complete threaded bottom-up left-to-right traversal
$bur\_thread(s^1)$	$=$	$\lambda\ t.$ let $(s_1^1, t') = all\_thread\_right(bur\_thread(s^1))\ t$ in $s_1^1\ t'$ end	A complete threaded bottom-up left-to-right traversal

4.5.9 Some Generic Higher-Order Strategies in TL

Strategy	Comment
$all\_seq\_tdl(s_0^n) =$ $\lambda\ t.$ let $(s_1^n, s^{n-1}) = s_0^n\ t$ in $s^{n-1}\ ;\ (all\_thread\_left(all\_seq\_tdl(s_1^n), I, ;)\ t)$ end	Construct a sequential composition of the result of applying s_0^n in accordance with a threaded top-down left-to-right traversal
$all\_seq\_bul(s_0^n) =$ $\lambda\ t.$ let $(s_1^n, s_1^{n-1}) = all\_thread\_left(all\_seq\_bul(s_0^n), I, ;)\ t$ $(s_2^n, s_2^{n-1}) = s_1^n\ t$ in $s_1^{n-1}\ ;\ s_2^{n-1}$ end	Construct a sequential composition of the result of applying s_0^n in accordance with a threaded bottom-up left-to-right traversal
$all\_cond\_tdl(s^n) =$ $\lambda\ t.$ let $(s_1^n, s^{n-1}) = s_0^n\ t$ in $s^{n-1}\ \mid\ (all\_thread\_left(all\_cond\_tdl(s^n), I, \mid)\ t)$ end	Construct a conditional composition of s_0^n in accordance with a threaded top-down left-to-right traversal
$all\_cond\_bul(s^n) =$ $\lambda\ t.$ let $(s_1^n, s_1^{n-1}) = all\_thread\_left(all\_cond\_bul(s_0^n), I, \mid)\ t$ $(s_2^n, s_2^{n-1}) = s_1^n\ t$ in $s_1^{n-1}\ \mid\ s_2^{n-1}$ end	Construct a conditional composition of s_0^n in accordance with a threaded bottom-up left-to-right traversal

4.5.10 A TL Implementation of Some Standard First-Order Generic Strategies

Strategy			Comment
$TD(s^1)$	:	$tdl\_thread(s^1)$	Will apply s^1 in a top-down left-to-right fashion provided s^1 is transient-free (i.e., contains no nested transients)
$BU(s^1)$	:	$bul\_thread(s^1)$	Will apply s^1 in a bottom-up left-to-right fashion provided s^1 is transient-free
$onceTD(s^1)$	:	$TD(transient(s^1))$	Will apply s^1 at most one time somewhere in a term provided s^1 is transient-free
$onceTD(s^1)$	:	$BU(transient(s^1))$	Will apply s^1 at most one time somewhere in a term provided s^1 is transient-free
$stopTD(s^1)$	:	$td\_broadcast(transient(s^1))$	Will apply s^1 at most one time on any path from a subterm to its root. E.g., if s^1 applies to t then s^1 did not apply to any ancestor of t

5 Examples: Set and Sequence Operations

Here, we consider three abstract strategies, union_s, intersect_s, and zip_s that can be applied to sets, sequences, and multi-sets. Our experiences lead us to believe that the union_s, intersect_s, and zip_s strategies described below lie at the core of a number of common program transformation objectives. Variations of the union_s strategy can be used as the basis for constant propagation, variable renaming, Java constant pool normalization, as well as field distribution and method method table construction in Java classfiles.

In the grammar shown in Figure 5 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 quoted. Figure 5 defines some strategies that perform simple operations such as adding an element to an empty list of elements, etc. Figure 5 presents higher-order strategies defining the operations union_s, intersection_s, and zip_s. These strategies when properly instantiated, will perform their corresponding set/sequence operations. Figure 5 defines strategies that properly instantiate strategies from Figure 5 and then apply the resulting instantiation to a term.

5.1 Union

The *union_s* strategy is a higher-order strategy that when given an element as described by the pattern *elements*[[$element_1$ $elements_1$]] will produce a transient strategy consisting of the conditional composition of the strategy *keep*($element_1$) with the strategy *add*($element_1$). The expression *all_cond_tdl union_s* set_1 is used to traverse set_1 and produce a sequential composition consisting of an appropriately instantiated transient strategy for every element in the set. For example, suppose $set_1 = \{x1, x2, x3, x4\}$. The result of *all_cond_tdl union_s* set_1 will be:

```
set_expr    ::= set set_op set | set
set         ::= "{" elements "}"

elements    ::= element elements | ()
element     ::= <id> | "(" <id> <id> ")"

set_op      ::= "union" | "intersect" | "zip"
```

Figure 8: A BNF describing set/sequence expressions

$$
\begin{array}{lcl}
keep(element_1) & : & elements[[element_1\ elements_2]] \rightarrow elements[[element_1\ elements_2]] \\
add(element_1) & : & elements[[\ ]] \rightarrow elements[[element_1]] \\
remove & : & elements[[element_1\ elements_2]] \rightarrow elements_2 \\
tuple(element_1) & : & elements[[element_2\ elements_2]] \rightarrow elements[[(element_1\ element_2)\ elements_2]]
\end{array}
$$

Figure 9: Some basic abstractions

$$
\begin{array}{lcl}
union\_s & : & elements[[\ element_1\ elements_1\ ]] \rightarrow transient(keep(element_1) \mid add(element_1)) \\
intersect\_s & : & elements[[\ element_1\ elements_1\ ]] \rightarrow transient(keep(element_1)) \\
zip\_s & : & elements[[\ element_1\ elements_1\ ]] \rightarrow transient(tuple(element_1))
\end{array}
$$

Figure 10: Second-order strategies realizing set/sequence operations

$$
\begin{array}{lcl}
make\_union & : & set\_expr[[set_1\ union\ set_2]] \\
 & & \rightarrow TD(all\_cond\_tdl\ union\_s\ set_1)\ set_2 \\
 & & \\
make\_intersect & : & set\_expr[[set_1\ intersect\ set_2]] \\
 & & \rightarrow TD((all\_cond\_tdl\ intersect\_s\ set_1) | remove)\ set_2 \\
 & & \\
make\_zip & : & set\_expr[[set_1\ zip\ set_2]] \\
 & & \rightarrow TD(all\_cond\_tdl\ zip\_s\ set_1)\ set_2
\end{array}
$$

Figure 11: Instantiation and application of second-order strategies to terms

$\text{transient}(elements[[x1\ elements_2]] \rightarrow elements[[x1\ elements_2]] \mid elements[[\,]] \rightarrow elements[[x1]])$
$\mid$
$\text{transient}(elements[[x2\ elements_2]] \rightarrow elements[[x2\ elements_2]] \mid elements[[\,]] \rightarrow elements[[x2]])$
$\mid$
$\text{transient}(elements[[x3\ elements_2]] \rightarrow elements[[x3\ elements_2]] \mid elements[[\,]] \rightarrow elements[[x3]])$
$\mid$
$\text{transient}(elements[[x4\ elements_2]] \rightarrow elements[[x4\ elements_2]] \mid elements[[\,]] \rightarrow elements[[x4]])$

Let s denote this first-order strategy. The strategy expression $TD\ s\ set_2$ now applies s to every element in set_2. Whenever a duplicate element is encountered a strategy/rule of the form $elements[[element_1\ elements_2]] \rightarrow elements[[element_1\ elements_2]]$ will apply after which the transient attribute will cause the strategy/rule to be removed from s. For example, suppose $set_2 = \{y1, x2, x3, y2\}$. The application of s to the first element in set_2 will leave s unchanged. However, after the application of s to the second element in set_2, s is changed to s' and has the value:

$\text{transient}(elements[[x1\ elements_2]] \rightarrow elements[[x1\ elements_2]] \mid elements[[\,]] \rightarrow elements[[x1]])$
$\mid$
$\text{transient}(elements[[x3\ elements_2]] \rightarrow elements[[x3\ elements_2]] \mid elements[[\,]] \rightarrow elements[[x3]])$
$\mid$
$\text{transient}(elements[[x4\ elements_2]] \rightarrow elements[[x4\ elements_2]] \mid elements[[\,]] \rightarrow elements[[x4]])$

After the application of s' to the third element in set_2, s' will be changed to s'' and have the value:

$\text{transient}(elements[[x1\ elements_2]] \rightarrow elements[[x1\ elements_2]] \mid elements[[\,]] \rightarrow elements[[x1]])$
$\mid$
$\text{transient}(elements[[x4\ elements_2]] \rightarrow elements[[x4\ elements_2]] \mid elements[[\,]] \rightarrow elements[[x4]])$

After the last element of set_2 has been traversed, the $add(element_1)$ strategy of all the remaning transients will cause the elements $x1$ and $x4$ to be added to the set after which they too will be removed from the strategy. The resulting set is the union of set_1 and set_2 which is:

$$\{y_1, x_2, x_3, y_2, x_1, x_4\}$$

5.2 Intersection

The *intersect_s* strategy is a higher-order strategy that when given an element as described by the pattern $elements[[\ element_1\ elements_1\ ]]$ will produce a transient instance of the strategy $keep(element_1)$. The expression *all_cond_tdl intersect_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 the set. To this resulting strategy the *remove*

strategy is then conditionally added, so the expression becomes $(all\_cond\_tdl\ intersect\_s\ set_1)|remove$. Let us examine what strategy gets created if $set_1 = \{x1, x2, x3, x4\}$. The result of $(all\_cond\_tdl\ intersect\_s\ set_1)|remove$ will be:

transient($elements[[x1\ elements_2]] \rightarrow elements[[x1\ elements_2]]$)
|
transient($elements[[x2\ elements_2]] \rightarrow elements[[x2\ elements_2]]$)
|
transient($elements[[x3\ elements_2]] \rightarrow elements[[x3\ elements_2]]$)
|
transient($elements[[x4\ elements_2]] \rightarrow elements[[x4\ elements_2]]$)
|
remove

Notice that the *remove* is not transient and is not created as a result of a higher-order application. In particular, $element_1$ in *remove* remains an uninstantiated schema variable. Let s denote the first-order strategy shown above. The strategy expression $TD\ s\ set_2$ now applies s to every element in set_2. Whenever a duplicate element is encountered a strategy/rule of the form $elements[[element_1\ elements_2]] \rightarrow elements[[element_1\ elements_2]]$ will apply after which the transient attribute will cause the strategy/rule to be removed from s. Furthermore, the conditional composition combinator will only enable *remove* to be executed if all of the prior transients fail. The failure of all transients implies that the element under consideration is not part of the intersection and should therefore be removed. For example, suppose $set_2 = \{y1, x2, x3, y2\}$. The application of s to the first element in set_2 will leave s unchanged, but will cause $y1$ to be removed from set_2. We will use set_2' to denote the new value of set_2. The application of s to the second element in set_2' will cause s to be changed but will leave set_2' unchanged. The new value of s which we denote s' will have the value:

transient($elements[[x1\ elements_2]] \rightarrow elements[[x1\ elements_2]]$)
|
transient($elements[[x3\ elements_2]] \rightarrow elements[[x3\ elements_2]]$)
|
transient($elements[[x4\ elements_2]] \rightarrow elements[[x4\ elements_2]]$)
|
remove

After the application of s' to the third element in set_2', s' will again be changed to s'' and have the value:

transient($elements[[x1\ elements_2]] \rightarrow elements[[x1\ elements_2]]$)
|
transient($elements[[x4\ elements_2]] \rightarrow elements[[x4\ elements_2]]$)

|
remove

This strategy when applied to the element $y2$ will cause it to be removed from set_2'. The resulting set is the intersection of set_1 and set_2 which is:

$$\{x_2, x_3\}$$

5.3 Zip

The *zip_s* strategy is a higher-order strategy that when given an element as described by the pattern $elements[[\ element_1\ elements_1\ ]]$ will produce a transient instance of the strategy $tuple(element_1)$. The expression *all_cond_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. Let us examine what strategy gets created if $set_1 = \{x1, x2, x3, x4\}$. The result of *all_cond_tdl intersect_s* set_1 will be:

transient($elements[[element_2\ elements_2]] \rightarrow elements[[(x1\ element_2)\ elements_2]]$)
|
transient($elements[[element_2\ elements_2]] \rightarrow elements[[(x2\ element_2)\ elements_2]]$)
|
transient($elements[[element_2\ elements_2]] \rightarrow elements[[(x3\ element_2)\ elements_2]]$)
|
transient($elements[[element_2\ elements_2]] \rightarrow elements[[(x4\ element_2)\ elements_2]]$)

Let s denote the first-order strategy shown above. The strategy $TD\ s\ set_2$ now applies s to every element in set_2. Whenever an element is encountered the first transient strategy will apply after which the transient attribute will cause the strategy to be removed from s. For example, suppose $set_2 = \{y1, y2, y3, y4\}$. The application of s to the first element in set_2 will yield the tuple $(x1\ y1)$. Let s' denote the new value of s that results. The strategy s' now contains the transients:

transient($elements[[element_2\ elements_2]] \rightarrow elements[[(x2\ element_2)\ elements_2]]$)
|
transient($elements[[element_2\ elements_2]] \rightarrow elements[[(x3\ element_2)\ elements_2]]$)
|
transient($elements[[element_2\ elements_2]] \rightarrow elements[[(x4\ element_2)\ elements_2]]$)

The application of s' to the next element in set_2 will yield the tuple $(x2\ y2)$, after which the applied transient is again removed. This continues on producing the final set:

$$\{\ (x1\ y1)\ (x2\ y2)\ (x3\ y3)\ (x4\ y4)\ \}$$

6 A Classloader 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)[25]. An application program for the SSP is called a *ROM image* and consists of a collection of classfile-like structures 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 classfiles across a network). The closed nature of the SSP's execution environment enables the classloading activities of the JVM to be performed statically. Under these conditions, the functionality of the classloader is well-suited to a strategic implementation.

In the discussion that follows, we assume that an *application* consists of one or more Java *classfiles* and that Java classfiles have the structure defined in [24] subject to some minor restructuring to facilitate strategic objectives. For the purposes of this discussion the important things to know about classfiles 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.
4. A *methods section* containing all of the methods explicitly defined in the class.

Activities that can be performed statically include (1) *constant pool normalization* – which consists of removing indirection from constant pool entries (2) *field distribution* – which consists of distributing field address information across all constant pool entries within an application, and (3) *method table construction* – which concerns itself with the construction of method tables subject to a set of constraints (see Section 6.3).

The results presented here have all been implemented in a system called HATS which is described in Section 7.

6.1 Constant Pool Normalization

In this section we will look at how dynamic 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 3.1.

Figure 6.1 gives an example in human readable form of the kind of information found in the constant pool of a Java classfile. In particular, the contents of a constant pool entry may be a utf8 (a string) or one or two indexes to other constant pool entries. For example, the fourth entry describes a field whose class name index can be found at entry 2 and whose (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.

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

Figure 12: Unresolved constant pool entries

Unresolved		Partially Resolved		Resolved
`class_index = 2`	→	`name_index = 1`	→	`animal`
`name_and_type_index = 3`	→	`name_index = 5`	→	`x`
		`descriptor_index = 6`	→	`I`

Figure 13: Index resolution sequence

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 normalization of entry 4 in Figure 6.1 is shown in Figure 6.1.

After constant pool normalization, utf8 entries are no longer needed and can be removed yielding a reduced constant pool consisting only of *relevant* entries. Figure 6.1 shows the normalized relevant constant pool entries for the constant pool given in Figure 6.1.

Our approach to solving the index resolution problem consists of a combination of language redesign together with supporting canonical forms. Shown in Figure 6.1 is a redesigned grammar fragment describing the structure of the constant pool within a Java classfile. The naming conventions have been taken from the JVM specification[24] 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.

Constant pool normalization can be achieved by the exhaustive application of proper

Original Index	Original Type	Contents
2	`constant_class_info`	`animal`
4	`constant_fieldref_info`	`animal x I`

Figure 14: Normalized relevant constant pool entries

```
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
```

Figure 15: A redesigned grammar fragment for the Java classfile structure

$$
\begin{array}{lcl}
resolve\_index(index_1, utf8_1) & : & data[[index_1]] \rightarrow data[[utf8_1]] \\
resolve\_nt\_index(index_1, utf8_1, utf8_2) & : & name\_and\_type[[index_1]] \\
 & & \rightarrow name\_and\_type[[utf8_1\ utf8_2]]
\end{array}
$$

Figure 16: Second-order strategies capturing resolution steps

$$
\begin{array}{lcl}
resolve\_data & : & cp\_info[[index_1\ constant\_utf8\_info_1]] \rightarrow resolve\_index(index_1, utf8_1) \\
 & & \text{if } constant\_utf8\_info_1 \gg constant\_utf8\_info[[utf8_1]] \\
 & & \\
resolve\_nt & : & cp\_info[[index_1\ constant\_name\_and\_type\_info_1]] \\
 & & \rightarrow resolve\_nt\_index(index_1, utf8_1, utf8_2) \\
 & & \text{if } constant\_name\_and\_type\_info_1 \gg constant\_name\_and\_type\_info[[utf8_1\ utf8_2]] \\
 & & \\
resolve & : & ClassFile_0 \\
 & & \rightarrow TD(all\_cond\_tdl(resolve\_data; resolve\_nt) ClassFile_0) ClassFile_0
\end{array}
$$

Figure 17: Strategies for instantiating and applying resolution strategies

instantiations of the two strategies shown in Figure 6.1:

The strategy *resolve_index* describes how a data *index* can be rewritten to a *utf8*, and the strategy *resolve_nt_index* describes how a name-and-type *index* can be rewritten to a pair of *utf8* elements. The grammar in Figure 6.1 has been redesigned in such a way that these resolutions are the only ones that need to be considered. The dynamic strategies used to instantiate *resolve_index* and *resolve_nt_index* as well as normalize the constant pool are given in Figure 6.1.

For a more detailed discussion of both the SSP project and transformation in this setting see [39].

6.2 Field Distribution

In this section we look at the problem of distributing field offset information among the constant pool entries in an 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 every *fields section* has been annotated with an appropriate offset address and every static field has been annotated with an appropriate absolute address. Though this article does not discuss the strategies needed to compute such addresses, we would like to mention

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"

Figure 18: A simplified Java grammar

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 I* denote an entry occurring in the normalized relevant 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 replacement

$$animal\ x\ I \rightarrow :0004\ I$$

to be applied to all constant pool entries having the value *animal x I*. Note that a constant pool entry such as *animal x I* need not be restricted to the class *animal* in which the field *x* is declared, but 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 in the context of the simplified Java grammar given in Figure 6.2.

Given this grammar, the an application consisting of classfiles for the classes A, B, and C could be expressed as follows:

```
{ 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  } }
```

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 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.

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

Figure 19: A higher-order strategic solution to the field distribution problem

The result after *field distribution* would be:

{ 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 } }

Field distribution can be realized by the strategies shown in Figure 6.2:

Let us take a closer look at how the *Field_Distribution* strategy shown in Figure 6.2 solves the *field distribution* problem. Recall that according to the BNF in Figure 6.2, an *application* consists of a list of *classfile*s and each *classfile* 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 $all\_cond\_tdl\ Class\ application_1$ accomplishes this and yields a strategy of the form:

$$Class_1 \oplus Class_2 \oplus ... \oplus\ Class_n$$

Within each $Class_i$ instance we need to produce an instance of the *Field* strategy for every *f_entry* in the *field_section*. The strategic expression $all\_cond\_tdl\ (Field\ class_i)\ field\_section_i$ accomplishes this and yields a strategy of the form:

$$Field_{i.1} \oplus Field_{i.2} \oplus ... \oplus\ Field_{i.m}$$

The resulting strategy contains the information needed to resolve every *cp_entry* in $application_1$. A full traversal that applies this strategy to the term $application_1$ solves the *field distribution* problem.

6.3 Method Table Construction for Java Classfile Hierarchies

The strategies given for *constant pool normalization* and *field distribution* do not involve the *transient combinator*. In practice however, the transient combinator is widely used.

Within the context of the classloader problem, *method table construction* is an example of an activity that involves the use of transient strategies. For the sake of brevity, we only present a sketch of a strategic approach for *method table construction.*

When implementing a Java Virtual Machine (JVM), method tables are often used as a mechanism for indirectly providing access to the methods associated with an object [32]. Each classfile has one method table whose entries contain information about particular methods such as the address of the first bytecode of a method and the address of the constant pool corresponding to a method. In order to provide correct information at runtime, it is sufficient for method tables within a class hierarchy to satisfy the properties given below.

Let s_m denote the signature (i.e., the name and descriptor) of method m. Let $e[C, s_m]$ denote a method table entry corresponding to a method having a signature s_m that is defined in class C. Let T_C denote the method table for the class C, and let $\prec$ denote a reflexive, transitive sub-type relationship between classes as defined by the Java *extends* directive. For example, given two classes B and C, if $C \prec B$ then either $C = B$ or C is a descendent of B within the inheritance hierarchy.

1. For every inherited method m that is **not** redefined in C there must be a corresponding entry of the form $e[B, s_m] \in T_C$ where the class B denotes most recent ancestor of C where m is defined.
2. For every method m (re)defined in a class C there must be a corresponding entry $e[C, s_m] \in T_C$.
3. The method table for the class C may only contain entries corresponding to inherited methods or methods that have been defined in C.
4. $\forall D_1, D_2,\ s_m : \exists B, C, i, j : T_{D_1}[i] = e[B, s_m] \wedge T_{D_2}[j] = e[C, s_m] \wedge D_1 \prec D_2 \rightarrow i = j$. That is, within an inheritance hierarchy table entries corresponding to the signature s_m must reside at the same location (index) in all tables containing such an entry. For example, if information for the method *foo* resides at the second entry of a method table, then all classes inheriting or redefining *foo* must also have the information for *foo* as the second entry in their method tables.

The properties above permit method tables to be constructed in a concatenated fashion provided that entries associated with redefined methods destructively overwrite the corresponding inherited method table entries (e.g., the method table entry for the new definition of *foo* overwrites the method table entry for the old definition *foo*). A variation of the *union_s* strategy can be employed to construct method tables adhering to the given consistency properties. One simply needs to change the *keep* strategy to a *replace* strategy that overwrites older method information with more recent method information. Using *replace* one now treats the method table of the parent class as a set, and takes the union of that set with the set of methods declared in the child class. Notice that all new methods (those that do not overwrite inherited methods) declared in the child class will be placed at the end of the method table, which is exactly how method tables should be constructed.

7 HATS: A Restricted Implementation of TL

HATS[37][38] 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 prettyprinter.

A domain of discourse can be defined by a context-free grammar. The HATS parser supports precedence and associativity of operators and grammar productions. It allows the user to describe a context-free grammars using an extended-BNF notation. The parser is an LR parser with the capability to do backtracking as needed in order to resolve local ambiguities. One can think of such a parser as an LR(K) parser for arbitrary K. Backtracking brings the parser's capability close to that of a scannerless generalized LR parser[31][6]. TL programs are executed by an interpreter written in ML and the output can be formatted using a powerful variation of abstract prettyprinting[7][30]. HATS runs on Windows NT/2000/XP and Unix-based platforms and is freely available at:

http://faculty.ist.unomaha.edu/winter/hats-uno/HATSWEB/index.html

The strategic language supported by HATS is a restriction of TL because (1) the combinators for sub-term traversals are not available to the programmer, (2) the definition of recursive user-defined strategies is not supported, and (3) higher-order combinators such as all_thread_left, etc. are not supported in a fully generalized fashion. At present the programmer is provided with a fixed set of generic first-order as well as higher-order traversals. Figure 7 shows some of the strategic constructs supported in HATS as well as their TL counterparts.

All of the examples discussed in this article have been implemented in HATS.

8 Related Work

8.1 Stratego

Stratego has two constructs related to the higher-order strategies presented in the paper: *contextual rules* and *scoped dynamic rewrite rules.*

In [35], contextual rules are used to distribute data within a term structure. Contextual rules can be seen as a first-order cousin of the higher-order dynamic rules presented in this paper. In a contextual rule one constructs a term in which semantically related data is enclosed within square brackets. Operationally, a nested traversal is employed to search for a set of terms satisfying the contextual rule. Contextual rules work well when their evaluation enables incremental progress to be made with respect to the overall strategic objective. That is, when each new evaluation of the contextual rule yeilds the next semantically related set of terms. In such a setting, the evaluation of the contextual rule inherently implements a

HATS Construct		TL Equivalent
$transient(s^n)$	$\equiv$	$transient\ s^n$
$s_1^n; s_2^n$	$\equiv$	$s_1^n; s_2^n$
$s_1^n \mid s_2^n$	$\equiv$	$s_1^n \mid s_2^n$
$once\_thread(pre\_order\_left, t, s)$	$\equiv$	$tdl\_thread\ s\ t$
$once\_thread(pre\_order\_right, t, s)$	$\equiv$	$tdr\_thread\ s\ t$
$once\_thread(post\_order\_left, t, s)$	$\equiv$	$bul\_thread\ s\ t$
$once\_thread(post\_order\_right, t, s)$	$\equiv$	$bur\_thread\ s\ t$
$once\_broadcast(pre\_order, t, s)$	$\equiv$	$td\_broadcast\ s\ t$
$eval(pre\_order\_left, cond, s^n, t)$	$\equiv$	$all\_cond\_tdl(s^n, \mathcal{I}, \mid)t$
$eval(post\_order\_left, cond, s^n, t)$	$\equiv$	$all\_cond\_bul(s^n, \mathcal{I}, \mid)t$
$eval(pre\_order\_right, seq, s^n, t)$	$\equiv$	$all\_seq\_tdl(s^n, \mathcal{I}, ;)t$
$eval(post\_order\_right, seq, s^n, t)$	$\equiv$	$all\_seq\_bul(s^n, \mathcal{I}, ;)t$

Figure 20: The constructs supported in HATS

counter of sorts. Conversely, contextual rules would not be appropriate in situations where the evaluation of contextual rules could contain cycles. The transient combinator describe in this article also can be seen as implementing a counter. However, the counter is explicit and lifted from the term-level to the strategy-level.

In [33], 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 program 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 rulebase. 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 {| ... |}.

Primary differences between our approach and the scoped dynamic rules described in [33] are the following:

1. In our approach, we view the rulebase as a strategy that is created dynamically. The $\oplus$ combinator provides the user explicit control over the structure of this strategy.

2. The transient combinator has no direct analogy within scoped dynamic rewrite rules.

8.2 The ρ-calculus

The ρ-calculus[8] is a rewriting framework in which the distinction between a rule and the term to which a rule is applied is blurred. Both the rule and the term are considered ρ-terms. This uniform treatment is reminiscent of the relationship between functions and terms in the λ-calculus. Similar to the λ-calculus, in the ρ-calculus there are no restrictions regarding variable occurrences within a term. In particular, free variables may be introduced on the right-hand side of a rule. In fact, the right-hand side of a rule may itself be a rule, seamlessly opening the door to what we have been calling higher-order strategies.

In the λ-calculus as well as the ρ-calculus the unrestricted treatment of variables in terms gives rise to the name capture problem. In the ρ-calculus the problem is addressed through higher-order substitution which performs α-conversion when necessary.

Binding is accomplished via matching modulo and equational theory. Two commonly used theories are AC and the trivial empty theory. Non-trivial theories such as AC generally will yield substitution sets as the result of matching. This gives rise to ρ-terms sets resulting from rule application. Such sets are different from the strategy sequences described in this paper in the following ways:

1. Sets are not sequences. In particular, they leave the notion of order undefined. Application order is essential when dealing with non-confluent systems.

2. While a set can be applied to a term, the semantics of set application is not compositional (i.e., the result is another set and not an individual term).

While it should be theoretically possible to simulate the constructs described in this paper within the ρ-calculus, the solution does not appear obvious.

8.3 ASF+SDF

In [4] a first-order system is built based on ASD+SDF where 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 [4] is to provide primitives so that the resulting traversal functions can be used in a type-safe manner.

In *Transformation Factories* [5], there are two kinds of traversal functions: transformers and analyzers. Analyzers may contain a combinator as a parameter and thus can be considered higher-order. However, it appears that the purpose of combinators in an analyzer is to provide a reusable way of manipulating data (e.g., the addition of two data values, etc.). As such, the nature of these higher-order combinators is significantly different from the dynamics presented in TL.

8.4 The S'_γ Calculus

The S'_γ calculus[22] is a strongly typed cousin of system S. It is a first-order system, supporting a variety of combinators for generic one-layer traversal which can be recursively

composed to produce typed generic traversals. Fundamental to the S'_γ calculus strategy extension combinator $\lhd$ which lifts a many-sorted strategy s to a generic type γ. 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 [22] a combinator $\bigcirc(\cdot)$ is introduced where $\cdot$ denotes the placeholders for arguments. A stratgey 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:

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

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 $fold$. An example is then given demonstrating why the fold combinator is more powerful than the $\bigcirc(\cdot)$ combinator. In spite of this, $fold$ is not added to the S'_γ calculus because without considerable care, such an addition would jeopardize its many-sorted type system. Therefore, the typing rules for $fold$ are not worked out. Nevertheless, the following useful type schemes are identified:

$TP \equiv$	$\forall\alpha.\alpha \to \alpha$	(Type preservation)
$TU(\tau) \equiv$	$\forall\alpha.\alpha \to \tau$	(Type unification)
$TA(\tau) \equiv$	$\forall\alpha.\langle\alpha, \tau\rangle \to \tau$	(Accumulation)
$TE(\tau) \equiv$	$\forall\alpha.\langle\alpha, \tau\rangle \to \alpha$	(TP with environment passing)
$TS(\tau) \equiv$	$\forall\alpha.\langle\alpha, \tau\rangle \to \langle\alpha, \tau\rangle$	(Tp with state passing)

The higher-order strategies of TL would most closely be characterized by the type scheme for $TS(\tau)$.

8.5 ELAN

ELAN [2] is a first-order rewrite system based on the ρ-calculus. Generic traversals are not supported and rule parameterization is the primary mechanism for transporting data from one term to another. Elan 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. In this framework, an evaluation mechanism (e.g., nondeterministic choice, backtracking, normalization, etc.) can be used to bring the parameterized data to appropriate (sub)terms.

8.6 Functional Strategies

Historically, the functional and strategic programming communities have had different research interests. Within the functional community, type systems[28][29], polytpyic programming[14],

morphisms[26], and monads[36][27] are being extensively investigated. In contrast, the strategic community has looked deeply into (1) term structure recognition[6][18], (2) controlling of term traversal[35], and (3) the development of a clean (i.e., generic) separation between generic control and rule definition[35][4]. 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[3] with the result being a *functional strategy.*

The notions of functional strategies originated from the observation that catamorphisms (i.e., what [26] refers to as "bananas") such as *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. Pursuing this idea further, fold algebras were developed in order to extend this notion of the fold morphism to systems of datatypes beyond the list datatype. While the idea of a fold algebra is appealing, it suffers from issues of scale due to the effort required to instantiate fold definitions with datatype systems arising from real-world programming languages (i.e., large bananas[20]). This scale issue was addressed by introducing updatable algebras[20]. In a framework supporting updateable algebras, one begins with a base algebra capturing generic behavior. For a given datatype system, two base algebras of particular interest are *idmap* – which leaves any term unchanged and *crush* – which collapses a term into an expression consisting of a collection of functions applied to base elements. Given an appropriate base algebra one can define a new algebra in terms of an update (or sequence of updates) to the base algebra. In this setting, updates are used to capture specific behavior. The resulting algebra can then be fed to an appropriate fold function.

Though updatable fold algebras can be used to capture generic and specific behaviors, the fold morphism will ultimately perform a complete traversal of the term to which it is applied. In a strategic framework, traversals can be controlled in a more refined fashion. In particular, a traversal need not recurse into the subterms of a term. In [19], algebra *generalization* is introduced as a mechanism for controlling traversal. A generalized algebra distinguishes the argument types from result types. Type-preserving and type-unifying strategies can both be expressed within the framework of a generalized algebra.

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. In [19], the core set of combinators for Stratego are implemented in a functional setting. Strafunski[21] 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 crush could be used to implement the TL dynamics described in this paper as well

[3]In contrast, the S'_γ calculus is an effort to bring strong typing into a strategic framework.

as transients. The basic idea would be to return an updated algebra as the result of a traversal rather than a function. The plumbing resulting from the application of transients could also be captured within a monad.

8.7 Maude

Maude[12] is an equational and rewriting system based on a refined form of algebraic specification built on top of a *membership equational logic*. The two fundamental constructs in Maude are the Σ-equation and the conditional Σ-equation. While conditional Σ-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 σ, the (irreducible) normal form of the left-hand side of the equation is equivalent to its right-hand side. That is, $\sigma(u) \downarrow_E \equiv \sigma(v) \downarrow_E$.

Within a specification, binary operations can be declared to satisfy various equational axioms such as associativity, commutativity, identity, and so forth. These properties are supported by a powerful matching algorithm which performs matching modulo equational theories.

While the notions of dynamics and transients are not primitive operations in Maude, its reflective capabilities [9][10][11] should easily support their implementation as well as the rest of the framework described in this paper.

9 Future Work

From the theoretical perspective we are working on developing combinators allowing a more dynamic interplay between the creation of strategy instances and their application. At present the creation and application phases are distinct. We believe that the expressive power of strategies could be significantly enhanced in a framework in which creation and application could be interleaved.

From the perspective of tool development, we are interested in extending HATS so that all the constructs of TL are supported. From the perspective of strategic programming, we are interested the effective use of higher-order strategies to solve real world applications in the area of program transformation. We are especially interested in the use of strategies beyond the second order for non-confluent systems involving user-defined binary combinators (e.g., $all\_thread\_right(s_0^n, \tau, \oplus)$ where $\oplus$ is user-defined).

10 Conclusion

The distributed data problem is characterized by the desire to be together semantically related terms from syntactically unrelated portions of a term. In a first-order strategic framework, accumulated data is frequently stored in a tuple or list and is then recursively distributed throughout a term structure by mechanisms such as parameterization. In this paper, higher-order strategies called *dynamics* are introduced as an alternate mechanism

for distributing data throughout term structures. Dynamic strategies have a behavior similar to the catamorphisms like the *fold* operator found in functional frameworks, though hylomorphisms perhaps most closely describe the behavior of dynamic strategies, especially when viewed from the perspective of non-list structures such as rose trees. A *transient* combinator is introduced that restricts the application of strategies to which it is applied. The interplay between transients and dynamics enables a variety of instances of the distributed data problem to be elegantly solved.

References

[1] 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.

[2] P. Borovansk, 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.

[3] 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.

[4] M. van den Brand, P. Klint, and J. Vinju. *Term rewriting with traversal functions.* Technical Report SEN-R0121, Centrum voor Wiskunde en Informatica, 2001.

[5] M. van den Brand, A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.* Science of Computer Programming 1997.

[6] M. van den Brand, A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.* IWPC 1998, June 24-26, Ischia, Italy.

[7] R. D. Cameron. *An abstract pretty printer.* IEEE Softw., 5(6):61–67, Nov. 1988.

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

[9] 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.

[10] M. Clavel. *Reflection in Rewriting Logic.* CSLI Publications, 2000.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer . *Metalevel Computation in Maude.* In 2^{nd} International Workshop on Rewriting Logic and its Applications (WRLA'98), Vol. 15, Electronic Notes in Theoretical Computer Science, Elsevier, 1998.

[12] 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.

[13] ELAN – User Manual.

[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] D. Kapur and P. Narendran. *Double-exponential Complexity of Computing a Complete Set of AC-Unifiers.* Logic in Computer Science (LICS), Santa Cruz, CA, June 1992.

[16] H. Kirchner and P. Moreau. *Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories.* Journal of Functional Programming, 11(2):207–251, 2001.

[17] P. Klint. *A meta-environment for generating programming environments.* ACM Transactions of Software Engineering and Methodology, 2(2):176–201, 1993.

[18] P. Klint and E. Visser. *Using filters for the disambiguation of context-free grammars.* In Proc. ASMICS Workshop on Parsing Theory, Milano, Italy, October 12–14, pp. 1–20, 1994. Technical Report 12694, Dipartimento di Scienze dell'Informazione, Universita degli Studi di Milano.

[19] R. Lämmel and J. Visser. *Type-safe Functional Strategies.* In Scottish Functional Programming Workshop, July 2000. 7.3

[20] 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.

[21] 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.

[22] R. Lämmel. *Typed Generic Traversal With Term Rewriting Strategies.* Journal of Logic and Algebraic Programming, Vol 54, pp 1–64, 2003.

[23] R. Lämmel, E. Visser, and J. Visser. *The Essence of Strategic Programming.* Draft.

[24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification 2^{nd} Edition.* Addison-Wesley, Reading, Massachusetts, 1999.

[25] J. A. McCoy. *An Embedded System For Safe, Secure And Reliable Execution Of High Consequence Software.* Proceedings of the 5^{th} IEEE International High-Assurance Systems Engineering Symposium, Nov. 2000.

[26] 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.

[27] 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.

[28] R. Milner. *A Theory of Type Polymorphism in Programming.* Journal of Computer and System Sciences, 17(3):348-375, December 1978.

[29] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[30] L. F. Rubin. *Syntax-directed pretty printing a first step towards a syntax-directed editor.* IEEE Trans. on Softw. Eng., SE-9(2):119–27, Mar. 1983.

[31] M. Tomita. *Efficient Parsing for Natural Languages – A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, 1986.

[32] B. Venners. *Inside the Java Virtual Machine.* McGraw-Hill, 1998.

[33] 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.

[34] E. Visser. *Language Independent Traversals for Program Transformation.* In Johan Jeuring, editor, Workshop on Generic Programming (WGP'00), Ponte de Lima, Portugal, July 2000.

[35] E. Visser. *Strategic Pattern Matching.* In: Rewriting Techniques and Applications (RTA '99), Trento, Lecture Notes in Computer Science (1999).

[36] P. Wadler. *Mondas for functional programming.* In J. Jeuring and E. Meijer, editors, Advanced Functional Programming, Springer Verlag, LNCS 925, 1995.

[37] V. L. Winter. *Program Transformation in HATS.* Proceedings of the Software Transformation Systems (STS) Workshop (part of ICSE '99), May 17 1999.

[38] V.L. Winter. *An Overview of HATS: A Lanaugage Independent High Assurance Transformation System.* Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET), March 24-27, 1999.

[39] V.L. Winter, S. Roach, G. Wickstrom. *Transformation-Oriented Programming: A Development Methodology for High Assurance Software.* Advances in Computers vol 58, to appear.

Distribution:

1	MS0510	Greg Wickstrom, 2116
1	MS0510	Anna Schauer, 2116
1		Victor Winter PKI 174 C 1110 South 67th Street Omaha, NE 68182
1		Mahadevan Subramaniam PKI 173 C 1110 South 67th Street Omaha, NE 68182 University of Nebraska at Omaha
1	MS9018	Central Technical Files, 8945-1
1	MS0899	Technical Library, 9616