

HIGHER-ORDER STRATEGIC PROGRAMMING: A ROAD TO SOFTWARE ASSURANCE

Victor L. Winter*

Department of Computer Science
University of Nebraska at Omaha
email: vwinter@mail.unomaha.edu

Steve Roach

Department of Computer Science
University of Texas at El Paso
email: sroach@cs.utep.edu

Fares Fraij

Department of Computer Science
University of Texas at El Paso
email: fzfraj@utep.edu

ABSTRACT

Program transformation through the repeated application of simple rewrite rules is conducive to formal verification. In practice, program transformation oftentimes requires data to be moved throughout the program structure. This article explores the use of higher-order rewrite rules as the mechanism for accomplishing such data movement. The effectiveness of higher-order rewrite rules is demonstrated by showing how they can be used to perform field distribution within a Java class loader. An approach to formal verification of a higher-order strategic implementation of a class loader is also briefly discussed.

KEY WORDS

program transformation, higher-order rules, distributed data problem, Java class loader, Sandia Secure Processor

1 Transformations

The transformation of program structures through rewriting is an active area of research [2][3][6][7]. A primary goal is to enable rewriting to be used as a formal and automatable mechanism for high-assurance software development. Driving this research is the idea that the repeated application of a properly constructed set of simple rewrite rules can effect a major change in the structure of a program. The implications of this span the entire software life cycle, ranging from the development of implementations from formal specifications all the way to software maintenance.

Equational reasoning lies at the heart of rewriting-based development and offers promise that the transformation of programs through rewriting can be scaled to real-world systems. A set of equations, called an *equational theory*, can be used to capture both domain as well as problem specific knowledge. The languages in which such substitutions are taking place must be referentially transparent with respect to the substitutions being performed. The question now remains as to how this knowledge is to be applied.

The fully general application of equational theories

*This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy. Victor Winter was also partially supported by NSF grant number CCR-0209187.

immediately leads to undecidability. However, rewriting provides restrictions that can result in a decidable theory. In rewriting, the symmetric equality relation $=$ is replaced by the anti-symmetric rewrite relation \rightarrow . For example, the equation $s = t$ states that the terms s and t may be freely substituted in place of one another. In contrast, the rewrite rule $s \rightarrow t$ states that s may be replaced by t , but does not specify that t may be replaced by s .

In a pure rewriting framework, the order in which rewrite rules are applied is left unspecified. For a confluent and terminating system of rewrite rules, any application order is guaranteed to produce the same *normal form*. Unfortunately, the manipulation of software generally involves sets of rewrite rules that are neither terminating nor confluent. In this setting, it is necessary to make explicit the way in which rewrite rules should be applied. One approach is to introduce additional function symbols to control the application of rewrite rules. This approach yields sets of rewrite rules that are not very reusable, and does not scale well to real-world problems [2].

2 Strategic Programming

In strategic programming [7], the control problem is solved by making explicit a number of combinators capable of specifying both which rewrite rules should be applied as well as where they should be applied. These combinators can be used together with rewrite rules to form expressions called *strategies*. In this context, rewrite rules themselves are referred to as strategies. Strategies that are fully parameterized on the rewrite rules to be applied are called *generic strategies*. Generic strategies can be reused by any problem domain provided the control they provide is suitable for the problem at hand. In some systems [2], the user is provided with a fixed set of generic strategies. In other systems [7][10][11], users may define their own generic strategies.

A common class of generic strategy is one that defines the traversal of a structure. Examples of these generic traversals include *TDL* which will traverse a structure in a top-down left-to-right fashion (i.e., outside-in), and *BUL* which will traverse a structure in a bottom-up left-to-right fashion (i.e., inside-out). Generic traversals such as *TDL* and *BUL* are parameterized on other strategies. For example, if s denotes a strategy (e.g., a rewrite rule) developed

for a particular problem domain, then the strategic expression $TDL(s)$ denotes a strategy that will traverse a structure from top to bottom and attempt to apply s at every point along the way.

Strategic systems typically will also provide several primitive combinators capable of controlling rule application. The two most common such combinators are those that enable the sequential and conditional composition of rewrite rules to be expressed. The semi-colon is typically used to denote the sequential composition combinator and the symbol $<+$ is typically used to denote the left-biased conditional combinator. Let s_1 and s_2 denote two strategies. The strategy $s_1 <+ s_2$ denotes the left-biased conditional composition of s_1 and s_2 . When applied to a term t , the strategy $s_1 <+ s_2$ will behave like s_1 if s_1 can be successfully applied to t , otherwise it will behave like s_2 .

3 Data Distribution

Strategic programming systems [2][6][7] have been successful in a number of software development areas and are being applied to increasingly complex activities. The *distributed data problem* (DDP) is common [12] and occurs when data found in one section of a program structure must be passed to another section. *Accumulation* is a standard approach taken in first-order systems to solve instances of the DDP. This approach involves the creation of auxiliary structures such as lists to hold data as well as accompanying lookup functions to extract data from such lists. Parameterization is then commonly used as the mechanism to transport these lists to the appropriate portions of a program.

A novel approach to the DDP is based on higher-order strategies [12][13]. The idea is to dynamically produce strategies that contain accumulated data rather than collect the data in auxiliary structures. The advantage of this approach is that data can be distributed via the application of strategies to terms (which is a primitive operation in a strategic framework). Thus, higher-order strategies provide an elegant approach for solving instances of the distributed data problem.

The strategic language TL [12][13] was developed to explore the implications of higher-order strategies. HATS is an IDE for strategic programming that implements a restricted dialect of TL. All examples presented here have been implemented in HATS. HATS runs on Windows NT/2000/XP and Unix-based platforms and is freely available [4].

The motivating application for this work is the *Sandia Secure Processor* (SSP), a subset of the Java Virtual Machine (JVM) developed at Sandia National Laboratories for use in high-consequence embedded applications. An application program for the SSP is called a *ROM image* and consists of a collection of class file-like structures stored on a read-only memory. In the SSP, all the structures used during execution must be present in the ROM image prior to execution. This enables class loading activities of

the SSP to be performed statically, prior to execution. Under these conditions, the functionality of the class loader is well-suited to a strategic implementation.

In the discussion that follows, we assume that an *application* consists of one or more Java *class files* having the structure defined in Lindholm and Yellin [8]. For the purposes of this discussion it is important to know that class files contain:

1. A *class* entry denoting the name of the class;
2. A *constant pool* whose entries contain a full description of the fields that are explicitly used within the class. This description forms a key that is unique within an application; and
3. A *fields section* containing all of the fields, both static and instance, declared within the class.

The remainder of the paper is as follows: Section 4 gives some basics of rewriting. In Section 5 the *table-entry distribution* problem is introduced as our running example. Section 6 picks up the table-entry distribution problem for the SSP. Section 7 touches on our research in verification. Section 8 concludes.

4 Rewriting

The examples presented in this paper are written in TL [12]. TL is a higher-order strategic programming language for transforming parse trees. Thus, in the context of TL, we use the words “structure”, “tree”, and “term” to denote a *parse tree*. For a given grammar, the notation $B[\alpha']$ denotes a *parse tree* corresponding to the derivation $B \xrightarrow{\pm} \alpha$ where the nonterminals in α have been subscripted yielding α' . Consider the BNF grammar production defining the structure of a simplified entry in a Java constant pool: $cp_entry ::= class\ field\ type$. A term corresponding to a constant pool entry having this form would be written as follows:

$$cp_entry \llbracket class_1\ field_1\ type_1 \rrbracket$$

Similarly, consider the BNF grammar production defining a greatly simplified Java class file structure consisting of a class name, a constant pool and a fields section, as with $classfile ::= \{class\ constant_pool\ field_section\}$. A term corresponding to such a class structure would be written as follows:

$$classfile \llbracket \{class_1\ constant_pool_1\ field_section_1\} \rrbracket$$

In TL, a first-order rewrite rule can have the form:

$$r : lhs \rightarrow rhs$$

where lhs and rhs denote terms and r is an optional label. The expression $r(t)$ 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 subscripted nonterminal symbols in leaf positions). When expressing strategy application, application is a left-associative implicit operator and parenthesis can be used to override precedence or to enhance readability. For example, $r t$ denotes the application of r to t and has the same meaning as $r(t)$.

A second-order rewrite rule has 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 .

Consider the abstract grammar shown in Table 1. Given this grammar, let us consider the second-order rewrite rule s^2 shown below:

$$s^2 : g[[i_1, data_1]] \rightarrow g[[i_2, i_1]] \rightarrow g[[i_2, data_1]]$$

Informally speaking, the rule s^2 can be seen as a template for relating information between the terms $data_1$ and i_1 in a specific context. The application $s^2(g[[1, b]])$ yields the first-order strategy $g[[i_2, 1]] \rightarrow g[[i_2, b]]$, and the application $(g[[i_2, 1]] \rightarrow g[[i_2, b]])(g[[2, 1]])$ yields $g[[2, b]]$. In this instance, s^2 provides a vehicle for transferring data from $g[[1, b]]$ to $g[[2, 1]]$.

g	::=	i data
data	::=	i char
i	::=	integer

Table 1. A small abstract grammar

5 Table-Entry Distribution

The *table-entry distribution* problem involves the distribution of table entry data between a set of tables. Though presented in an idealized form here, this problem is interesting because field distribution within the class loading phase of the JVM can be seen as an instance of table-entry distribution.

Let T denote a two-column table whose entries are of the form $(key, data)$ where $data$ is either a dash or a value of type integer. Given an entry (k, d) if d is of type integer, then we say that the entry (k, d) is *resolved*. Otherwise, d is a dash and the entry is *unresolved*.

Definition 1 A table T is well-formed and only if its key space is unique.

Definition 2 Given a set of well-formed tables $S_T = \{T_1, T_2, \dots, T_n\}$, a **distribution step** for S_T updates exactly

Table A		Table B		Table C	
key	data	key	data	key	data
x_A	1	x_B	10	x_C	100
y_A	2	y_B	20	y_C	200
z_A	3	z_B	30	z_C	300
x_B	—	x_A	—	y_A	—
y_C	—	z_C	—	z_B	—

Table 2. A well-formed table set

Table A		Table B		Table C	
key	data	key	data	key	data
x_A	1	x_B	10	x_C	100
y_A	2	y_B	20	y_C	200
z_A	3	z_B	30	z_C	300
x_B	10	x_A	1	y_A	2
y_C	200	z_C	300	z_B	30

Table 3. The normal form of a table set

one unresolved table entry in S_T . The update causes the table entry to become resolved.

Definition 3 A set of tables S_T is well-formed if and only if (1) every table in S_T is well-formed, and (2) every table entry in S_T can be resolved.

Given these definitions, the distribution step sequences are confluent and terminating for well-formed table sets. The term *distribution-set* refers to a set of ground rewrite rules capable of normalizing a given table set. Table 2 shows a well-formed table set S_T consisting of the three tables. The normal form of S_T is shown in Table 3. Table 4 shows a distribution-set for Table 2. The fixed point application of the distribution-set shown in Table 4 to the table set in Table 2 yields the table in Table 3.

From the perspective of correctness, we claim that distribution-sets, as shown in Figure 4, provide an attractive strategic solution to the table-entry distribution problem. This is primarily due to the direct relationship between the rules in the distribution-set and the definition of what it means to perform a distribution step. However, a drawback of the approach described is that it is highly

r_1	:	$(x_A, -) \rightarrow (x_A, 1)$
r_2	:	$(y_A, -) \rightarrow (y_A, 2)$
r_3	:	$(z_A, -) \rightarrow (z_A, 3)$
r_4	:	$(x_B, -) \rightarrow (x_B, 10)$
r_5	:	$(y_B, -) \rightarrow (y_B, 20)$
r_6	:	$(z_B, -) \rightarrow (z_B, 30)$
r_7	:	$(x_C, -) \rightarrow (x_C, 100)$
r_8	:	$(y_C, -) \rightarrow (y_C, 200)$
r_9	:	$(z_C, -) \rightarrow (z_C, 300)$

Table 4. A distribution-set

app	::=	cl_file cl_file app
cl_file	::=	{ class const_pool field_section }
const_pool	::=	{ cp_list }
cp_list	::=	cp_entry cp_list cp_entry
cp_entry	::=	class field type offset type
field_section	::=	{ field_list }
field_list	::=	f_entry field_list f_entry
f_entry	::=	field offset
class	::=	id
field	::=	id
offset	::=	HEX
id	::=	ident
type	::=	int long byte

Table 5. A simplified grammar for Java Classfiles

problem-specific. Furthermore, the manual construction of distribution-sets can be cumbersome and error prone. The advantage offered by a higher-order strategies is that they can be used to construct distribution-sets automatically.

6 Field Distribution for Java

The *field distribution problem*, i.e., distributing field offset information among the constant pool entries in a Java application, is an instance of the table-entry distribution problem. A simplified version of the problem is presented here. We assume that the structure of Java class files has been simplified as defined by the grammar in Table 5; and all fields are associated strictly with offset addresses. We further assume that field entries in the *constant pool* table have been resolved (by an earlier transformation) into *keys* and that field entries in the *field section* table have been resolved into *data*. Field distribution can be realized by the higher-order strategies shown in Table 9.

Table 6 shows an application consisting of three simplified class files for which the constant pool entries are in resolved form and offsets have been computed for field entries in every field section table. The classes are named *A*, *B*, and *C*. The constant pool of *A* contains three field keys *C*, *B*, and *A*. The class *A* locally declares the fields *x*, *y*, and *z*. The computed hexadecimal offsets for these fields are :0004, :000C, and :0014 respectively.

Consider the field entry *x* : 0004 belonging to the field section of *A* and the key *A x long* belonging to constant pool of *C*. In this context, a distribution step is defined as *A x long* \rightarrow :0004 *long*.

The higher-order solution to field distribution is realized using a few rewrite rules and two generic strategies: *BUL* and *lcond_bul*. Recall that the strategy *BUL* performs a bottom-up left-to-right traversal of a term structure. The expression *BUL*(*s*)*t* will result in the first-order rewrite *s* being applied to every sub-term encountered during the traversal of term *t*.

The strategy *lcond_bul* is similar to *BUL* except that

lcond_bul will will apply a higher-order strategy to the terms it encounters. The successful application of a higher-order strategy will not yield a term, but rather will produce another strategy whose order is one less than the strategy being applied. For example, let *s*² denote a second-order rewrite rule and let *t* denote a term. The successful application of *s*² resulting from the evaluation of *lcond_bul*(*s*²)*t* produces a set of first-order strategies {*s*₁, *s*₂, ..., *s*_{*n*}}. However, the higher-order strategic framework of TL goes beyond this and enables the resulting (first-order) strategies to be composed using a binary strategic combinator. The generic strategy *lcond_bul* has been defined in such a way that it will use the conditional combinator, denoted by the symbol <+ to compose the strategies produced. Thus, the resulting strategic expression is *s*₁ <+ *s*₂ <+ ... <+ *s*_{*n*}.

Let us now consider the evaluation of the strategic expression: *lcond_bul* FIELD *field_section*₁ where

$$\begin{aligned} \text{FIELD} &= f_entry[[field_1\ offset_1]] \\ &\rightarrow \\ &cp_entry[[A\ field_1\ type_1]] \\ &\rightarrow \\ &cp_entry[[offset_1\ type_1]] \end{aligned}$$

and *field_section*₁ is the field section table belonging to the class file *A*. This strategic expression is taken from the Class and Field strategies defined in Table 9. Under the given conditions, the result of evaluation will yield the first-order strategy shown in Table 8.

<i>cp_entry</i> [[<i>A x long</i>]] \rightarrow <i>cp_entry</i> [: 0004 <i>long</i>]]
<+
<i>cp_entry</i> [[<i>A y long</i>]] \rightarrow <i>cp_entry</i> [: 000C <i>long</i>]]
<+
<i>cp_entry</i> [[<i>A z long</i>]] \rightarrow <i>cp_entry</i> [: 0014 <i>long</i>]]

Table 8. A field distribution-set strategy for the class file *A*

Given the appropriate instantiations of FIELD and *field_section*₁, the strategic expression *lcond_bul* FIELD *field_section*₁ can be used to generate field distribution-set strategies for the class files *B* and *C*. When taken collectively, these strategies form a distribution-set strategy for the Java application under consideration.

7 Assurance

Sandia requires that the development of the SSP produces strong evidence of the correctness of the system. The class loader is a weak link in the assurance chain of the SSP. Considerable effort is being devoted towards providing assurance in the translation performed by the SSP class loader.

Class Name	Constant Pool			Field Section				
{ 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	} }

Table 6. A simplified application consisting of the class files A, B, and C

Class Name	Constant Pool			Field Section				
{ 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	} }

Table 7. Application after field distribution

Formal reasoning techniques are being used to verify general properties of the class loader. The assurance provided by formal reasoning comes in the form of a mathematical proof and typically involves a model of the system under analysis rather than the actual system itself. General properties are stated in terms of theorems involving the model. The proof of theorems provides strong assurance that the model behaves as required. Assurance of the correctness of the system under analysis relies on (1) confidence that the proofs themselves are sound, and (2) confidence that the model faithfully describes the system (e.g., theorems that hold for the model hold for the system).

The modeling and verification framework provided by ACL2 [5] is used to prove theorems about the class loader. ACL2 is a programming language based on the applicative subset of Common Lisp. In this language, users can build executable models of software systems. ACL2 is also a tool that assists users in proving theorems about their ACL2 programs. It has been used to prove the correctness of hardware implementations of microprocessors and floating point algorithms [1] and more recently to verify parts of implementations of the JVM [1][9].

The class loader is modelled as a system described in terms of states and state transitions. In the context of the class loader, we model the class files of the Java application as our state and the transformations on this application as our state transitions. The JVM and the SSP provide the basis for formally understanding equivalence between states.

A model of TL is constructed by defining an abstract machine that controls the application of transformation rules. Each transformation rule modifies the state of the system. The abstract machine operates according to the following sequence: fetch the next transformation rule and node from the current state, apply the transformation, and return a new machine state.

Though we are exploring the verification of a number of properties of the class loader, our ultimate goal is to verify that the transformation rules preserve the meaning of the term to which they are applied (i.e., the class loader

is correct). In the context of the SSP, the initial term is a set of class files, C_0 , generated by a Java compiler. The semantics of this term is defined by the JVM specification [8]. Let $Eval_{JVM} : classfiles \times inputs \rightarrow outputs$ denote the mapping from classfiles and inputs to outputs. $Eval_{JVM}$ defines the behavior of the program encoded in the classfiles. The final term is a ROM image, which we denote C_{ROM} . The semantics of this term is defined by the SSP hardware, $Eval_{SSP}$. HATS accomplishes the conversion of C_0 to C_{ROM} , as indicated by the notation $C_{ROM} = T^*(C_0)$, where T^* denotes a sequence of transformations. In this notational framework, what must be shown for inputs I is:

$$\forall(C_0, I) Eval_{JVM}(C_0, I) = Eval_{SSP}(T^*(C_0), I)$$

The problem above can be decomposed by defining a sequence of normal forms, C_0, C_1, C_2, \dots in the transformation of C_0 to C_{ROM} . These normal forms are formally specified and are theorems within our verification framework. Constant pool normalization and field distribution are two examples of normal forms. Informally stated, in constant pool normalization all indirection is removed from the constant pool entries of the classfiles in C_0 . Let T^1 denote the transformation that accomplishes this task. Similarly, let T^2 denote the normal form resulting from field distribution. At present, a sequence of five intermediate normal forms have been defined. For each normal form, there is an evaluation function, $Eval_n$. Thus, the original correctness conjecture can be restated as a sequence of conjectures:

$$\begin{aligned} \forall(C_0, I) Eval_{JVM}(C_0, I) &= Eval_1(T^1(C_0), I) \\ &= Eval_2(T^2(T^1(C_0)), I) \\ &\dots \\ &= Eval_{SSP}(T^*(C_0), I) \end{aligned}$$

where T^* is the composition of the individual transformations. This allows the proof to be constructed incrementally, and therefore, reduces the complexity of the proof.

To date, a class loader for the SSP has been completely implemented using HATS. Because Sandia Na-

$Field(class_1)$: $f_entry[[field_1\ offset_1]]$ $\rightarrow cp_entry[[class_1\ field_1\ type_1]] \rightarrow cp_entry[[offset_1\ type_1]]$
$Class$: $classfile[[\{class_1\ constant_pool_1\ field_section_1\}]]$ $\rightarrow lcond_bul (Field\ class_1)\ field_section_1$
$Field_Distribution$: $application_1 \rightarrow BUL(lcond_bul\ Class\ application_1)\ application_1$

Table 9. A higher-order strategic solution to the field distribution problem

tional Laboratories places a high priority on the provision of strong evidence that the system behaves correctly, attention is being turned to the verification of the transformations used. ACL2 proofs of a simplified version of HATS have been completed for the first two canonical forms. Using this as a basis, a complete model of HATS will be implemented and the remaining canonical forms verified.

8 Conclusion

In practice, program transformation oftentimes requires data to be distributed throughout a program structure. TL is a strategic programming language in which higher-order rules and strategies provide a mechanism for distributing data. Conceptually, the functionality of a higher-order strategy can be understood in terms of the first-order rules they ultimately produce. In the case of the table-entry distribution problem, a higher-order strategy can be used to construct *distribution-sets* which, in turn, can be used to resolve table entries. A similar approach can be used to solve the field distribution problem within a Java class loader.

The ability of higher-order strategies to automatically generate first-order strategies leads to elegant solutions to instances of the distributed data problem. In turn, this elegance can benefit efforts to formally verify the correctness of the strategies. Formal verification is a technique that can be used to provide strong evidence of the correctness of a software implementation such as a class loader. Such a level of assurance is important when considering high-consequence systems such as the SSP.

References

- [1] Robert S. Boyer and Yuan Yu. *Automated proofs of object code for a widely used microprocessor*. Journal of the ACM, 43(1):166-192, January 1996.
- [2] M. van den Brand, P. Klint, and J. Vinju. *Term rewriting with traversal functions*. Technical Report SEN-R0121, Centrum voor Wiskunde en Informatica, 2001.
- [3] H. Cirstea and C. Kirchner. *Intoduction to the rewriting calculus*. INRIA Research Report RR-3818, December 1999.
- [4] HATS, <http://faculty.ist.unomaha.edu/winter/hats-uno/HATSWEB/index.html>
- [5] M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.
- [6] R. Lämmel. *Typed Generic Traversal With Term Rewriting Strategies*. Journal of Logic and Algebraic Programming, Vol 54, pp 1–64, 2003.
- [7] R. Lämmel, E. Visser, and J. Visser. *The Essence of Strategic Programming*. Draft.
- [8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification 2nd Edition*. Addison-Wesley, Reading, Massachusetts, 1999.
- [9] J.S. Moore. *Proving Theorems about Java and the JVM with ACL2*. Models, Algebras and Logic of Engineering Software, M. Broy and M. Pizka (eds), IOS Press, Amsterdam, pp 227-290, 2003.
- [10] 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.
- [11] E. Visser. *Strategic Pattern Matching*. In: Rewriting Techniques and Applications (RTA '99), Trento, Lecture Notes in Computer Science (1999).
- [12] V.L. Winter and M. Subramaniam. *The Transient Combinator, Higher-Order Strategies, and the Distributed Data Problem*. Science of Computer Programming Special Issue on Program Transformation (to appear).
- [13] V.L. Winter. *Strategy Construction in the Higher-Order Framework of TL*. The 5th International Workshop on Rule-Based Programming (RULE 2004) (to appear).