

SAND2004-0871
Unlimited Release
Printed March 2004

Strategy Application, Observability, and the Choice Combinator

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

Abstract

In many strategic systems, the choice combinator provides a powerful mechanism for controlling the application of rules and strategies to terms. The ability of the choice combinator to exercise control over rewriting is based on the premise that the success and failure of strategy application can be observed.

In this paper we present a higher-order strategic framework with the ability to dynamically construct strategies containing the choice combinator. To this framework, a combinator called *hide* is introduced that prevents the successful application of a strategy from being observed by the choice combinator. We then explore the impact of this new combinator on a real-world problem involving a restricted implementation of the Java Virtual Machine.

Contents

1	Introduction	7
2	An Overview of TL	8
2.1	Term Notation	8
2.2	Some First-Order Traversals from the TL Library	9
2.3	Higher-Order Strategies	10
2.3.1	Some Higher-Order Traversals from the TL Library	10
2.4	The <i>transient</i> Combinator	11
2.4.1	Example	12
2.5	The <i>hide</i> Combinator	12
2.5.1	Example	13
3	Absolute Address Calculation for Static Fields in Java Class Files	14
3.1	Static Field Address Calculation	16
4	HATS: A Restricted Implementation of <i>TL</i>	20
5	Conclusion	20

List of Figures

1	The basic constructs of TL	8
2	General first-order traversals	9
3	General higher-order traversals	11
4	A simple grammar involving sums	12
5	A simple grammar involving integer lists	13
6	A simplified grammar for Java class files	15
7	A list of Java types	15
8	Three abstract class files	16
9	A strategic program for incrementing static fields	17
10	Initial configuration of strategy and app_0	18
11	Configuration of strategy and app_0 after application of the first transient to the first sfield	18
12	Configuration of strategy and app_0 after application of the first hide strategy to the second sfield	18
13	Configuration of strategy and app_0 after application of the second transient to the second sfield	19
14	Configuration of strategy and app_0 after application of the first hide strategy to the third sfield	19
15	Configuration of strategy and app_0 after application of the second hide strategy to the third sfield	19

16	Configuration of strategy and <i>app₀</i> after application of the third transient strategy to the third sfield	20
17	Address assignments for static fields	20

Nomenclature

BNF	Backus-Naur Form
HATS	High Assurance Transformation System
IDE	Integrated Development Environment
JVM	Java Virtual Machine
ROM	Read Only Memory
SSP	Sandia Secure Processor
TL	Transformation Language – a higher-order strategic programming language

1 Introduction

The notion of choosing the application of one rule over another is central to many strategic programming systems. 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. A biased choice combinator is also common in the literature. Stratego and the S'_γ calculus, 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 if and only if s_1 fails. Given these combinators, *left-biased* choice (first try s_1 and if that fails try s_2) and *right-biased* choice (first try s_2 and if that fails try s_1) can be defined as follows:

$$\begin{aligned} s_1 <+ s_2 &\stackrel{def}{=} s_1 + (\neg s_1; s_2) \\ s_1 >+ s_2 &\stackrel{def}{=} (\neg s_2; s_1) + s_2 \end{aligned}$$

In this paper, we restrict our attention to the *left-biased* choice and *right-biased* choice combinators. An essential component of both of these combinators is the ability to observe the behaviour of *strategy application* (i.e., whether the application of a strategy to a term has succeeded or failed). We use the term *failure-based* to denote a semantic framework where a special value *fail* is returned when a strategy or rule fails to apply to a term. Conversely, we use the term *identity-based* to denote a semantic framework where a term is left unchanged when the application of a strategy or rule to a term fails. Systems like Stratego, ELAN, and the S-calculus have semantic frameworks that are failure-based. In contrast, ASF+SDF as well as most classical rewriting systems have semantic frameworks that are identity-based. In this paper we consider the higher-order strategic system TL whose semantic framework is identity-based.

In a failure-based framework, the observation of strategy application is straightforward since the value *fail* explicitly indicates when a rule application has failed. However, in an identity-based framework such as TL, the implementation of observation becomes a bit more involved. One way to solve the problem is to implement an observer predicate $observe(s, t)$ that evaluates to *true* if and only if the strategy s applies to the term t . Note that in addition to being computationally expensive, simply performing an equality comparison on the terms t and $s(t)$ is not correct (e.g., if $t \neq s(t)$ then $observe(s, t)$ is true otherwise it is false). In particular, such a test would not be able to distinguish between the failure or success of applications involving identity-like rules (e.g., the application of the rule $b \rightarrow b$ to the term b). The proper semantics for the *observe* predicate is that it must actually track when the right-hand side of a rule is substituted for the term to which the rule has been applied. Conceptually speaking, in an identity-based framework one must be able to observe when a computation traverses the “arrow” separating the left and right-hand sides of a rewrite rule. From this foundation, the definition of the observe predicate can be extended to include strategies.

Let us consider the introduction of a combinator called *hide* into an identity-based strategic framework. The purpose of the *hide* combinator is to prevent the application of a strategy from being observed. For example, $observe(hide(s), t)$ will always evaluate to false, and a strategy of the form

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

Figure 1: The basic constructs of TL

$hide(s_1) \dagger s_2$ will always attempt to apply s_1 followed by s_2 . In this paper, we explore the consequences of extending the system TL with a *hide* combinator having the semantics just described.

The remainder of the paper is organized as follows: Section 2 gives an overview of the higher-order strategic language TL. Section 3 describes static field address calculation for the Sandia Secure Processor (SSP), a hardware implementation of a restricted subset of the Java Virtual Machine for use in high-consequence safety-critical applications. In this section, a strategic program written in TL for calculating static fields is analyzed. Section 4 gives a brief overview of a system call HATS which implements a restricted dialect of TL. All examples mentioned and discussed in this paper have been implemented in HATS. Section 5 concludes.

2 An Overview of TL

TL is an indentity-based higher-order strategic system for rewriting parse trees. In TL, a domain (i.e., a term language) is defined using an Extended-BNF notation and terms also called *parse expressions* are described using a special notation (see Section 2.1). TL supports the combinators and strategic constants shown in Figure 1.

In addition to the constructs shown above, TL also provides a number of one-layer generic traversals providing the ability to define special purpose traversals. These constructs are not central to the topic of this paper and are therefore omitted. Instead we present a number of generic traversals that form part of the TL traversal library.

2.1 Term Notation

Let $G = (N, T, P, S)$ denote a context-free grammar where N is the set of nonterminals, T is the set of terminals, P is the set of productions, and S is the start symbol. Given an arbitrary symbol $B \in N$ and a string of symbols $\alpha = X_1 X_2 \dots X_m$ 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 $B \xRightarrow{*} \alpha$ is used to denote that B can derive α in zero or more expansion steps. Similarly, one can write $B \xRightarrow{+} \alpha$ to denote

a derivation consisting of one or more expansion steps.

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

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

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

2.2 Some First-Order Traversals from the TL Library

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

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

Traversal	bottom-up	top-down	left-to-right	right-to-left
TDL		■	■	
TDR		■		■
BUL	■		■	
BUR	■			■
FIX_TDL		■	■	
FIX_TDR		■		■

Figure 2: General first-order traversals

2.3 Higher-Order Strategies

In TL a second-order strategy s^2 can be applied to a term t yielding a first-order strategy s^1 , and more generally, the application of a strategy of order n to a term t will result in a strategy of order $n - 1$. From a conceptual standpoint, the purpose of a second-order strategy is to create a first-order strategy that is *specific* to a particular term. Typically this will mean that one or more schema variables have been bound to specific terms. For example, suppose that in the context of identifier renaming one wants to rename the identifier x to a newly generated identifier y . In this case, it would be convenient if one could dynamically generate a rule of the form $ident[[x]] \rightarrow ident[[y]]$ and apply this rule to the appropriate term. The attractiveness of such a capability has been recognized in Stratego which provides a mechanism for dynamically creating rules and controlling their scope of application. TL lifts and extends this idea to a higher-order framework. In particular, higher-order traversals can be employed to dynamically construct *strategies* as opposed to adding rules to rule bases.

From a conceptual standpoint, a higher-order traversal traverses a term and applies a higher-order strategy s^n to every term encountered. Because the strategy being applied is of order n , the result of an application will be a strategy of order $n - 1$. If a traversal visits m terms, then m strategies of order $n - 1$ will be produced. Let $s_1^{n-1}, s_2^{n-1}, \dots, s_m^{n-1}$ denote the strategies resulting from such a traversal. Let \oplus denote a binary combinator such as sequential composition, left-biased choice, or right-biased choice. In TL, binary strategic combinators can be used to combine strategic results into a single strategy. That is, higher-order traversals will combine a sequence of resultant strategies $s_1^{n-1}, s_2^{n-1}, \dots, s_m^{n-1}$ into a strategy of the form:

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

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

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

In practice, the unary combinators that are most useful are: *transient*, *hide*, and *I*. The *transient* and *hide* combinators are described in Sections 2.4 and 2.5 respectively.

2.3.1 Some Higher-Order Traversals from the TL Library

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

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

Travaersal	bottom-up	top-down	left-to-right	right-to-left	\oplus	τ
<i>rcond_tdl</i>		■	■		\rightarrow	<i>I</i>
<i>rcond_tdr</i>		■		■	\rightarrow	<i>I</i>
<i>lcond_tdl</i>		■	■		\leftarrow	<i>I</i>
<i>lcond_tdr</i>		■		■	\leftarrow	<i>I</i>
<i>rcond_bul</i>	■		■		\rightarrow	<i>I</i>
<i>rcond_bur</i>	■			■	\rightarrow	<i>I</i>
<i>lcond_bul</i>	■		■		\leftarrow	<i>I</i>
<i>lcond_bur</i>	■			■	\leftarrow	<i>I</i>
<i>seq_tdl</i>		■	■		;	<i>I</i>
<i>seq_tdr</i>		■		■	;	<i>I</i>
<i>seq_bul</i>	■		■		;	<i>I</i>
<i>seq_bur</i>	■			■	;	<i>I</i>

Figure 3: General higher-order traversals

2.4 The *transient* Combinator

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

$$\begin{aligned} \textit{skip} \leftarrow s &\equiv s \\ \textit{skip} \rightarrow s &\equiv s \end{aligned}$$

Operationally, we define a strategy of the form *transient*(*s*) as a strategy that *reduces* to the strategy *skip* if the application of the strategy *s* has been observed. Thus, transients open the door to *self-modifying* strategies. When using a traversal to apply a self-modifying strategy to a term, a different strategy may be applied to every term encountered during a traversal. For example, let $\textit{int}_1 \rightarrow \textit{int}[[2]]$ denote a rule that rewrites an integer to the value 2. If such a rule is applied to a term in, say, a top-down fashion all of the integers in the term will be rewritten to 2. Now consider the following self-modifying strategy:

$$\textit{transient}(\textit{int}_1 \rightarrow \textit{int}[[1]]) \leftarrow \textit{transient}(\textit{int}_1 \rightarrow \textit{int}[[2]]) \leftarrow \textit{transient}(\textit{int}_1 \rightarrow \textit{int}[[3]])$$

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

2.4.1 Example

The transient combinator can be used in a higher-order setting with interesting results. Consider the language defined by the BNF grammar shown in Figure 4.

term	::=	int id “add” “(” term “,” term “)”
int	::=	integer
const	::=	id

Figure 4: A simple grammar involving sums

This language defines terms consisting of sums involving integers and symbolic constants. Suppose that one wants to construct a strategy capable of reversing the first three integers in a term without otherwise altering the term structure. For example, $add(add(1,b),add(add(2,3),4))$ should be rewritten to $add(add(3,b),add(add(2,1),4))$. In TL, such a reversal could be accomplished by the following strategic program.

Implementation in TL

```

replace : int1 → transient(int2 → int1)
load3  : transient(replace) +> transient(replace)+> transient(replace)
reverse3 : t → TDL (rcond_tdl load3 t) t

```

Here, the strategy *replace* is a labeled second-order strategy that when applied to an integer i will return the strategy $transient(int_2 \rightarrow i)$. This strategy is capable of rewriting a single (arbitrary) integer to i . The strategy *load3* will enable the strategy *replace* to be applied at most three times during a traversal. Assuming the semantics of *rcond_tdl* given in Section 2.3.1 the evaluation of the strategic expression

$$rcond_tdl \text{ load3 } add(add(1,b),add(add(2,3),4))$$

will yield the strategy

$$transient(int_2 \rightarrow 1)+> transient(int_2 \rightarrow 2)+> transient(int_2 \rightarrow 3)$$

This first-order strategy, when applied by the traversal *TDL* to the term $add(add(1,b),add(add(2,3),4))$ will yield $add(add(3,b),add(add(2,1),4))$. Thus, the strategy *reverse3* will correctly reverse the first three integers of the term t .

2.5 The *hide* Combinator

The strategic combinator *hide* provides an interesting extension to the framework of TL. This unary combinator restricts the observability of strategy application from the perspective of the choice combinators. In particular, the *hide* combinator satisfies the following properties:

$$\begin{aligned} \text{hide}(s_1) <+ s_2 &\equiv s_1; s_2 \\ s_1 +> \text{hide}(s_2) &\equiv s_2; s_1 \end{aligned}$$

2.5.1 Example

When combined within a strategy, the *hide* and *transient* combinators can interact with each other in interesting ways. Consider the following grammar:

$\begin{aligned} \text{int_list} &::= \text{int int_list} \mid \text{int} \\ \text{int} &::= \text{integer} \end{aligned}$
--

Figure 5: A simple grammar involving integer lists

Given this grammar we are interested in developing a strategy that will transform the list of zeros into a list of integers denoting the position of the element in the list. That is, we want a strategy that would transform a term $\text{int_list}[[0\ 0\ 0]]$ into the term $\text{int_list}[[1\ 2\ 3]]$. Granted there is more than one way this can be accomplished, but we will see in Section 3.1 that the approach taken in the strategy shown below has some desirable properties when considering more complex term structures.

Pseudo-TL

$\text{increment} : \text{int}_2 \rightarrow (\text{transient}(\text{int}_1 \rightarrow \text{int}_1 + 1) <+ \text{hide}(\text{int}_1 \rightarrow \text{int}_1 + 1))$

$\text{position} : \text{int_list}_1 \rightarrow \text{TDL} (\text{lcond_tdl increment int_list}_1) \text{int_list}_1$

The strategy $\text{int}_1 \rightarrow \text{int}_1 + 1$ takes an integer value int_1 and increments it by 1. Technically speaking, the syntax given for the addition would not be allowed in TL because int_1 is a parse tree while $+$ is an operation defined on integers. The actual syntax is only slightly more involved but not particularly interesting in the context of this example and therefore abstracted away. When applied to an integer int_2 , the second-order strategy *increment* will produce a first-order strategy of the form:

$$\text{transient}(\text{int}_1 \rightarrow \text{int}_1 + 1) <+ \text{hide}(\text{int}_1 \rightarrow \text{int}_1 + 1)$$

The strategic expression $(\text{lcond_tdl increment int_list}_1)$ will traverse int_list_1 (e.g., $\text{int_list}[[0\ 0\ 0]]$) in a top-down left-to-right fashion applying the *increment* strategy to every term (e.g., integer term) encountered. The first-order results will then be composed using the left-biased combinator. The resulting strategy will then be applied to int_list_1 by the traversal *tdl*.

Let us trace the application of *position* to the term $\text{int_list}[[0\ 0\ 0]]$. First, the strategic expression $(\text{lcond_tdl increment int_list}_1)$ will be evaluated. This will yield the following first-order strategy:

$$\begin{aligned} \text{transient}(\text{int}_1 \rightarrow \text{int}_1 + 1) &<+ \text{hide}(\text{int}_1 \rightarrow \text{int}_1 + 1) <+ \\ \text{transient}(\text{int}_1 \rightarrow \text{int}_1 + 1) &<+ \text{hide}(\text{int}_1 \rightarrow \text{int}_1 + 1) <+ \\ \text{transient}(\text{int}_1 \rightarrow \text{int}_1 + 1) &<+ \text{hide}(\text{int}_1 \rightarrow \text{int}_1 + 1) \end{aligned}$$

Now the traversal *TDL* will traverse the term *int_list*[[0 0 0]] applying the above strategy to every (integer) term encountered. The first *transient* in the strategy will apply to the first integer 0 encountered thus incrementing its value to 1. This application can be observed by the left-biased choice combinator, so no further applications are attempted, and the traversal moves on to the next term with the altered strategy:

$$\begin{array}{llll} \textit{skip} & <+ & \textit{hide}(\textit{int}_1 \rightarrow \textit{int}_1 + 1) & <+ \\ \textit{transient}(\textit{int}_1 \rightarrow \textit{int}_1 + 1) & <+ & \textit{hide}(\textit{int}_1 \rightarrow \textit{int}_1 + 1) & <+ \\ \textit{transient}(\textit{int}_1 \rightarrow \textit{int}_1 + 1) & <+ & \textit{hide}(\textit{int}_1 \rightarrow \textit{int}_1 + 1) & <+ \end{array}$$

When the second 0 is encountered, the *hide* strategy is first to be applied and increments 0 to produce 1. Since the application of a *hide* strategy cannot be observed, the application of the following *transient* is attempted. This application increments 1, the current value of the term, to produce 2. As in the previous case, the application of the *transient* can be observed by the left-biased choice combinator and so the application of the strategy to the second term stops. The successful application of the transient causes it to be removed from the strategy and we are left with:

$$\begin{array}{llll} \textit{skip} & <+ & \textit{hide}(\textit{int}_1 \rightarrow \textit{int}_1 + 1) & <+ \\ \textit{skip} & <+ & \textit{hide}(\textit{int}_1 \rightarrow \textit{int}_1 + 1) & <+ \\ \textit{transient}(\textit{int}_1 \rightarrow \textit{int}_1 + 1) & <+ & \textit{hide}(\textit{int}_1 \rightarrow \textit{int}_1 + 1) & <+ \end{array}$$

And finally, the last 0 is encountered. The first two *hide* strategies apply incrementing the value of the third 0 to 2. Now a *transient* is encountered which increments the value of the third term to 3. Again, the observation of this *transient* causes the strategy application to stop. Thus the resulting term is: *int_list*[[1 2 3]].

3 Absolute Address Calculation for Static Fields in Java Class Files

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) [11]. An application program for the SSP is called a *ROM image* and consists of a collection of structures similar to class files that have been stored on a read-only memory. The SSP is a *closed system* in the sense that the execution of an application program may only access the structures in the ROM (e.g., no dynamic loading of class files across a network). The closed nature of the SSP's execution environment enables the class loading activities of the JVM [10] to be performed statically. Under these conditions, the functionality of the class loader is well-suited to a strategic implementation.

In the discussion that follows, we assume that an *application* consists of one or more Java *class files* and that Java class files have the greatly simplified structure defined by the grammar shown in Figure 6. However, we have hopefully left enough structural detail so that the reader gets some sense of the complexity of the term structures that one must deal with when rewriting Java applications.

app	::=	[app] class
class	::=	“{” class_id super_id cp “[” fields “]” “[” methods “]” “}”
class_id	::=	id
super_id	::=	id
cp	::=	[cp] c_entry
c_entry	::=	key
fields	::=	field [fields]
field	::=	sfield ifield
sfield	::=	key “:” addr
ifield	::=	key “:” addr
methods	::=	method_list
method_list	::=	m_entry [method_list]
m_entry	::=	key “(” “)”
key	::=	id “.” id “.” desc
desc	::=	id id “(” [id] “)”
index	::=	integer
addr	::=	integer
id	::=	ident

Figure 6: A simplified grammar for Java class files

Given this structure, we are interested in assigning a unique absolute address to every static field occurring within an application. In this example we will assume that memory is byte addressable and that the size of a static field in memory is dependent upon its type. In the example given, we restrict ourselves to the types shown in Figure 7.

Typically, additional constraints are imposed on memory mappings (e.g., an integer value should not span a 32-bit (i.e., word) boundary). This constraint impacts the definition of the “addition” function but does not otherwise impact the strategic approach, and can be therefore omitted from the example without significant loss of generality. Other things to know about static fields include:

Field Descriptor	Memory Size	Comment
B	1 byte	byte
C	2 bytes	character
I	4 bytes	integer
J	8 bytes	long integer
S	2 bytes	short integer
Z	1 byte	boolean

Figure 7: A list of Java types

Class	Super	Constant Pool	Fields	Methods	
{ C	A	C C.x1.I x1 ...	[C.x1.I::- C.c1.J:- C.x2.J::- C.x3.S::- C.c2.B:-]	[C.bar.I(J) C.f.I(J) ...]	}
{ A	obj	A A.x1.I x1 A.x2.J x2 ...	[A.x1.I::- A.a1.I:- A.x2.J::- A.a2.I:- A.x3.C::-]	[A.foo.I(I) A.bar.I(J) ...]	}
{ B	A	x1 B.x1.B B ...	[B.x1.B::- B.b1.S:- B.x2.Z::- B.x3.I::- B.b2.J:-]	[B.foo.I(I) B.f.I(J) ...]	}

Figure 8: Three abstract class files

1. The definition of static fields and instance fields may be interleaved within the fields section of a class file, and
2. a class may declare zero or more static fields.

In Figure 8 we see three class files presented in no particular order. The class files have already been partially resolved so that all constant pool indexes have been replaced by their symbolic references. In the first table, the class file C declares the static fields x1, x2, and x3. In the second table, the class file A declares the static fields x1 and x2, and in the third table the class file B declares the static fields x1, x2, and x3. Our strategic objective in this particular case is to assign each static field within the class files C, A, and B a unique absolute address. Collectively, there are eight static fields declared between these class files C.x1::-, C.x2::-, C.x3::-, A.x1::-, A.x2::-, B.x1::-, B.x2::-, B.x3::- . A solution to the absolute address assignment problem would be: C.x1:::0, C.x2:::1, C.x3:::2, A.x1:::3, A.x2:::4, B.x1:::5, B.x2:::6, B.x3:::7. We would like to point out that the order of the static fields is irrelevant. The TL solution to this problem is given in the following section.

3.1 Static Field Address Calculation

Figure 9 shows a TL program for assigning unique addresses to static fields found within a Java application. In particular, it is the the strategy *assign_address* that assigns a unique address to each static field in the Java application. Furthermore, the address assignments will take into account the

$inc(x)$:	$sfield[[key_1 :: addr_1]] \rightarrow sfield[[key_1 :: addr_2]]$
	if	$addr_2 \ll addr_1 + x$
$sfield_counter(x)$:	$transient(inc(0)) \lt+ hide(inc(x))$
$make_sfield_counter$:	$sfield[[d_1.d_2.B]] \rightarrow sfield_counter(1)$
		$\lt+ sfield[[d_1.d_2.C]] \rightarrow sfield_counter(2)$
		$\lt+ sfield[[d_1.d_2.I]] \rightarrow sfield_counter(4)$
		$\lt+ sfield[[d_1.d_2.J]] \rightarrow sfield_counter(8)$
		$\lt+ sfield[[d_1.d_2.S]] \rightarrow sfield_counter(2)$
		$\lt+ sfield[[d_1.d_2.Z]] \rightarrow sfield_counter(1)$
$assign_addresses$:	$app_0 \rightarrow TDL(lcond_tdl\ make_sfield_counter\ app_0)\ app_0$

Figure 9: A strategic program for incrementing static fields

memory requirements for each static field. In the solution given, the addition operator has the following semantics:

$$addr_1 + y \stackrel{def}{=} \left\{ \begin{array}{ll} addr[[z]] & \text{if } \exists x : addr_1 = addr[[x]] \text{ and } x \text{ is of type integer and } z = x + y \\ addr[[y]] & \text{if } addr_1 = addr[[x]] \text{ and the value of } x \text{ is } - \end{array} \right.$$

Within $assign_address$, the strategic expression $(lcond_tdl\ make_sfield_counter\ app_0)$ will traverse the application app_0 and apply the strategy $make_sfield_counter$ to every static field. Depending upon the type of static field encountered, $make_sfield_counter$ will generate an appropriate call to the strategy $sfield_counter$. For example, if the descriptor of the static field is I then $sfield_counter$ will be called with the integer value 4 which denotes the space requirements (in bytes) of an integer field.

When given an integer value x , the strategy $sfield_counter$ will generate a left-biased composition of a $transient$ and $hide$ strategy. The left-biased composition of this composite strategy will have the effect of a summation when applied to the static field elements in app_0 . To see how this works, let us consider an application app_0 containing only the following static fields: $sfield[[C.x1.I :: -]]$, $sfield[[C.x2.J :: -]]$, and $sfield[[C.x3.S :: -]]$. The following tables shows the strategy resulting from the evaluation the strategic expression $(lcond_tdl\ make_sfield_counter\ app_0)$.

$transient(inc(0)) \lt+ hide(inc(4)) \lt+$
$transient(inc(0)) \lt+ hide(inc(8)) \lt+$
$transient(inc(0)) \lt+ hide(inc(2))$

The following figures provide a trace of the application of the above strategy to the static fields in app_0 . Figure 10 shows the value of the strategy and static fields prior to application.

$transient(inc(0)) \lt+ hide(inc(4)) \lt+$ $transient(inc(0)) \lt+ hide(inc(8)) \lt+$ $transient(inc(0)) \lt+ hide(inc(2))$	$sfield[[C.x1.I :: -]] sfield[[C.x2.J :: -]] sfield[[C.x3.S :: -]]$
---	---

Figure 10: Initial configuration of strategy and app_0

$skip \lt+ hide(inc(4)) \lt+$ $transient(inc(0)) \lt+ hide(inc(8)) \lt+$ $transient(inc(0)) \lt+ hide(inc(2))$	$sfield[[C.x1.I :: 0]] sfield[[C.x2.J :: -]] sfield[[C.x3.S :: -]]$
--	---

Figure 11: Configuration of strategy and app_0 after application of the first transient to the first sfield

In Figure 11, we see how the application of the first *transient* to the first field has caused the field to be assigned the absolute address 0. The evaluation of the *transient* causes it to be reduced to *skip* and its observation by the left-biased choice combinator causes the application of the strategy to stop, at which point the traversal proceeds on to the next static field.

Figure 12 shows the result of applying the strategy $hide(inc(4))$ to the second static field. Since a *hide* strategy cannot be observed by the left-biased choice combinator the application of the strategy continues and applies the second *transient*.

Figure 13 shows the result of applying the second *transient* strategy to the second field. Again, the evaluation of the *transient* causes it to be reduced to *skip* and its observation by the left-biased choice combinator causes the application of the strategy to stop, at which point the traversal proceeds on to the next static field.

Figures 14, 15 and 16 trace the strategy applications to the third static field. In particular, the third static field will be rewritten by two *hide* strategies followed by a *transient* strategy.

$skip \lt+ hide(inc(4)) \lt+$ $transient(inc(0)) \lt+ hide(inc(8)) \lt+$ $transient(inc(0)) \lt+ hide(inc(2))$	$sfield[[C.x1.I :: 0]] sfield[[C.x2.J :: 4]] sfield[[C.x3.S :: -]]$
--	---

Figure 12: Configuration of strategy and app_0 after application of the first hide strategy to the second sfield

$skip$ $\langle + \text{hide}(inc(4)) \rangle \langle +$ $skip$ $\langle + \text{hide}(inc(8)) \rangle \langle +$ $transient(inc(0)) \langle + \text{hide}(inc(2))$	$sfield[[C.x1.I :: 0]]$ $sfield[[C.x2.J :: 4]]$	$sfield[[C.x3.S :: -]]$
---	--	-------------------------

Figure 13: Configuration of strategy and app_0 after application of the second transient to the second sfield

$skip$ $\langle + \text{hide}(inc(4)) \rangle \langle +$ $skip$ $\langle + \text{hide}(inc(8)) \rangle \langle +$ $transient(inc(0)) \langle + \text{hide}(inc(2))$	$sfield[[C.x1.I :: 0]]$	$sfield[[C.x3.S :: 4]]$ $sfield[[C.x2.J :: 4]]$
---	-------------------------	--

Figure 14: Configuration of strategy and app_0 after application of the first hide strategy to the third sfield

$skip$ $\langle + \text{hide}(inc(4)) \rangle \langle +$ $skip$ $\langle + \text{hide}(inc(8)) \rangle \langle +$ $transient(inc(0)) \langle + \text{hide}(inc(2))$	$sfield[[C.x1.I :: 0]]$	$sfield[[C.x2.J :: 4]]$ $sfield[[C.x3.S :: 12]]$
---	-------------------------	--

Figure 15: Configuration of strategy and app_0 after application of the second hide strategy to the third sfield

<i>skip</i>	<+ <i>hide(inc(4))</i> <+	<i>sfield</i> [[<i>C.x1.I</i> :: 0]]	
<i>skip</i>	<+ <i>hide(inc(8))</i> <+		<i>sfield</i> [[<i>C.x2.J</i> :: 4]]
<i>skip</i>	<+ <i>hide(inc(2))</i>		<i>sfield</i> [[<i>C.x3.S</i> :: 12]]

Figure 16: Configuration of strategy and *app₀* after application of the third transient strategy to the third sfield

Static Field	Address
<i>C.x1.I</i>	0
<i>C.x2.J</i>	4
<i>C.x3.S</i>	12
<i>A.x1.I</i>	14
<i>A.x2.J</i>	18
<i>A.x3.C</i>	26
<i>B.x1.B</i>	28
<i>B.x2.Z</i>	29
<i>B.x3.I</i>	30

Figure 17: Address assignments for static fields

When applied to the application in the example shown in Section 3 the absolute address assignments shown in Figure 17 result.

Of course, these assignments will be embedded within the structure of the class files in the application and are presented here in summarized form.

4 HATS: A Restricted Implementation of *TL*

HATS 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. All of the examples discussed in this article have been implemented in HATS. HATS runs on Windows NT/2000/XP and Unix-based platforms and is freely available [18].

5 Conclusion

Strategic programming solution often require *data* to be moved throughout a term structure (e.g., from one subterm to another). The development of *TL* is based on the premise that higher-order rewriting

provides a mechanism for dealing with the movement of such data conforming to the tenets of rewriting. In a higher-order framework, the use of auxiliary data is expressed as rule. Instantiation of such rules can be done using standard (albeit higher-order) mechanisms controlling rule application (e.g., traversal). Typically, a traversal-driven application of a higher-order rule will result in a number of instantiations. If left unstructured, these instantiations can be collectively seen as constituting a rule base whose creation takes place dynamically. However, the utility of dynamically created unstructured rule bases is limited. Thus, TL also lifts the notion of strategy construction to the higher-order. That is, instantiations of rules are structured to form strategic expressions rather than rule bases. Nevertheless, in many cases, simply lifting first-order control mechanisms to the higher-order does not permit the construction of strategic expressions that are sufficiently refined. This difficulty is alleviated though the introduction of the *transient* and *hide* combinators. The interplay between the *transient* and *hide* combinators and more traditional control mechanisms enables a variety of strategies to be elegantly expressed in a higher-order setting.

At present we are exploring the addition of one more unary combinator into the framework of TL. We call this combinator *opaque*. The application of a strategy enclosed in an *opaque* combinator cannot be observed by the *transient* combinator. Thus, an *opaque* prevents the reduction to *skip* that a *transient* would normally initiate. We are presently exploring the consequences of such an extension with promising preliminary results.

References

- [1] 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.
- [2] H. Cirstea and C. Kirchner. *Intoduction to the rewriting calculus*. INRIA Research Report RR-3818, December 1999.
- [3] 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.
- [4] P. Jansson and J. Jeuring. *Polyp - a polytypic programming language extension*. In Conference record of POPL'97, pages 470-482. ACM Press, 1997.
- [5] R. Lämmel and J. Visser. *Type-safe Functional Strategies*. In Scottish Functional Programming Workshop, July 2000. 7.3
- [6] 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.
- [7] 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.
- [8] R. Lämmel. *Typed Generic Traversal With Term Rewriting Strategies*. Journal of Logic and Algebraic Programming, Vol 54, pp 1-64, 2003.

- [9] R. Lämmel, E. Visser, and J. Visser. *The Essence of Strategic Programming*. Draft.
- [10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification 2nd Edition*. Addison-Wesley, Reading, Massachusetts, 1999.
- [11] J. A. McCoy. *An Embedded System For Safe, Secure And Reliable Execution Of High Consequence Software*. Proceedings of the 5th IEEE International High-Assurance Systems Engineering Symposium, Nov. 2000.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] E. Visser. *Language Independent Traversals for Program Transformation*. In Johan Jeuring, editor, Workshop on Generic Programming (WGP'00), Ponte de Lima, Portugal, July 2000.
- [16] E. Visser. *Strategic Pattern Matching*. In: Rewriting Techniques and Applications (RTA '99), Trento, Lecture Notes in Computer Science (1999).
- [17] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT 0.9. Draft, November 2003.
- [18] V. L. Winter. The Hats System. <http://faculty.ist.unomaha.edu/winter/hats-uno/HATSWEB/index.html>
- [19] V.L. Winter, S. Roach, G. Wickstrom. *Transformation-Oriented Programming: A Development Methodology for High Assurance Software*. Advances in Computers vol 58, to appear.
- [20] V. L. Winter and M. Subramaniam. *The Transient Combinator, Higher-Order Strategies, and the Distributed Data Problem*.

Distribution:

- 1 MS0510 Greg Wickstrom, 2116
- 1 MS0510 James McCoy, 2116
- 1 MS0510 Anna Schauer, 2116

- 2 Victor Winter
PKI 174 C
1110 South 67th Street
Omaha, NE 68182

- 1 MS9018 Central Technical Files, 8945-1
- 2 MS0899 Technical Library, 9616