

# A Prototype of a Generic Weaver

Victor Winter<sup>1</sup>, Mansour Zand<sup>1</sup>, Harvey Siy<sup>1</sup>, and Günter Kniesel<sup>2</sup>

<sup>1</sup> University of Nebraska at Omaha, Omaha NE 68182, USA,  
vwinter@mail.unomaha.edu,

WWW home page: <http://faculty.ist.unomaha.edu/winter/>

<sup>2</sup> University of Bonn, Bonn, Germany

**Abstract.** A *generic weaver* is presented capable of realizing the weaving function over a large class of languages. There are several reasons why such a weaver is interesting. First, properties that can be proven about the weaver hold for all languages that fall within the domain of the weaver. Second, the problem of constructing a weaver for a particular language is reduced to constructing a language falling within the domain of the weaver.

## 1 Overview

In the realm of software development, type systems have proven to be extremely useful. So much so that the type system often forms the foundation of modern programming language designs. Specifically, language constructs are designed around the type system (in contrast to the type system being designed around the language constructs).

Inspired by the philosophy described in the previous paragraph, we explore language design that is centered around aspect-orientation. In particular, we identify a syntactic property  $P_{\mathcal{A}}$  and semantic property  $Q_{\mathcal{A}}$  that, if satisfied by a language  $\mathcal{L}$ , make  $\mathcal{L}$  intrinsically suited to a particular aspect-oriented idiom as defined by an aspect language  $\mathcal{A}$ . Given a definition of  $P_{\mathcal{A}}$  and  $Q_{\mathcal{A}}$  one can formally define the set of all such languages as follows:

$$\mathcal{D} = \{\mathcal{L} \mid P_{\mathcal{A}}(\mathcal{L}) \wedge Q_{\mathcal{A}}(\mathcal{L})\} \quad (1)$$

Over this domain, one can postulate the existence of a generic weaver  $\mathcal{D}_{\mathcal{W}}$  capable of producing the tangled form of any program  $prog_i \in \mathcal{L} \forall \mathcal{L} \in \mathcal{D}$ . In effect, such a generic weaver is fixed on  $\mathcal{A}$  and parameterized on  $\mathcal{L}$ .

In order for the weaver  $\mathcal{D}_{\mathcal{W}}$  to be possible, constructs of  $\mathcal{L}$ , which are to be subjected to aspect-oriented analysis and manipulation, must be described in generic fashion (i.e., in a manner that is common to all  $\mathcal{L} \in \mathcal{D}$ ). It is with respect to these generic constructs that weaving takes place. However, the set of strings described by  $\mathcal{L}$  should also remain unchanged. This conflict can be resolved by defining a set of syntactic categories  $\mathcal{W}$  together with an approach for linking these syntactic categories to the syntax of  $\mathcal{L}$ . The scope of the weaver  $\mathcal{D}_{\mathcal{W}}$  is exclusively limited to the syntactic categories defined in  $\mathcal{W}$ . In turn, the

determination of what constitutes a syntactic category in  $\mathcal{W}$  is driven by  $\mathcal{A}$ . A semantic link is therefore also established between  $\mathcal{W}$  and  $\mathcal{A}$ .

Let  $\langle \mathcal{L}, \mathcal{A} \rangle_{\mathcal{W}}$  denote a context-free grammar describing our domain of discourse. That is,  $\langle \mathcal{L}, \mathcal{A} \rangle_{\mathcal{W}}$  describes the union of the non-aspect language  $\mathcal{L}$  with the aspect language  $\mathcal{A}$ , linked by  $\mathcal{W}$  over which weaving is possible by  $\mathcal{D}_{\mathcal{W}}$ .

There are several reasons why a generic weaver is interesting. First, properties that can be proven about such a weaver will hold for all languages  $\mathcal{L}$  that fall within the domain of the weaver. Second, the problem of constructing a weaver for a particular language (e.g., a domain-specific aspect language) is reduced to constructing a language satisfying  $P_{\mathcal{A}}$  and  $Q_{\mathcal{A}}$ .

This paper has two goals: (1) the development a systematic approach for constructing  $\langle \mathcal{L}, \mathcal{A} \rangle_{\mathcal{W}}$ , and (2) the presentation of an implementation of  $\mathcal{D}_{\mathcal{W}}$ .

The rest of the paper is organized as follows: Section 2 discusses the properties  $P_{\mathcal{A}}$  and  $Q_{\mathcal{A}}$ . Section 3 presents a concrete example of a language  $\langle \mathcal{L}_i, \mathcal{A} \rangle_{\mathcal{W}_i}$ . Section 4 gives a brief overview of the program transformation system that is used to implement the generic weaver  $\mathcal{D}_{\mathcal{W}}$ . The implementation of  $\mathcal{D}_{\mathcal{W}}$  is given in Section 5. Section 6 concludes.

## 2 Language Properties

Our approach to identifying language properties is driven by two factors: The first factor is the creation of a syntactic framework where join points are members of syntactic categories that can easily be recognized. The second factor is the creation of a semantic framework in which the composition between join points and advice is straightforward.

### 2.1 The Syntactic Property $P_{\mathcal{A}}$

Informally stated, the syntactic property  $P_{\mathcal{A}}$  restricts a  $\mathcal{L}$  in the following way:

- $L$  and  $A$  must be syntactically compatible.
- Each join point type  $jp$  within a program in  $\mathcal{L}$  can only be generated via a derivation containing a syntactic category from which the join point type  $jp$  can be inferred.
- Each join point environment type within a program in  $\mathcal{L}$  can only be generated via a derivation containing a syntactic category from which the join point environment type  $env_j$  can be inferred.

**The Syntactic Compatibility of  $\mathcal{L}$  and  $\mathcal{A}$**  Let  $G_{\mathcal{L}} = (V_L, T_L, P_L, S_L)$  denote a context-free grammar describing the language  $\mathcal{L}$  where  $V_L$  denotes the set of nonterminal symbols,  $T_L$  denotes the set of terminal symbols,  $P_L$  denotes the set of production rules, and  $S_L$  denotes the start symbol. Let  $G_{\mathcal{A}} = (V_A, T_A, P_A, S_A)$  denote a context-free grammar describing the aspect language  $\mathcal{A}$ .

In order to be syntactically compatible, the grammars  $G_{\mathcal{L}}$  and  $G_{\mathcal{A}}$  must share their understanding of two distinguishable kinds of identifiers, but should

otherwise be disjoint. The first kind of shared identifier denotes typical *identifiers* that can occur in  $\mathcal{L}$  and we require they be generated using only the production  $id ::= ident$ , where  $ident$  denotes a regular expression defining the set of identifier tokens. The second kind of shared identifier denotes *aspect identifiers* that can be used to name various entities belonging to  $\mathcal{A}$  (e.g., advice functions) and we require they be generated using only the production  $id.a ::= ident.a$ , where  $ident.a$  denotes a regular expression defining the set of aspect identifier tokens.

**Definition 1.** Given two language grammars  $G_{\mathcal{L}_i}$  and  $G_{\mathcal{A}}$ , the predicate **compatible**( $G_{\mathcal{L}}, G_{\mathcal{A}}$ ) holds, if the following conditions are satisfied:

1. Constraints on V:
  - (a)  $\{id, id.a\} \subseteq V_L \wedge \{id, id.a\} \subseteq V_A$
  - (b)  $V_L - \{id, id.a\} \cap V_A = \emptyset$
  - (c)  $L(id) \cap L(id.a) = \emptyset$
2. Constraints on T
  - (a)  $(ident \cup ident.a) \subseteq T_L \wedge (ident \cup ident.a) \subseteq T_A$
  - (b)  $T_L - (ident \cup ident.a) \cap T_A = \emptyset$
3. Constraints on P
  - (a)  $\{id ::= ident, id.a ::= ident.a\} \subseteq P_L$
  - (b)  $\{id ::= ident, id.a ::= ident.a\} \subseteq P_A$

**The Syntactic Categories of  $\mathcal{W}$**  As was mentioned in Section 1 the languages  $\mathcal{L}$  and  $\mathcal{A}$  are linked by a set of syntactic categories that are defined in  $\mathcal{W}$ . Linking, while driven by the semantics of  $\mathcal{A}$ , alters only the syntax of  $\mathcal{L}$  and centers around two models: the *join point model* and the *join point environment model*. A *join point model* identifies constructs within  $\mathcal{L}_i$  where advice can be woven. A *join point environment model* identifies contexts (e.g., all join points within class C) within  $\mathcal{L}$  in which join point models can arise.

**Definition 2.** Let  $Model_{jp} = \{jp_1, \dots, jp_n\}$  denote a finite set of join point types. Concrete examples of join point types include *get*, *set*, and *call*.

**Definition 3.** Let  $Model_{env} = \{env_1, \dots, env_m\}$  denote a finite set of join point environment types. Concrete examples of join point environment types include structures such as classes and methods.

In practice, there are two kinds of model structures that should be considered: *atomic models* and *recursive models*. An atomic model contains only a single element (i.e., join point or join point environment). A recursive model may contain an arbitrary number of join points or join point environments. For example, the standard *set* join point, as it is defined in languages like AspectJ [2, 1], is atomic since it contains a single field. On the other hand, the *call* join point is an example of a recursive join point model, since it may contain other join point models (e.g., *call* and *get*) within its structure.

Given these considerations, our goal is to devise an approach to formally express the syntax of each distinct type  $\tau \in Model_{jp} \cup Model_{env}$  in a manner

that can easily be linked with the syntax of the language  $\mathcal{L}$  described in Section 1 thereby linking  $\mathcal{L}$  and  $\mathcal{A}$ . This link is accomplished by representing  $\tau$  in terms of a grammar *component* having the form  $G_\tau[B_1 \dots B_j]$ . The component  $G_\tau[B_1 \dots B_j]$  denotes a context-free grammar fragment whose start symbol is  $\tau$  containing nonterminal symbols  $B_1, \dots, B_j$  which are non-generating<sup>3</sup>.

Let  $G_{\mathcal{L}} = (V_L, T_L, P_L, S_L)$  denote a context-free grammar describing  $\mathcal{L}$ . Let  $G_\tau[B_1 \dots B_j] = (V_\tau, T_\tau, P_\tau, S_\tau)$  denote a context-free grammar describing a component to be linked to  $G_{\mathcal{L}}$ . Let  $G_{\mathcal{L},\tau} = (V, T, P, S)$  denote an instance of  $G_\tau[B_1 \dots B_j]$  linked to  $G_{\mathcal{L}}$ . Such a *linked instance* can be created in the following manner:

**1. Construction:**

- (a) **Import**  $\tau$ : Make  $\tau$  reachable by a production of the form  $D ::= \alpha \tau \beta$  where  $D \in V_L \wedge \alpha, \beta \in (V_L \cup T_L)^*$ .
- (b) **Export**  $B_i$ : For each  $i : 1 \leq i \leq j$ , make  $B_i$  generating by a production of the form  $B_i ::= \alpha$  where  $\alpha \in (V_L \cup T_L)^*$ .

**2. Properties:**

- (a)  $V_L \cap V_\tau = \emptyset$
- (b)  $T_L \cap T_\tau = \emptyset$

**Definition 4.** *The set of syntactic categories  $\mathcal{W}$  is the union of each linked instance  $G_{\mathcal{L},\tau}$  for all  $\tau \in Model_{jp} \cup Model_{env}$ .*

$$\mathcal{W} = \bigcup_{\tau \in \mathcal{T}} G_{\mathcal{L},\tau} \text{ where } \mathcal{T} = Model_{jp} \cup Model_{env} \quad (2)$$

## 2.2 The Semantic Property $Q_{\mathcal{A}}$

We believe that the aspect-oriented paradigm would greatly benefit from a semantic construct(s) that is to aspect orientation what the semaphore is to the multiprocessing environment. Specifically, the problem that our AO construct must solve is the precise insertion of a sequence of computational steps (e.g., the sequence resulting from the execution of the advice body) into another sequence of computational steps (e.g., the sequence containing the join point).

In this paper, we consider two semantic requirements: First, all advice must be implementable as a polymorphic function whose parameter passing semantics is non-strict. Second, there must exist a function `eval` that when applied to an advice function parameter will cause the parameter to be evaluated. In effect, we have a mechanism that is similar to a *thunk*.

For example, let  $f$  denote an advice function applicable to a particular join point type  $jp$  defined as follows:

<sup>3</sup> A nonterminal symbol  $B$  is non-generating if  $\neg \exists w \in T : B \xRightarrow{*} w$ ; otherwise  $B$  is generating.

$$fun\ f\ x_{jp} = (advice\_body; eval(x_{jp})) \quad (3)$$

Let  $e_{jp}$  denote a value of type  $jp$ . Under the assumptions stated, weaving can be realized by the following rewrite:

$$weaving : e_{jp} \rightarrow f(e_{jp}) \quad (4)$$

### 3 An Example: $\langle \mathcal{L}_i, \mathcal{A} \rangle_{\mathcal{W}_i}$

In this section, we present an instance of  $\langle \mathcal{L}_i, \mathcal{A} \rangle_{\mathcal{W}_i}$ . Our goal here is to show how the pieces of  $\langle \mathcal{L}_i, \mathcal{A} \rangle_{\mathcal{W}_i}$  must be related in order for the result to be well-formed (i.e., subject to weaving by  $\mathcal{D}_{\mathcal{W}}$ ). For the sake of readability, we have somewhat relaxed the requirements that grammars be disjoint. In particular, left and right parenthesis arise in  $\mathcal{L}_i$ ,  $\mathcal{A}$ , and  $\mathcal{W}_i$ . Curly braces and the semi-colon arise in both  $\mathcal{L}_i$  and  $\mathcal{A}$ .

A BNF grammar for the language components  $\mathcal{L}_i$  and  $\mathcal{A}$  is shown in Figure 1.

The language  $\mathcal{L}_i$  is a small OO language that supports class, method, and field definitions as well as a handful of imperative constructs. It is important to note that in  $\mathcal{L}_i$  class and method definitions can be nested allowing for a rich set of join point environments. Similarly, method calls can also be nested allowing for a rich set of join point structures.

The language  $\mathcal{A}$  is a small aspect language supporting only weaving based on static analysis. Three standard join point types and two join point environment types are defined:

$$Model_{jp} = \{\mathbf{get}, \mathbf{set}, \mathbf{call}\}$$

$$Model_{env} = \{\mathbf{class\_env}, \mathbf{method\_env}\}$$

$\mathcal{A}$  provides only two composition types: **before** and **after**. Pointcut expressions can include wildcard symbols as well as paths (i.e., environments) describing the static structures in which weaving should be performed. The semantics of  $\mathcal{A}$  supports weaving multiple advice elements to a single join point, but does not prioritize weaving order. It is worth noting that the pointcut expressions supported by  $\mathcal{A}$  do not include type information (e.g., `int`) as is usually the case. The body of an advice declaration is also left unspecified.

A BNF grammar describing the syntactic categories of  $\mathcal{W}_i$  is shown in Figure 2. To enhance readability, in Figure 2, a comment is added to every production rule indicating whether the rule is an Import rule, an Export rule, or an Internal rule used by  $\mathcal{D}_{\mathcal{W}}$ .

<b>The Language <math>\mathcal{L}_i</math></b>	
L	::= class_def_list
class_def_list	::= class_def class_def_list   $\epsilon$
class_keyword	::= "class"
class_body	::= [ "extends" id ] "{" dec_list "}"
dec_list	::= dec [ "," ] dec_list   $\epsilon$
dec	::= class_def   field_def   method_def
type	::= "int"   "bool"
method_body	::= args "{" stmt_list "}"
field_def	::= type id
args	::= "(" [ arg_list ] ")"
arg_list	::= arg [ "," arg_list ]
arg	::= type id
stmt_list	::= stmt ";" stmt_list   $\epsilon$
stmt	::= assign   cond   block   dec   return_expr
assign	::= lvalue "=" expr
cond	::= "if" expr "then" stmt "else" stmt
block	::= "{" stmt_list "}"
return_expr	::= "return" expr
expr_list	::= expr [ "," expr_list ]
expr	::= expr "+" expr %PREC "L1"
expr	::= base
base	::= num
num	::= number
<b>The Aspect Language <math>\mathcal{A}</math></b>	
A	::= aspect_def_list jp_defs
aspect_def_list	::= aspect_def aspect_def_list   $\epsilon$
aspect_def	::= "aspect" id "{" advice_def_list "}"
advice_def_list	::= advice_def advice_def_list   $\epsilon$
advice_def	::= initial_a   final_a
initial_a	::= "advice" id_a type_a pc body_a
final_a	::= head_f "=" body_f
head_f	::= "fun" id_a "(" id ")"
body_f	::= "(" body_a ";" eval_jp ")"   "(" eval_jp ";" body_a ")"
eval_jp	::= "eval" "(" id ")"
type_a	::= "before"   "after"
body_a	::= "{" ... "}"
pc	::= jp_match
jp_match	::= jp_type "(" match_expression ")"
jp_type	::= "set"   "get"   "call"
match_expression	::= full_match_path wild_id
full_match_path	::= full_match_path wild_id "."   $\epsilon$
<b>Shared between <math>\mathcal{L}_i</math> and <math>\mathcal{A}</math></b>	
id	::= ident
id_a	::= ident_a

**Fig. 1.** The Components  $\mathcal{L}_i$  and  $\mathcal{A}$

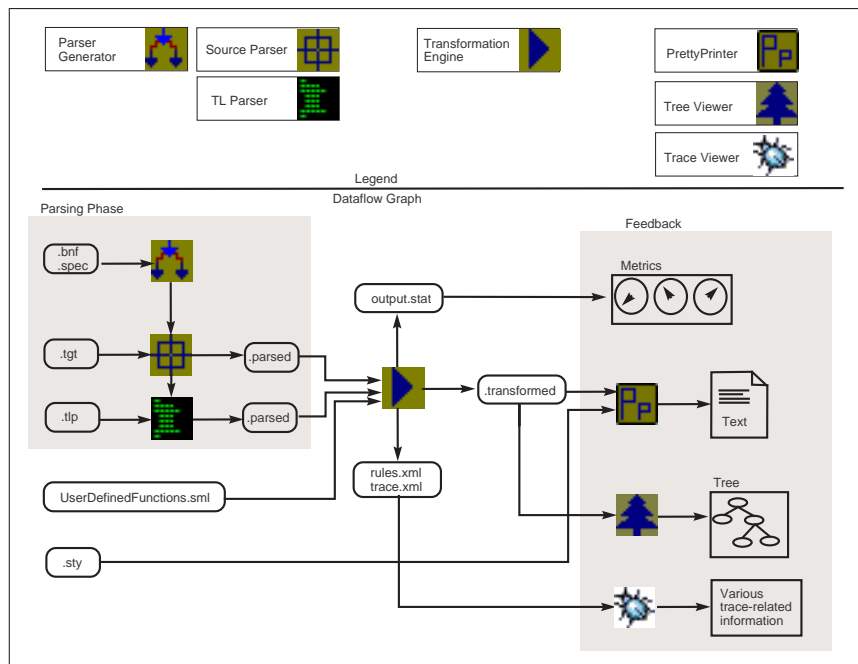
<b>Class Environment Model</b>		
Import:	class_def	::= class_env
Export:	head_c	::= class_keyword
Export:	body_c	::= class_body
Internal:	class_env	::= head_c id body_c
<b>Method Environment Model</b>		
Import:	method_def	::= method_env
Export:	head_m	::= type
Export:	body_m	::= method_body
Internal:	method_env	::= head_m id body_m
<b>Join Point Model</b>		
Import:	lvalue	::= set_context
Import:	base	::= get_context   call_context
Export:	set_model	::= id
Export:	get_model	::= id
Export:	call_model	::= id tuple
Export:	tuple	::= "(" [ expr_list ] ")"
Internal:	set_context	::= set_model   advice_call
Internal:	get_context	::= get_model   advice_call
Internal:	call_context	::= call_model   advice_call
Internal:	advice_call	::= id_a "(" jp_context ")"
Internal:	jp_context	::= set_context   get_context   call_context

**Fig. 2.** The Syntactic Categories of  $\mathcal{W}_i$

## 4 Program Transformation Using HATS

HATS [7] is an IDE that provides a variety of capabilities germane to transformation-based software development. These capabilities include: (1) an engine where transformation can be performed through the execution of programs written in a special purpose language called TL, (2) a parser generator having GLR-like capabilities accepting as input extended BNF grammars together with precedence and associativity rules, (3) an abstract pretty printer, (3) graphical display facilities for viewing the structure of parse trees, (4) text editors, (5) a display showing various metrics associated with TL program execution (e.g., number of rewrites applied, etc.) and (6) some rudimentary tracing capabilities to assist in debugging transformations [9].

A dataflow diagram showing the interaction of various HATS components in the transformational process is given in Figure 3. The HATS IDE is implemented in Java and TL is implemented in an interpretive fashion in SML. HATS is freely available.



**Fig. 3.** The Architecture of HATS from a Dataflow Perspective

## 4.1 TL

TL is a language that has been developed exclusively for describing transformation-based computation [8][6]. TL has several attributes that make it well suited to implementing the weaver  $\mathcal{D}_W$  described in this paper:

- TL supports wildcard matching on identifiers (i.e., lexical tokens) as a primitive operation.
- TL supports the construction of higher-order strategies. These are useful for moving information from the aspect portion of a program (i.e.,  $\mathcal{A}$ ) to the target portion of the program (i.e.,  $\mathcal{L}_i$ ).
- TL supports a rich environment for specifying generic traversals both first-order as well as higher-order. Generic traversal capabilities are essential in order for  $\mathcal{D}_W$  to be generic.
- TL supports a novel set of unary strategic combinators including *transient*, *raise*, and *opaque*.
- TL has a semantics that treats strategy application failure as a *nop* on terms. When extended with the various unary combinators described, this semantic paradigm provides a rich environment in which control can be expressed.

The principle artifacts manipulated during the execution of a TL program are *parse trees*, which we will also refer to as *terms*. TL provides a notation for describing parse tree structures relative to a given (assumed) grammar  $G$ . Trees expressed using this notation are referred to as *patterns*.

For example, suppose we are given a grammar where the derivation  $stmt \xrightarrow{\pm} id = 5$  is possible. The pattern  $stmt[id_1 = 5]$  describes a tree corresponding to this derivation. In this context, the subscripted variable  $id_1$  denotes a typed variable quantified over the domain of all trees having the nonterminal  $id$  as their root node.

In general, a pattern of the form  $A[\alpha']$  is well-formed if and only if the derivation  $A \xrightarrow{\pm} \alpha$  is possible according to the grammar and  $\alpha'$  is obtained from  $\alpha$  by subscripting all nonterminals occurring in  $\alpha$ .

In TL, transformation is accomplished by the application of rewrite rules to terms (i.e., parse trees). In TL, a rewrite rule has the following syntactic structure:

$$lhs \rightarrow rhs [ \textit{if condition} ]$$

where  $[$  and  $]$  are syntactic meta symbols indicating that the enclosed section (i.e., the conditional portion) of a rule is optional. In order for a rule to be well-formed it is necessary that  $lhs$  be a *pattern*, that  $rhs$  be an expression whose order is greater-than or equal-to 0, and that *condition* be an expression consisting of one or more *match expressions* combined using the Boolean connectives  $\{ \textit{and}, \textit{or}, \textit{not} \}$ .

A *match expression* is a first-order match between two patterns. Let  $t_1$  denote a pattern, possibly non-ground, and let  $t_2$  denote a ground pattern. The expression  $t_1 \ll t_2$  denotes a match expression and evaluates to *true* if and only if a substitution  $\sigma$  can be constructed so that  $\sigma(t_1) = t_2$ .

**Standard Control Mechanisms** In contrast to a pure rewriting system, TL requires that the application of rewrite rules to terms (i.e., the transformation process) be explicitly controlled. A specification of such control is called a *strategy*. The presence of strategies classifies TL as a *strategic programming* language [3]. Other examples of strategic programming systems include Stratego [5] and Strafinski [4].

In TL, a strategy is an expression composed of various elements including *rewrite rules*, *rewrite rule abstractions* (i.e., rule labels), *combinators*, and *traversals*. The application of a strategy  $s$  to a tree  $t$  is expressed using the traditional function application syntax:  $s(t)$ . Within a strategy, control is expressed using the following standard control mechanisms:

1. The application of rules to a single term is controlled as follows:
  - At the rule level control is exercised through *first-order matching* and optional rule *conditions* (i.e., conditional rewrite rules).
  - At the strategy level control is exercised through *combinators*. The set of standard binary strategic combinators includes the sequential composition ( $<;$ ) and left-biased choice ( $<+$ ) combinators. Let  $s_1$  and  $s_2$  denote two strategies. The composition  $s_1 <; s_2$  is a strategy that when applied to a term  $t$  will first apply  $s_1$  and then apply  $s_2$ . In contrast, the composition  $s_1 <+ s_2$  is a strategy that when applied to a term  $t$  will first apply  $s_1$  to  $t$  and only apply  $s_2$  (to  $t$ ) if the application of  $s_1$  to  $t$  fails.
2. The application of a strategy to a term structure is controlled by *traversals*. TL provides a rich framework for defining traversals as well as a library of standard generic first-order. Traversals specify the order in which the sub-terms of a given term are visited. Standard first-order traversals include top-down left-to-right (*TDL*) and bottom-up left-to-right (*BUL*). Let  $s$  denote a strategy. In TL, the expression  $TDL\{s\}$  denotes a strategy that will traverse the term to which it is applied in a top-down left-to-right fashion and apply the strategy  $s$  to every (sub)term visited.

**Special Control Mechanisms** In addition to the standard strategic mechanisms described, TL also provides several mechanisms unique to TL. These mechanisms include a number of unary combinators and constructs that enable generic traversals to be generalized to higher-order strategies.

An important unary combinator in TL is the *transient* combinator. This combinator restricts a strategy so that it may be applied *at most once*. The “at most once” property is the hallmark of the *transient* combinator. Theoretically, a *transient* can be understood as follows. Let  $s$  denote a transient-free strategy enclosed in the context  $transient(\dots s \dots)$ , where the ellipsis denote a transient-free strategic context. Furthermore, suppose that  $s$  has just been successfully applied to a term. Under these conditions, the following strategy reduction occurs:

$$transient(\dots s \dots) \rightarrow SKIP \tag{5}$$

where *SKIP* is a strategic constant whose application to a term is never successful.

Transients open the door to *self-modifying* strategies. When using a traversal to apply a self-modifying strategy to a term, it becomes possible to apply a different strategy to every term encountered during a traversal.

The unary combinators *opaque* and *raise* compliment the semantics of the *transient* combinator. In particular, *opaque* restricts the strategy reduction that normally occurs when the contents strategy of a *transient* has been successfully applied, and the *raise* combinator widens the reduction. More specifically, the *raise* combinator causes the *transient* reduction to cascade to the next enclosing *transient*.

Assuming the same conditions as with our previous example, the semantics of *opaque* and *raise* can be formally described as follows:

$$\frac{\text{transient}(\dots \text{opaque}(s) \dots) \rightarrow \text{transient}(\dots \text{opaque}(s) \dots)}{\text{transient}(\dots \text{transient}(\dots \text{raise}(s) \dots) \dots) \rightarrow \text{SKIP}} \quad (6)$$

**Higher-Order Generic Traversal** In addition to unary combinators, TL also lifts the notion of generic traversal to higher-order strategies. In TL, a higher-order strategy is a strategy that when applied to a term returns a strategy as a result instead of returning a term. An abstract example of a rule having order  $k + 1$  is the following:

$$\text{pattern}_{k+1} \rightarrow \text{pattern}_k \rightarrow \dots \rightarrow \text{pattern}_0 \quad (7)$$

The successful application of this strategy to a term  $t$  yields the result:

$$\text{pattern}'_k \rightarrow \dots \rightarrow \text{pattern}'_0 \quad (8)$$

where  $\text{pattern}'_i$  is an instance of  $\text{pattern}_i$  that has been instantiated with the bindings obtained from  $\text{pattern}_{k+1} \ll t$ . Thus, higher-order rules are simply rules whose parameters are curried. However, their interplay with generic traversal is particularly interesting.

Let  $s^{n+1}$  denote a strategy of order  $n + 1$ . If  $s^{n+1}$  is applied to a term using a traversal, the strategy sequence  $\langle s_1^n, \dots, s_m^n \rangle$  will be produced (assuming that  $s^{n+1}$  applies successfully  $m$  times during this traversal). This sequence can be turned into a strategy by composing the  $s_i^n$  using a binary combinator such as  $\langle +$  or  $\langle ;$ . To generate these kinds of strategies, TL provides a library of higher-order traversals that includes *lcond\_tdl* and *lseq\_tdl*. The expression  $\text{lcond\_tdl}\{s^{n+1}\}(t)$  will traverse the term  $t$  in a top-down left-to-right (tdl) fashion and compose the resulting strategies using the combinator  $\langle +$ . The expression  $\text{lseq\_tdl}\{s^{n+1}\}(t)$  is similar and composes the results using the combinator  $\langle ;$ .

**DW:**  $LA[[L_{untangled} A_1]] \rightarrow LA[[L_{tangled} A_2]]$

if  $L_{tangled} \ll TD\{weave[A_1]\}(L_{tangled})$   
and  $A_2 \ll TDL\{transcribe\_advise\}(A_1)$

---

**meta\_call:**

$advise\_def[[advise\ id\_a_1\ type\_a_1\ get(full\_match\_path_{jpm}\ wild\_id_{jpm})\ body\_a_1]]$

$\rightarrow$

$transient($

$\quad lcond\_bul\{recognize\_env\}[full\_match\_path_{jpm}]$

$\quad <+$

$\quad BUL\{call\_context[[id_{jp}\ tuple_{jp}]]$

$\quad \rightarrow$

$\quad call\_context[[id\_a_1(id_{jp}\ tuple_{jp})]]$

$\quad if\ id_{jp} \ll wild\_id_{jpm}$

$\quad \}$

$)$

**meta\_get:** ...

**meta\_set:** ...

**weave:**  $lseq\_tdl\{meta\_call\ <+\ meta\_get\ <+\ meta\_set\}$

**Fig. 4.** Highlights from  $\mathcal{D}_W$  : Weaving

```

bad_context: class_env[[head_c1 id1 body_c1]]
  →
  class_env[[head_c1 id1 body_c1]]

  <+

  method_env[[head_m1 id1 body_m1]]
  →
  method_env[[head_m1 id1 body_m1]]

recognize_env: wild_id1
  →
  transient(
    class_env[[head_c1 id1 body_c1]]
    →
    class_env[[head_c1 id1 body_c1]]
    if wild_id1 << id1

    <+

    method_env[[head_m1 id1 body_m1]]
    →
    method_env[[head_m1 id1 body_m1]]
    if wild_id1 << id1

    <+

    raise(bad_context)

    <+

    opaque(ID)
  )

```

**Fig. 5.** Highlights from  $\mathcal{D}_W$ : Context Recognition

```

transcribe_before_advice:

  advice_def[[advice id_a1 before jp_type1(match_expression_jpm) body_a1]]
  →
  advice_def[[fun id_a1(x) = (body_a1; eval(x))]]

transcribe_after_advice: ...

transcribe_advice: transcribe_before_advice <+ transcribe_after_advice

```

Fig. 6. Highlights from  $\mathcal{D}_{\mathcal{W}}$ : Advice Transcription

## 5 The Weaver $\mathcal{D}_{\mathcal{W}}$

Highlights of the implementation of  $\mathcal{D}_{\mathcal{W}}$  are shown in Figure 5. Weaving is accomplished by invoking the strategy  $DW$  on an input term belonging to the language  $\langle \mathcal{L}_i, \mathcal{A} \rangle_{\mathcal{W}_i}$ . Such terms will always match the pattern  $LA[[L_{untangled} A_1]]$ . The transformed term  $LA[[L_{tangled} A_2]]$  is produced as follows: First, a weaving strategy is created based on  $A_1$ . This strategy is created through the evaluation of the higher-order expression  $weave[A_1]$ . The resulting strategy is then applied to  $L_{untangled}$  to produce its tangled form  $L_{tangled}$ . Second, the advice components in  $A_1$  are transcribed into functions yielding  $A_2$ . It is in this transcription step that the composition semantics of **before** and **after** are realized.

Within the conditional portion of  $DW$ , the higher-order strategic expression  $weave[A_1]$  represents an application of  $weave$  to the term  $A_1$ . Note that the identifier  $weave$  is actually just an abstraction for the higher-order strategy:

$$lseq\_tdl\{meta\_call \lt;+ meta\_get \lt;+ meta\_set\} \quad (9)$$

The evaluation of  $weave[A_1]$  causes  $A_1$  to be traversed in a top-down left-to-right fashion (tdl). To each term encountered during this traversal, the strategy  $meta\_call \lt;+ meta\_get \lt;+ meta\_set$  is applied. Each strategy,  $meta\_call$ ,  $meta\_get$ , and  $meta\_set$  corresponds to a particular join point type belonging to  $Model_{jp}$  as defined by the language  $\mathcal{A}$ .

The successful application of the meta-rule strategy to a term  $t_i$  results in the creation of a strategy  $\omega_i$ . In particular, a strategy of the form  $\omega$  is produced for every advice method encountered in  $A_1$ . The resulting  $\omega$  strategies are then composed using  $\lt;+$  (as defined by the semantics of  $lcond\_tdl$ ) to form the following weaver instance:

$$\omega_1 \lt;+ \dots \lt;+ \omega_n \quad (10)$$

Each strategy  $\omega_i$  in this weaver instance embodies a considerable amount of intelligence. Specifically, each  $\omega_i$  is able to select only those join points to which the corresponding advice is to be woven.

Let us take a look at the instance of  $\omega_i$  derived from an application of the strategy *meta\_call* to an *advice\_def* term  $t_i$ . The match with  $t_i$  provides values for  $id_{a_1}$ ,  $full\_match\_path_{jpm}$ , and  $wild\_id_{jpm}$ . These values are then used to perform the following simplification on the strategic value to be returned by *meta\_call*:

```

transient(
  lcond_bul{recognize_env}[full_match_path_{jpm}]
  <+
  BUL{call_context[[id_{jp} tuple_{jp}]]
    →
    call_context[[id_{a_1}(id_{jp} tuple_{jp})]]
    if id_{jp} ≪ wild_id_{jpm}
  }
)
→
transient(
   $\mathcal{E}$ 
  <+
  BUL{call_context[[id_{jp} tuple_{jp}]]
    →
    call_context[[id_{a_1}(id_{jp} tuple_{jp})]]
    if id_{jp} ≪ wild_id_{jpm}
  }
)

```

Here  $\mathcal{E}$  denotes a strategy that works together with its enclosing *transient* to assure that advice is only woven to join points occurring in the proper environments (e.g., within the method environment  $f$  occurring within the class environment  $C$ ). This will be discussed further in a moment. The strategy  $\mathcal{E}$  is conditionally composed with the strategy

```

BUL{call_context[[id_{jp} tuple_{jp}]]
  →
  call_context[[id_{a_1}(id_{jp} tuple_{jp})]]
  if id_{jp} ≪ wild_id_{jpm}
}

```

that we will refer to as  $\eta$  for the remainder of this discussion. The strategy  $\eta$  shown above, weaves advice to method call join points by passing these join points as a parameter to the advice function  $id_{a_1}$ . In order for such a weaving step to occur, two conditions must be met: First, control must be passed from  $\mathcal{E}$  to  $\eta$ . Second, the method identifier  $id_{jp}$  must match with  $wild\_id_{jpm}$ .

Since a call context is a recursive structure (e.g.,  $f(f(x))$ ), a bottom-up left-to-right traversal (*BUL*) is needed to weave the advice  $id_{a_i}$  to all join points  $id_{j_p}$  (e.g.,  $f1$ ,  $f2$ , etc.) that match the pointcut  $wild_{id_{j_{pm}}}$  (e.g.,  $f^*$ ). This concludes the discussion of the weaving strategy and we now turn our attention back to  $\mathcal{E}$ .

The strategy  $\mathcal{E}$  is obtained by applying the higher-order strategy  $lcond\_bul\{recognize\_env\}$  to the term  $full\_match\_path_{j_{pm}}$ . This term has the form:

$$wild\_id_1 . wild\_id_2 . \dots . wild\_id_n \quad (11)$$

where each  $wild\_id_i$  corresponds to a class/method environment (the weaver presented does not distinguish between a class environment and a method environment). For each  $wild\_id_i$  encountered in  $full\_match\_path_{j_{pm}}$ , a strategy is created which we will denote  $\varepsilon_i$ . The strategy resulting from the evaluation of  $lcond\_bul\{recognize\_env\}[full\_match\_path_{j_{pm}}]$  has the form:

$$\mathcal{E} = \varepsilon_1 <+ \varepsilon_2 <+ \dots <+ \varepsilon_n \quad (12)$$

In this strategy, each  $\varepsilon_i$  corresponds to an instance of:

```
transient(
  (* — Case 1: Positive Match — *)
  class_env[[head_c1 id1 body_c1]]
  →
  class_env[[head_c1 id1 body_c1]]
  if wild_id1 << id1

  <+

  method_env[[head_m1 id1 body_m1]]
  →
  method_env[[head_m1 id1 body_m1]]
  if wild_id1 << id1

  <+

  (* — Case 2: Negative Match — *)
  raise(bad_context)

  <+

  (* — Case 3: Short-circuit — *)
  opaque(ID)
)
```

As commented, this strategy consists of three cases: Case 1 consists of two rules (*class\_env* and *method\_env*) corresponding to rewrite identities that apply if the *class\_env/method\_env* identifier  $id_1$  can be matched with the given  $wild\_id_1$

(i.e., the *wild\_id* from which  $\varepsilon_i$  is derived). When such a rule fires the enclosing *transient* and its contents (i.e., all three cases) are reduced to *SKIP*. This implies that an environment matching the *wild\_id<sub>i</sub>* in *full\_match\_path<sub>jpm</sub>* has been encountered.

Case 2 arises when a *class\_env/method\_env* structure is encountered whose identifier cannot be matched with *wild\_id<sub>1</sub>*. In this situation, the application of strategy *bad\_context* succeeds which implies that the search for candidate join points can be terminated (i.e., one need not look any further into this structure). Enclosing the strategy *bad\_context* is the combinator *raise* which causes the outer transient strategy enclosing this inner transient strategy to reduce to *SKIP*. In other words:

$$transient(\dots \varepsilon_i \dots <+ \eta) \rightarrow SKIP \tag{13}$$

This reduction has the effect of removing strategy  $\eta$  responsible for weaving advice (described earlier in this section).

Case 3 arises when a term is encountered that does not correspond to an environment (e.g., a join point). In this case, we have not yet encountered the desired environment in which weaving should occur. As a result, we would like to prevent the strategy  $\eta$  from applying. Since all these strategies are conditionally composed, the application of  $\eta$  can be prevented by inserting into Case 3 a strategy that will always apply. The strategic constant *ID* is such a strategy. However, we also want to prevent the reduction to *SKIP* caused by enclosing *transient*. This effect can be achieved by enclosing *ID* in the combinator *opaque*. Thus, we have achieved the desired control.

We would like to point out that several variations on the weaving semantics presented here are possible. Several have been implemented in TL but were not included in this paper.

## 6 Conclusion

In this paper, an approach was described whereby the weaving function for a language  $\mathcal{L}$  can be realized by a generic weaver  $\mathcal{D}_{\mathcal{W}}$  provided that  $\mathcal{L}$  satisfies the properties  $P_{\mathcal{A}}$  and  $Q_{\mathcal{A}}$ . Using the higher-order transformation language *TL* the weaver  $\mathcal{D}_{\mathcal{W}}$  was implemented with a small number of strategies. In the resulting framework, weaving produces a program where advice has been transcribed into polymorphic functions having a non-strict parameter passing semantics and weaving consists of wrapping join points with calls to advice functions. A further transformational step could be considered in which the advice functions are actually in-lined at join points. Such in-lining can be useful when the software artifact over which weaving is performed is an informal document such as a requirements document.

## APPENDIX

### A An Example Program in Untangled and Tangled Form

#### A.1 Untangled Form

```
class A {
    int x1 = 1;
    int x2 = 2;
    int y1 = 2;
    int y2 = 4;
    int z = x1 + x2 + y1 + y2;

    class B1 {
        int z = x1 + x2 + y1 + y2;

        int f(int x, int y2) {
            return f(f(y2,5),f(z,z));
        }
    }

    class B2 {
        int z = x1 + x2 + y1 + y2;
    }

    class C {
        int z = x1 + x2 + y1 + y2;
    }

    int f(int x) {
        x = 5;
        z = f(f(y2,5),f(z,z));
        if true then x = x1 + y1 else x = x2 + y2;
        return z + x;
    }
}

// -----
aspect X {
    advice _getx before get(A*.B*.x1) { body_getx }
    advice _gety before get(y*) { body_gety }

    advice _setx before set(A*.z) { body_setz }
    advice _callf before call(A*.B*.f) { body_callf }
}
```

## A.2 Tangled Form

```
class A {
  int x1 = 1;
  int x2 = 2;
  int y1 = 2;
  int y2 = 4;
  int z = x1 + x2 + _gety(y1) + _gety(y2);
  class B1 {
    int z = _getx(x1) + x2 + _gety(y1) + _gety(y2);
    int f ( int x , int y2 ) {
      return _callf ( f(_callf(f(_gety(y2),5)),_callf(f(z,z))));
    }
  }
}
class B2 {
  int z = _getx(x1) + x2 + _gety(y1) + _gety(y2);
}
class C {
  int z = x1 + x2 + _gety(y1) + _gety(y2);
}
int f ( int x ) {
  x = 5;
  _setx(z) = f(f(_gety(y2),5),f(z,z));
  if true then x = x1 + _gety(y1) else x = x2 + _gety(y2);
  return z + x;
}
}
aspect X {
  fun _getx ( x ) = ( { body_getx } ; eval ( x ) )
  fun _gety ( x ) = ( { body_gety } ; eval ( x ) )
  fun _setx ( x ) = ( { body_setz } ; eval ( x ) )
  fun _callf ( x ) = ( { body_callf } ; eval ( x ) )
}
}
```

## References

1. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting Started with ASPECTJ. *Commun. ACM*, 44(10):59–65, 2001.

2. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
3. R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming. Oct. 2002.
4. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.
5. E. Visser, Z. e. A. Benaïssa, and A. Tolmach. Building program optimizers with rewriting strategies. In *third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
6. V. Winter. Strategy Construction in the Higher-Order Framework of TL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 124, 2004.
7. V. Winter and J. Beranek. Program Transformation Using HATS 1.84. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 378–396, 2006.
8. V. Winter and M. Subramaniam. The Transient Combinator, Higher-order Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.
9. V. L. Winter, C. Scalzo, A. Jain, B. Kucera, and A. Mametjanov. Comprehension of generative techniques. In *Software Transformation Systems Workshop (STS)*, 2006.