The Essence of Strategic Programming

— Draft as of October 8, 2003 —

Ralf Lämmel[1] and Eelco Visser[2] and Joost Visser[3]

[1] Free University, Amsterdam, The Netherlands, ralf@cs.vu.nl
[2] University Utrecht, The Netherlands, visser@cs.uu.nl
[3] SIG, Amsterdam, The Netherlands, Joost.Visser@software-improvers.nl

Abstract. Strategic programming is generic programming with the use of *strategies*. A strategy is a generic data-processing action which can *traverse* into heterogeneous data structures while mixing uniform and *type-specific* behaviour. With strategic programming, one gains full control over the application of basic actions, most notably full *traversal control*. Using a combinator style, traversal schemes can be defined, and actual traversals are obtained by passing the problem-specific ingredients as parameters to suitable schemes. The prime application domain for strategic programming is *program transformation and analysis*. In this paper, we provide a *language-independent definition* that generalises over existing incarnations of this idiom in term rewriting, functional programming, and object-oriented programming.

Keywords: Generic programming, Strategic programming, Traversal, Strategy, Language design, Program transformation, Program analysis

1 In need of traversal control

In various areas of programming, control over the application of basic data-processing actions is needed. *Data* is meant here in the sense of heterogeneously typed data such as terms, documents, ASTs, and object structures. *Control* can be classified as follows:

i the *order* of applying the basic actions,
ii the *side conditions* enabling the basic actions,
iii the propagation of *effects* caused by the actions, and
iv the *traversal* over the compound input data.

Such control is crucial, for example, in program transformation in order to guarantee correctness and termination of many transformations. Control in the sense of (i) ordering, (ii) side conditions, and (iii) effects is reasonably understood — just think of structured programming, design by contract, and monadic style programming. A challenge arises from (iv) *traversal control*, and from its interaction with (i)–(iii).

The established programming paradigms provide little support to effectively separate traversal control and basic actions. We contend that there is not just a lack of *expressiveness* but also a lack of a general *design method* for traversal functionality. It is common practice to implement traversals according to one of the following options:

The following function descends into terms representing regular expressions to perform simplifications according to the first two equations. The remainder of the program encodes descent.

```
simplify :: RegExp → RegExp
simplify (Alt Epsilon exp)   = Opt (simplify exp)      -- introduce '?'
simplify (Opt (Plus exp))    = Star (simplify exp)     -- introduce '*'
  -- recurse into compound regular expressions
simplify (Star exp)          = Star (simplify exp)
simplify (Plus exp)          = Plus (simplify exp)
simplify (Opt exp)           = Opt (simplify exp)
simplify (Seq exp1 exp2)     = Seq (simplify exp1) (simplify exp2)
simplify (Alt exp1 exp2)     = Alt (simplify exp1) (simplify exp2)
simplify exp                 = exp
```

Fig. 1. Tangling of traversal control and basic computation in a Haskell program

a rely on a built-in normalisation strategy, or
b entangle traversal and basic computations, or
c compromise the primary, typed representation of data.

Option (a) is common practice in rule-based programming (e.g., algebraic term rewriting, expert systems, abstract state machines). Here, the programmer supplies a set of computation rules and relies on the system's *built-in evaluation strategy* (e.g., leftmost-innermost) for applying these to the relevant parts of the input data. This approach breaks down when extra precautions are required to enforce termination, restriction to a scope, or side conditions involving extra data. Then, programmers end up adhering to option (b), that is, they encode traversal functionality in computation rules.

Option (b) is common practice in standard functional programming. Here, the programmer encodes traversals as families of mutually recursive functions via pattern-match equations. In this approach, the *traversal strategy is heavily entangled with the basic actions* as illustrated in Fig. 1. Virtually, the same could be said about the basic visitor technique in standard object-oriented programming. The problem with this approach is that the size of the entangled traversal code is proportional to the number of constructors regardless of the specific problem. Also, the tangling has to be repeated for every new piece of traversal functionality. As an instructive example, a transformation system for GOTO elimination in Cobol consists of a few dozens components traversing over the Cobol syntax with around 300 constructors. Clearly, option (b) does not scale here.

Option (c) suggests to operate on an untyped representation type, maybe even in a basically untyped environment such as Prolog. This approach suffers from a lack of the very valuable *static checks* that are available for heterogeneously typed data in most programming languages. A sophistication of this option is to restrict the employment of the untyped representation to just generic traversal functionality in an otherwise typed environment. Normally, this implies that data frequently needs to be converted to and from the representation type. Also, generic functionality is still untyped.

In this paper, we systematically explain the generic programming idiom of *strategic programming*. This idiom dramatically improves on options (a)–(c). With strategic

programming, one obtains complete *traversal control* over the application of basic actions (as opposed (a)), without tangling basic actions and traversal control (as in the case of (b)), and without resorting to an artificial representation type (as in the case of (c)). In fact, control patterns for traversal over heterogeneous data structures, say *traversal schemes*, become programmer-definable abstractions. As a result, both traversal schemes and problem-specific ingredients can be reused across modules.

In previous work, we incarnated strategic programming in different paradigms, namely in term rewriting [34,45,25], functional programming [30,10,23,29], and object-oriented programming [44]. Knowledge of this material is not assumed. In the present paper, we generalise over these incarnations. In Sec. 2, we will explain the strategic method and the strategic expressiveness that is needed. In Sec. 3, we will characterise strategies — the principle notion underlying strategic programming. In Sec. 4, we will provide a set of basic combinators for constructing strategies. In Sec. 5, we will sketch a strategic programming library capturing common traversal schemes. In Sec. 6, we will review existing incarnations of strategic programming in light of our generalised characterisation. In Sec. 7, related work is discussed. In Sec. 8, an assessment of strategic programming is given including pointers to strategic programming environments and applications. In Sec. 9, the paper is concluded.

2 Strategic traversal = basic actions + traversal control

The key idea underlying strategic programming is the separation of problem-specific ingredients of traversal functionality (i.e., basic actions) and reusable traversal schemes (i.e., traversal control). In order to enable the definition of traversal schemes, an incarnation of strategic programming provides primitives for 'one-layer' traversal that cater for *generic access* to the *immediate* components of heterogeneous data structures.

To this end, we assume that compound data (say, terms, objects, XML elements, and others) can be decomposed into immediate components. One-layer combinator only operate on these immediate components. Typical one-layer traversal combinators are the following:[1]

all Apply an argument strategy to all immediate components while preserving the overall shape of the datum.

one Apply an argument strategy to one 'fit' component while preserving the overall shape of the datum. Fitness is derived from the success/failure behaviour of the argument strategy.

reduce Similar to *all* but the intermediate results of processing the immediate components are reduced by a binary operation.

select Similar to *one* but the successfully processed immediate component is returned as the result.

By *not* anticipating any scheme of recursion, one-layer traversal can still be completed into 'deep' traversal in different ways by plain recursion. Two specific one-layer combinators and derived recursive completions are illustrated in Fig. 2.

[1] The names for these combinators differ in the SP literature. Here, we use the terms favoured in [25].

Below we illustrate two one-layer traversal combinators: **all** to process all immediate components, and **one** to process the leftmost one for which the argument strategy succeeds. (Shaded vs. black nodes represent failure vs. success of processing.)

Below we illustrate two recursive completions of the one-layer combinators. The upper one completes **all** into a full top-down traversal (for short, $full\_td$). The lower one completes **one** into a single-hit bottom-up traversal (for short, $once\_bu$).

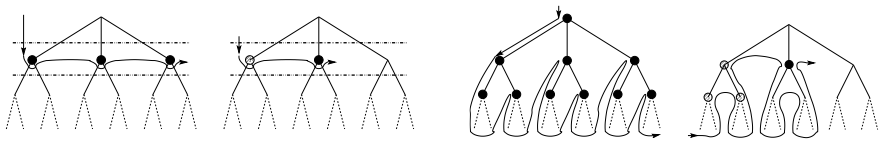

Fig. 2. Strategic traversal with one-layer combinators

Expressiveness complemented by a method The strategic programming idiom encompasses both *expressiveness* and *a method* for designing and implementing traversal functionality. The 'strategic' expressiveness is basically that strategies are first-class citizens, and that recursive traversal schemes can be composed in all kinds of ways from primitive one-layer traversal combinators. This expressiveness is sometimes hard, sometimes easy to achieve — depending on the targeted programming paradigm, the required strength of typing and convenience for the programmer. The 'strategic' method can be summarised in the following steps for the implementation of a piece of strategic traversal functionality:

1. identification of a reusable traversal scheme,
2. definition of the problem-specific ingredients, and
3. synthesis of the traversal by parameter passing.

Normally, the reusable traversal schemes is completely generic where all problem-specific actions are anticipated via parameters. The problem-specific ingredients of a traversal are type-specific actions or generic actions with type-specific branches. These actions are meant to describe how data of 'interesting' types is processed when encountered in the course of the traversal. The strategic method, although general, is in no way difficult. This method of decomposition has been simply overlooked elsewhere.

Variation points in traversal The strategic method enables and even encourages the programmer to reflect on the variation points in traversal control for a piece of traversal functionality in a given context. The separation of basic actions and traversal control is at the very heart of strategic programming, and it makes it easy to alter the design of a traversal. These are some *variation points* for traversal control:

- transformation vs. query,
- top-down vs. bottom-up traversal,
- depth-first vs. breadth-first traversal,
- left-to-right traversal and vice versa,
- single vs. nested or cascaded traversal,

– local choice vs. cut vs. full backtracking,
– optimised variations on traversal schemes,
– full vs. one-hit vs. cut-off vs. path traversal,
– fixpoint by failure vs. fixpoint by equality test,
– effectfull traversal (accumulation, binding, etc.),
– type-specific vs. generic problem-specific ingredients.

We have experienced these variation points in actual applications of strategic programming such as those reported in [45,41,16,1,18,30,9,46,6,24]. In addition to general purpose traversal schemes, there is ample opportunity for domain-specific schemes, e.g., name analyses for language processing [41,24]. There is no other generic programming idiom which would have addressed these variation points. This includes adaptive programming which is mainly about paths through object graphs.

A strategic example We refactor the tangled functionality from Fig, 1 to adhere to the strategic style. In a first phase, we separate out the basic 'rewrite steps' for simplification. We use the term 'rewrite step' to suggest that this functionality takes the form of rewrite rules in the sense of term rewriting. In Haskell, the extracted simplification rules are represented as follows:

$$
\begin{array}{lcl}
simplifyStep :: RegExp \rightarrow Maybe\ RegExp & & \\
simplifyStep\ (Alt\ Epsilon\ exp) & = & Just\ (Opt\ exp) \\
simplifyStep\ (Opt\ (Plus\ exp)) & = & Just\ (Star\ exp) \\
simplifyStep\ \_ & = & Nothing
\end{array}
$$

Here, we use the $Maybe$ type constructor in order to express whether any simplification rule triggers or not. Now, we need a traversal scheme that iterates the simplification rules all over the tree. In this simple example, we attempt two beginner's favourites: $full\_td$ (read as full top-down) and $innermost$. Then, the tangled Haskell function $simplify$ from Fig. 1 is reconstructed in strategic style by passing $simplifyStep$ to either of these:

$$
\begin{array}{lcl}
simplify & = & full\_td\ (choice\ simplifyStep\ id) \\
simplify' & = & innermost\ simplifyStep
\end{array}
$$

The first definition assumes a 'full' traversal, a data processor is required that always succeeds. Hence, we adapt the rewrite step $simplifyStep$ in $simplify$ to recover from the potential failure of $simplifyStep$ (cf. $choice$). Although this first definition is perfectly modular, and a faithful reconstruction of the original tangled code, a shortcoming becomes obvious. The definition does not enforce the exhaustive application of $simplifyStep$. This is because $full\_td$ applies its argument to the input datum before its immediate components were traversed. The variation $simplify'$ eliminates this problem because it performs simplifications according to the $innermost$ traversal scheme. This scheme operates bottom-up, and it loops until a fixpoint is reached. Hence, in this example, $innermost$ is more appropriate than $full\_td$. In another context, the opposite situation is possible, for example, if the use of $innermost$ would cause a nonterminating strategy due to the nature of the given rewrite step.

3 An abstract notion of strategy

Strategic programming is programming with the use of strategies. Depending on the incarnation of strategic programming within a certain programming paradigm, strategies might correspond to pure functions, impure functions, objects, and others. Further strategies might be statically typed, dynamically typed and others. We would first like to provide an *abstract* notion of strategy that is not bound to any particular programming language or paradigm, nor do we want to include unnecessary requirements.

Definition Strategies are data-processing actions with the following characteristics:

Genericity Strategies are generic in the sense that they are applicable to data of any type (say, sort, or class).

Specificity Though generic, strategies provide access to the actual data structures by means of type-specific operations.

Composability There are means to express compound, conditional, and iterated strategy application.

Traversal Strategies enable generic traversal into the immediate components of heterogeneous data structures.

Partiality The application of a strategy to a given datum may fail, and recovery from failure is feasible.

First-class Strategies are first-class citizens in the sense that they can be named, can be passed as arguments, etc.

We contend that the synergy of strategic programming is gone if any of these characteristics is not present. The abstract notion of strategy serves two purposes. Firstly, it corresponds to a requirement specification for incarnating strategic programming in a given programming language or paradigm. Secondly, it is a useful reference chart to assess other generic programming approaches. As for adaptive programming, its conception of traversal is not centred around one-layer traversal on immediate components but around paths in object structures. Normally, adaptive traversal strategies are not first-class citizens, although recent implementations [37] seem to admit this potential.

We should note that the term *strategy* or related terms like *tactics* and *tacticals* are also used in contexts other than generic programming. Usually some sort of 'control' is associated to this use, but not the means to cater for generic access to components of heterogeneous data structures. Rewriting strategies (represented as higher-order functions) are used in the LCF theorem prover to describe proof tactics and tacticals [39]. Programmable evaluation strategies are used in term rewriting [8,2]. Strategies in the sense of [40] target the synthesis of parallel functional programs. Our use of the term strategy starts with the work on traversal strategies à la Stratego [34,45]. Obviously, calculi tend to overlap to some extent because of omnipresent concepts such as sequential composition and choice.

4 Basic strategy combinators

In strategic programming, we require a combinator style which supports the first-class status of strategies. We can make the strategy notion significantly more concrete by

prescribing a set of combinators that must be supported by an incarnation of strategic programming. Fig. 3 specifies such a set. We suggest a semi-formal reading of Fig. 3. In particular, the given semantic sketch leaves open how to blend with the expressiveness offered by the host paradigm of an eventual incarnation. In a term-rewriting setting, the semantics of strategies was formally defined in [34,45,25], and special attention to static typing is paid in [25]. In fact. term rewriting provides the reference model for strategic programming. Actual incarnations of strategic programming may include further combinators than those from the figure. Also, the identified combinators are not necessarily primitives in a specific incarnation but they might be defined in terms of other expressiveness. We will now explain all the combinators.

Standard combinators The nullary strategy id succeeds for any datum and returns its input without change. The strategy fail fails for any datum, indicated by the output $\uparrow$. The sequence combinator seq applies its two argument strategies in succession. The left-biased choice combinator first attempts application of its first argument strategy. If and only if this application fails, the second argument is attempted. There is no recursion or iteration combinator. Instead, we assume that the definition of new named combinators can involve recursion.

One-layer traversal The definitions of the combinators all and one formalise the intuitions from Fig. 2. They both push their argument strategy one level down into the input datum to process all immediate components, or just the leftmost one for which the argument strategy succeeds. We use dedicated notation to differentiate between indivisible data and compound data. Note that all and one preserve the shape of the input datum because the constructor c reappears in the result. We also say that this kind of strategies is type-preserving, or that they perform a transformation. There exist similar combinators for the type-unifying type scheme [25,30], that is, for combinators that perform a query or an analysis with a fixed result type regardless of the input datum's type. To illustrate the definition of recursive traversal schemes in terms of one-layer combinators, we define $full\_td$ for full top-down traversal in terms of all. The following recursive definition means that $full\_td(s)$ applies its argument strategy s at the root of the incoming datum, and then (cf. seq) it applies itself to all immediate components of the datum:

$$full\_td(s) \;=\; \mathsf{seq}(s, \mathsf{all}(full\_td(s)))$$

Lifting type-specific actions In Fig. 3, we distinguish *type-specific* actions vs. *generic* actions — the latter being called strategies. There are means to mediate between the two categories. Obviously, a generic action s can be applied to a datum d of any type without much ado. Application $\ldots @ d$ is overloaded for type-specific and generic actions accordingly. Notably, a type-specific action can also be applied in a generic context either by explicit 'lifting' via the adhoc combinator or by implicit lifting. Such lifting is needed because the most basic ingredients of a strategic program are *type-specific* actions but they eventually need to be applied to components of different sorts in the course of traversal. Using the adhoc combinator for *type-based dispatch*, one can construct a new strategy from a from a generic default s and a type-specific action a. The strategy $\mathsf{adhoc}(s, a)$ behaves like s except for data a's input type where it dispatches to

Combinators

$s ::= \mathsf{id}$		Identity strategy
$\mid \mathsf{fail}$		Failure strategy
$\mid \mathsf{seq}(s, s)$		Sequential composition
$\mid \mathsf{choice}(s, s)$		Left-biased choice
$\mid \mathsf{all}(s)$		All immediate components
$\mid \mathsf{one}(s)$		One immediate component
$\mid \mathsf{adhoc}(s, a)$		Type-based dispatch

Notation

d	...	data
c	...	data constructors
$\overline{d}$	...	data with failure "↑"
a	...	type-specific actions
s	...	strategies
$a@d$	...	application of a to d
$s@d$	...	application of s to d
$d \Rightarrow \overline{d}$	...	big-step semantics
$a : t$	...	type handled by a
$d : t$	...	type of a datum d
$[d]$	...	indivisible data
$c(d_1 \cdots d_n)$	...	compound data

Meaning

$$
\begin{array}{ll}
\mathsf{id}@d & \Rightarrow d \\
\mathsf{fail}@d & \Rightarrow \uparrow \\
\mathsf{seq}(s, s')@d & \Rightarrow \overline{d} \quad \text{if } s@d \Rightarrow d' \wedge s'@d' \Rightarrow \overline{d} \\
\mathsf{seq}(s, s')@d & \Rightarrow \uparrow \quad \text{if } s@d \Rightarrow \uparrow \\
\mathsf{choice}(s_1, s_2)@d & \Rightarrow d' \quad \text{if } s_1@d \Rightarrow d' \\
\mathsf{choice}(s_1, s_2)@d & \Rightarrow \overline{d} \quad \text{if } s_1@d \Rightarrow \uparrow \wedge s_2@d \Rightarrow \overline{d} \\
\mathsf{all}(s)@[d] & \Rightarrow [d] \\
\mathsf{all}(s)@c(d_1 \cdots d_n) & \Rightarrow c(d'_1 \cdots d'_n) \quad \text{if } s@d_1 \Rightarrow d'_1, \ldots, s@d_1 \Rightarrow d'_n \\
\mathsf{all}(s)@c(d_1 \cdots d_n) & \Rightarrow \uparrow \quad \text{if } \exists i.\ s@d_i \Rightarrow \uparrow \\
\mathsf{one}(s)@[d] & \Rightarrow \uparrow \\
\mathsf{one}(s)@c(d_1 \cdots d_n) & \Rightarrow c(\cdots d'_i \cdots) \quad \text{if } \exists i.\ s@d_1 \Rightarrow \uparrow \wedge \cdots \wedge s@d_{i-1} \Rightarrow \uparrow \wedge s@d_i \Rightarrow d'_i \\
\mathsf{one}(s)@c(d_1 \cdots d_n) & \Rightarrow \uparrow \quad \text{if } s@d_1 \Rightarrow \uparrow, \ldots, s@d_n \Rightarrow \uparrow \\
\mathsf{adhoc}(s, a)@d & \Rightarrow a@d \quad \text{if } a : t \text{ and } d : t \\
\mathsf{adhoc}(s, a)@d & \Rightarrow s@d \quad \text{if } a : t \wedge d : t' \wedge t \neq t'
\end{array}
$$

Identities

$$
\begin{array}{lll}
\text{[unit]} & s & \equiv \mathsf{seq}(\mathsf{id}, s) \equiv \mathsf{seq}(s, \mathsf{id}) \equiv \mathsf{choice}(\mathsf{fail}, s) \equiv \mathsf{choice}(s, \mathsf{fail}) \\
\text{[zero]} & \mathsf{fail} & \equiv \mathsf{seq}(\mathsf{fail}, s) \equiv \mathsf{seq}(s, \mathsf{fail}) \equiv \mathsf{one}(\mathsf{fail}) \\
\text{[skip]} & \mathsf{id} & \equiv \mathsf{choice}(\mathsf{id}, s) \equiv \mathsf{all}(\mathsf{id})
\end{array}
$$

Nested type dispatch

$$
\begin{array}{l}
\mathsf{adhoc}(\mathsf{adhoc}(s, a), a') \equiv \mathsf{adhoc}(s, a') \text{ if } a : t \wedge a' : t \\
\mathsf{adhoc}(\mathsf{adhoc}(s, a), a') \equiv \mathsf{adhoc}(\mathsf{adhoc}(s, a'), a) \text{ if } a : t \wedge a' : t' \wedge t \neq t' \\
\mathsf{adhoc}(\mathsf{adhoc}(\mathsf{fail}, a), a') \equiv \mathsf{choice}(\mathsf{adhoc}(\mathsf{fail}, a), \mathsf{adhoc}(\mathsf{fail}, a')) \text{ if } a : t \wedge a' : t' \wedge t \neq t'
\end{array}
$$

Fig. 3. Specification of a guideline set of basic strategy combinators

a. Not every incarnation of strategic programming needs to perform explicit lifting via adhoc. Instead, one can favour implicit lifting which can be thought of as $\mathsf{adhoc}(\mathsf{fail}, a)$. Implicit lifting is problematic regarding static checking [25]. We illustrate explicit lifting by adding applications of adhoc to the strategic Haskell snippet from Sec. 2:

$$
\begin{array}{lcl}
\mathit{simplify} & = & \mathit{full\_td}\ (\mathsf{choice}\ (\mathsf{adhoc}\ \mathsf{fail}\ \mathit{simplifyStep})\ \mathsf{id}) \\
\mathit{simplify'} & = & \mathit{innermost}\ (\mathsf{adhoc}\ \mathsf{fail}\ \mathit{simplifyStep})
\end{array}
$$

In both variations, we use fail as generic default. In the first case, id is feasible, too. Generic defaults other than id and fail are also sensible. One could, for example, consider recursive descent as generic default which is only meant to happen if the type of the basic action and the type of the given datum do not fit.

5 A strategic programming library

The power of our small set of basic combinators can best be demonstrated with a few examples. Fig. 4 shows a list of combinators defined in terms of the basic ones. The first two control patterns *try* and *repeat* do not involve traversal control whereas the remaining combinators define different traversal schemes.

$$
\begin{aligned}
try(s) &= \mathsf{choice}(s, \mathsf{id}) \\
repeat(s) &= try(\mathsf{seq}(s, repeat(s))) \\
full\_td(s) &= \mathsf{seq}(s, \mathsf{all}(full\_td(s))) \\
full\_bu(s) &= \mathsf{seq}(\mathsf{all}(full\_bu(s)), s) \\
once\_td(s) &= \mathsf{choice}(s, \mathsf{one}(once\_td(s))) \\
once\_bu(s) &= \mathsf{choice}(\mathsf{one}(once\_bu(s)), s) \\
stop\_td(s) &= \mathsf{choice}(s, \mathsf{all}(stop\_td(s))) \\
path\_td(s) &= \mathsf{seq}(s, \mathsf{choice}(\mathsf{all}(\mathsf{fail}), \mathsf{one}(path\_td(s)))) \\
naive\_innermost(s) &= repeat(once\_bu(s)) \\
innermost(s) &= \mathsf{seq}(\mathsf{all}(innermost(s)), try(\mathsf{seq}(s, innermost(s)))) \\
beloweq(s_1, s_2) &= once\_td(\mathsf{seq}(s_2, once\_td(s_1))) \\
untileq(s_1, s_2) &= \mathsf{seq}(s_1, \mathsf{choice}(s_2, \mathsf{one}(untileq(s_1, s_2))))
\end{aligned}
$$

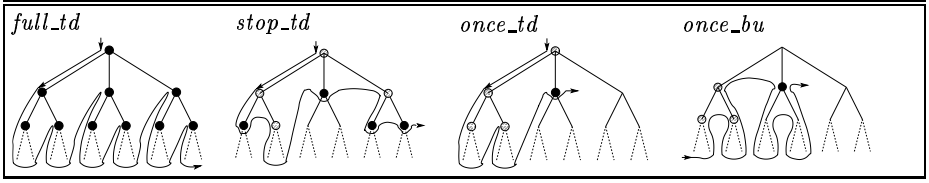

Fig. 4. Some defined strategy combinators

Non-traversal control The combinator *try* manipulates the success value of its argument strategy: $try(s)$ recovers from failure of s via id if necessary. This control pattern is useful whenever it must be enforced that a given action s succeeds. The incoming datum is returned when s normally would fail. The *repeat* combinator serves for fixpoint computation: $repeat(s)$ applies its argument strategy s repeatedly until s fails. This control pattern is useful in the definition of traversal schemes whenever traversal involves exhaustive application of actions.

Traversal schemes The combinators *full_td* and *full_bu* model a full top-down or bottom-up traversal, respectively. They apply their argument strategy at the root of the incoming datum, *and* at *all* its immediate and non-immediate components. The combinators *once_td* and *once_bu* are variations that apply the argument strategy only to the first component at which it succeeds. The combinator *stop_td* attempts the application of the argument strategy to components along all branches, and it stops in a given branch when an application succeeds. The combinator *path_td* searches for a complete path through the compound datum such that the argument strategy succeeds at all levels. The *naive_innermost* and *innermost* combinators both implement the leftmost innermost evaluation strategy, but the second is more efficient than the first. The combinators *beloweq* and *untileq* take two argument strategies for some form of path traversal very much in the sense of adaptive programming [32]. The combinator *beloweq* searches for a component that can be successfully processed by the first argument nested inside a component for which the second argument succeeds. The combinator *untileq* searches for a path such that the first argument holds for every node down the path until eventually the second argument holds. For generality, in both cases, it is not yet ruled out that the two argument strategies succeed at the same node.

6 Incarnations of strategic programming

The strategic programming idiom is not just a theoretical artifact. It has actually been realized within several programming paradigms and applied in the construction of numerous tools. The best worked-out incarnations are in term rewriting based on the Stratego language [45,18], in functional programming based on Haskell [30,23,29], and in object-oriented programming based on Java [44]. Furthermore, strategic programming has been incarnated for logic programming based on Prolog (in the context of the Language Processing Toolbox [27]), and for imperative programming based on C (in the context of the Grammar Deployment Kit [21]). We will not work through all these incarnations. Instead, we want to specify how to incarnate, in general, strategic programming. Also, we want to compare the incarnations at a higher level of abstraction.

Obligations We will start by listing the overall obligations for the incarnation process:

Inventory Before embarking on an integration of strategies into the relevant paradigm (or language), one has to analyse existing approaches to separate basic actions and traversal control in this paradigm. One can then assess the contribution that strategic programming is able to make to the paradigm. Also, one might identify available language-specific expressiveness to implement strategies.

Semantics The abstract notion of strategy must be mapped onto the host paradigm. This necessitates the identification of a corresponding abstraction form for strategies. Also, one has to instantiate subsidiary notions such as *type-specific action*, *datum*, *component*, *partiality* (cf. Fig. 6). The mapping culminates in the implementation of the guideline set of basic strategy combinators (recall Fig. 3).

Hosting All programming idioms that are ‘native’ to the host paradigm should remain available to the programmer when using strategies. For instance, in object-oriented programming, strategies should blend with reference semantics, and side-effects.

Functional strategies should have value semantics, allow monadic effects, and be strict or lazy depending on the host language.

Typing If a strongly typed setting is favoured, 'strategic polymorphism' [29,25,23] poses a challenge. It necessitates second-order polymorphism, and it goes beyond parametric polymorphism and ad-hoc polymorphism. With liberal typing as in Stratego, terms are formed according to a many-sorted signature, but strategy combinators are only subject to weaker checks, e.g., arity checking.

A sample incarnation We demonstrate the incarnation process for Prolog. This is a rather arbitrary example which however allows for a compact discussion. Our overall intention is to exemplify a number of issues that naturally show up during incarnation. We start with the inventory. Prolog is heavily used for problems that involve traversal over nontrivial term formats, e.g., in language prototyping. However, traversal functionality is typically implemented in tangled fashion. Higher-order predicates for list processing are folklore but not so for traversal schemes. The separation of logic and control has been advocated in Kowalski's seminal *"Algorithm = Logic + Control"* [22], but this was never achieved for traversal control. Hence, we are motivated to incarnate strategies in Prolog. In fact, mapping the strategy semantics onto the Prolog language is straightforward. As an indication, in Fig. 5, we implement the `all` combinator, and everything else what is needed for the strategic reconstruction of the *simplify* function from Fig. 1. We model both type-specific actions and strategies as Prolog predicates. As for the latter, we make use of higher-order predicates. It is quite natural not to define the combinators `seq` and `choice` but rather to directly use Prolog's conjunction and disjunction. So the data-flow is wired up just with shared variables in the literals, and "," is used to model `seq`. Multiple clauses for a predicate with 'cut' (cf. "!" in `try/3`) faithfully encode left-biased choice. We assume an implicit lifting approach, that is, the `adhoc` combinators is omitted. In addition to the higher-order facilities of Prolog, the present incarnation crucially relies on the univ operator "`=..`" for term destruction and construction (cf. `map/3`). The univ operator allows us to implement one-layer traversal without much ado based on heterogeneous list processing. On the other hand, this approach precludes typing of predicates for one-layer traversal. However, functionality that merely uses one-layer traversal combinators can be typed [25,27].

Comparison In Fig. 6, we compare strategic programming in several paradigms based on the instantiation of the relevant concepts. The incarnations exhibit different tradeoffs which would be excessive to discuss, though we make a few observations below. To this end, we first focus on Stratego [45] as a language devoted to the strategic programming idiom, and then we discuss incarnations that qualify strategic programming as general-purpose generic programming.

Stratego—a DSL for program transformation In the design of Stratego, the prime issue was to effectively support the development of program transformation systems. Hence, Stratego can be viewed as a domain-specific language (DSL). Domain-specific constructs were integrated, namely hygienic symbol generation, scoped dynamic rewrite rules [42], and a foreign language interface. The DSL character is also reflected by specific optimisations [17], and the use of a designated run-time term representation that

```
% Apply a goal to immediate subterms
all(G,X,Y) :- X=..[F|Xs], map(G,Xs,Ys), Y=..[F|Ys].

% Map a goal over a list
map(_,[],[]).
map(G,[X|Xs],[Y|Ys]) :- call(G,X,Y), map(G,Xs,Ys).

% Recover from failure of a goal
try(G,X,Y) :- call(G,X,Y),!.
try(G,X,X).

% Perform a full top-down traversal
full_td(G,X,Z) :- call(G,X,Y), all(full_td(G),Y,Z).

% Encode rewrite step for simplifications
simplifyStep(alt(epsilon,Exp),opt(Exp)).
simplifyStep(opt(plus(Exp)),star(Exp)).

% Perform simplifications in a traversal
simplify(X,Y) :- full_td(try(simplifyStep),X,Y)
```

Fig. 5. Strategic programming in Prolog

allows for sharing, constant time equality test, and hidden transportation of comments and layout.

Strategies in general-purpose languages The incarnations for functional and object-oriented programming make these paradigms fit for strategic programming based on Haskell and Java. This is an economic approach because of the only few 'strategic' constructs. The respective principal form of first-class citizens is used to model strategies: functions in Haskell, and objects in Java, while varying models are possible. Strategic programming over the programmer-supplied datatypes or classes is most easily enabled via generative tool support. The flagship Haskell model in [29] is strongly typed where 'strategic polymorphism' is based on a combination of second-order polymorphism and type classes. This model also clearly suggests a language extension for strategic programming because all the original expressiveness is captured in two simple and well-defined combinators. The Java model allows for traversal over object structures (i.e., graphs) as opposed to terms in the other incarnations. This is convenient in the context of certain program analyses that operate on intermediate formats such as control-flow graphs. The Java model also allows for mutable objects in traversal. The typing model necessitates some casting owing to limitations of the Java language. This problem will be remedied with generics in Java.

7 Non-strategic programming

The strategic programming idiom has emerged from numerous approaches that contain some of its ingredients. Various concepts enable, to some extent, separation of traversal

Concept per paradigm	Term rewriting	Functional programming	OO programming	Logic programming	Imperative programming
Datum	many-sorted term	term of an algebraic datatype	object structure	basically untyped term	data structure behind term API
Immediate component	subterm	subterm	referenced object	subterm	subterm in the API sense
Basic action	rewrite rule	type-specific function	class-specific visit method	first-order predicate	function on specific term type
Strategy	term rewriting strategy	'strategically' polymorphic function	generic visitor object	term representing a goal	term of a specific strategy type
Strategy application	dedicated operator	function application	visit method invocation	meta-predicate call	C-function for strategy interpretation
Strategy combinators	strategy definitions	higher-order functions	visitor classes	meta-predicates	ordinary C functions
Type-based dispatch	implicit	type-safe cast	RTTI and dynamic binding	implicit	C-like cast
Types	liberal checks against signature	rank-2 types with constrained "∀"	subtype polymorphism	initially untyped	liberal checks against term API
Partiality	built-in	monadic effect	exceptions	backtracking	via designated result NULL
Host idioms	'DSL-like' extensions	monadic effects	graphs, side effects	meta-logical features	side effects

Fig. 6. Overview of incarnations per paradigm

control and basic actions. In our previous work, we have analysed related work in the more technical context of certain programming paradigms [34,45,44,43,25,30,23,29]. In the present paper, we will restrict ourselves to related approaches that are of special interest in light of our language-independent definition of strategic programming.

Levels of traversal control Our discussion of related work will be centred around traversal control. To this end, we propose the following levels to measure the sophistication of traversal control in different styles of generic programming:

Level 1 entangled traversal,
Level 2 fixed traversal schemes,
Level 3 composable traversal strategies,
Level 4 first-class strategies with one-layer traversal.

We only list Level 1 here as the borderline level which hosts non-generic programming. We also do not reserve a level for 'implicit traversal' in the sense of a built-in normalisation strategy such as innermost. Recall that such an approach usually degrades to

Level 1, that is, entangling traversal and basic computations — as soon as some amount of traversal control is needed. We will place related approaches at levels 2–3. In particular, adaptive programming ends up at Level 3. There are no Level 4 relatives of the strategic programming idiom.

Level 2 Generalised folds [35] in functional programming provide an orthogonal example of a fixed traversal scheme. Note that this approach does not immediately enable generic term traversal because one has to define the basic computation for traversal as a fold algebra the type of which is however specific to the data structure at hand. This problem has been solved in [31] based on the idea of updatable primitive fold algebras. This provides adaptiveness for programming with traversal schemes. Similar ideas have been integrated with term rewriting; cf. traversal functions in [4,3]. Also, visitors [12] in object-oriented programming can be programmed in a way that some default traversal behaviour is customised by inheritance. To summarise, using fixed traversal schemes one can define traversals by customisation, function composition, and nesting. In strategic programming, traversal schemes are programmable.

Level 3 At this level, we require capabilities to *compose* generic traversals in a more general way. The traversal strategies in adaptive programming [32,37] correspond to descriptions of paths through object graphs that are composed from predicates for anchor nodes, end nodes, intermediate nodes, and disfavoured nodes and edges. There is no notion of a completely generic traversal scheme. Traversal control is limited by the bias towards paths and due to the absence of one-layer traversal primitives. Polytypism [15] in functional programming supports traversal over data of any type based on induction over the type structure. Polytypic functions gain generic access to the components of compound terms by viewing term types as sums of products. Polytypic functions can be customised by type-specific cases. Until recently these refinements had to be made *in place* with the polytypic cases. In [7], this problem was resolved by allowing for the postponed customisation of polytypic functions. However, even in this approach, polytypic functions are not first-class citizens.

We should also relate strategic programming to general aspect-oriented programming (AOP [20,11]), even if this is largely unrelated to the levels of traversal control. In the introduction of this paper, we quoted from [33] to state that adaptive programming is an instance of aspect-oriented programming. Strategic programming qualifies as an instance of aspect-oriented programming for very similar reasons. In general, AOP aims at separation of concerns by allowing features that *cross-cut* a given system's primary modularisation to be factored out. In SP, the primary modularisation would be the data structure (given as a signature, a class graph, XML schema), and the concern which is no longer entangled with these data structures is traversal behaviour. When we compare the strategic setup with general purpose aspect-oriented languages such as AspectJ [19], the following differences are worth mentioning. The aspects in such a language employ name-based or property-based *pointcut designators* to pinpoint where in the execution flow the aspect's code should be activated. These aspects are defined and applied to base functionality that is unaware of the aspects. By contrast, strategy combinators are parameterised building blocks, whose functionality is only activated through composition.

Prominent models for XML document processing live at Level 3 as well. In XSLT [47], for example, one can explicitly regulate how to descend into the components within each template rule, e.g.:

```
<xsl:apply-templates select=
  "*|@*|comment()|processing-instruction()|text()"/>
```

Here, we just recurse into *all* children without indicating any special mode. In general, one can select children, and, one can switch to different modes. However, templates à la XSLT are not first-class citizens, and there is fixed model that regulates how a collection of templates is used to organise the traversal over a document.

8 Assessment of strategic programming

We discuss the benefits of strategic programming complemented by the characterisation of strategic programming environments, and a brief list of typical applications.

Benefits of strategic programming The strategic method enables concise traversal code, the separation of basic actions and traversal control, the reuse of basic actions and traversal schemes, and resistance to change. These benefits deserve a closer look:

Conciseness When programming on heterogeneous data structures, strategic programming can be radically more concise than non-generic programming. Due to the generic access to specific data structures, the strategic programmer need not repeatedly implement, for each type, the behaviour that is conceptually generic.

Separation of concerns Strategy *combinators* allow one to implement conceptually separable concerns in distinct abstractions which otherwise would be entangled in a single code fragment. Strategies support separation of basic actions, traversal control, ordering, applicability conditions, and effects. These concerns can be understood, reasoned about, developed, tested, and maintained separately.

Reuse Strategies enable reuse at two levels. Within an application, a single concern, such as a problem-specific basic action, needs to be implemented only once. Across applications, strategies can be reused that capture generic behaviour in the sense of general-purpose traversal schemes or domain-specific generic algorithms.

Adaptiveness Using strategies, each concern can be implemented with explicit reference only to types and data constructors that are relevant to it. As a result, changes in the data structures (say, signatures, algebraic data type definitions, class hierarchies) will not unduly affect the implementation of the concern. This isolation from changes diminishes the resistance to change of software systems, and makes them more maintainable.

These benefits, together with our simple language-independent characterisation, and the firm foundations of strategic programming suggest that our notion of strategy forms yet another piece of reusable language-design knowledge.

Another note on adaptive programming is in place. What we call adaptiveness above is presumably called structure shyness in AP. It turns out that this shyness is sometimes considered harmful in AP [38]. The problem is that traversal strategies might go

along surprising paths especially when the object model changes. One can call this a robustness issue. To complement Demeter's declarative strategies, a domain-specific language (DSL) has been proposed to express recursive traversals at a lower, more explicit level [38]. This traversal DSL sacrifices some compactness and adaptiveness in order to gain more control over propagation and computation of results, and to prevent unexpected traversal paths due to under-specification of traversals. This under-specification problem is differently perceived in the SP context. Since our application context is programming transformation and analysis, we mainly expect genericity in the sense of conciseness from strategic programming — adaptiveness in its AP sense to a reduced extent. A strategic programmer is usually very well aware of the language syntax (s)he is dealing with — so few surprises are to be expected. Also, languages do not change so dramatically as ordinary object models. This does not per-se rule out unexpected hits in a traversal but the situation is different when compared with rather arbitrary object models.

Strategic programming environments Strategic programming is general-purpose generic programming since the implementation of traversal functionality is a recurring theme. The more complex the traversed data structures are (as for the numbers of types and constructors), and the more components with traversal functionality are needed, the more substantial are the benefits one can expect from strategic application development. This motivates the typical application domains for strategic programming: language processing, document processing, program transformation and analysis. For these domains, it is however crucial that plain strategic programming is embedded into a programming environment with the following architectural facilities (see Fig. 7):

- Converters are needed to derive datatypes that are native to the strategic programming incarnation from external definitions of data structures such as context-free grammars (SDF grammars [13] in our case) and XML schemas (or XML DTDs).
- Interfaces are needed to integrate external components for parsing and unparsing external representations such as sources, documents, or storage formats (the ATerm format [5] in our case). 'Strategic' components interact with other 'external' components of the application using an efficient interchange format (such as ATerms).
- Application development takes advantage of a library that provides reusable traversal schemes, functionality for XML processing and ATerm interchange, and themes for language processing [41,28,24],

The rewriting, the functional and the object-oriented incarnations are supported by corresponding programming environments Stratego/XT, Strafunski, JJForester/JJTraveler. These software bundles together with documentation, and related research papers are freely available from the following locations:

- `http://www.stratego-language.org/`
- `http://www.cs.vu.nl/Strafunski/`
- `http://www.jjforester.org/`

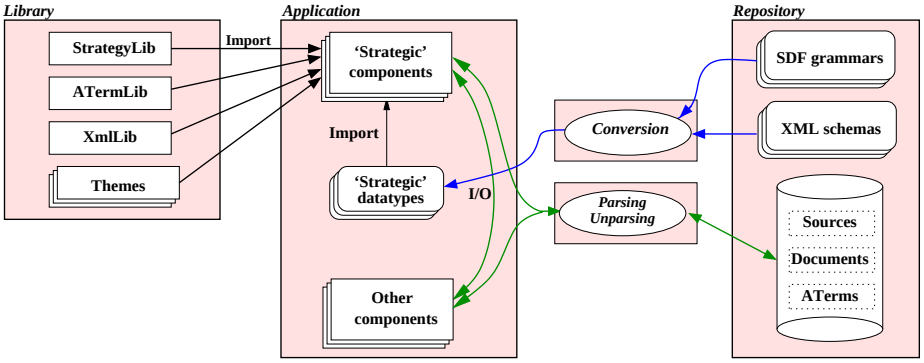

Fig. 7. The sketch of a strategic programming environment

Applications Strategic programming has been applied to a range of applications such as those discussed in [45,41,16,1,18,30,9,46,6,24]. We will briefly explain a few typical applications:

- In [1], a transformation system *CodeBoost* for *domain-specific optimisation* of C++ programs in the domain of numeric programming is described. It was implemented in Stratego making use of the XT bundle [18] of tools for program transformation which includes packages for parsing and pretty printing.
- In [30], program refactoring for Java is chosen as the running example to illustrate generic functional programming based on Strafunski. In [24], the refactoring theme is generalised in the sense that a language-parametric framework is specified. Higher-orderness and strong typing as provided by the Haskell incarnation proved to be valuable in this framework development.
- In [9], the *program understanding* tool ControlCruiser is described which reconstructs and visualises COBOL control flow. The JJForester/JJTraveler architecture has been used for the implementation.

9 Conclusion

We have reconstructed strategic programming in a language-independent manner to clearly identify its essence, its contribution, and its benefits. Strategic programming has been successfully incarnated in a number of paradigms. Applications of this idiom cover a wide range and clearly illustrate the benefits of the strategic method. So what's next? Here are a few indications:

- The issue of optimisation deserves future work, e.g., typed-based optimisation of traversals, memoisation techniques, or fusion-like principles for traversal strategies (the latter being addressed to some extent in [16]).
- Strategic programming should be elaborated to cover generic datatype-changing transformations [26], as needed, for example, in the context of XML format evolution [26]. The notion of type-indexed datatypes [14] might be useful in this context.

– The interaction of strategies and constraint-programming needs to be investigated. Also, the precise relation between strategic programming and attribute grammars has to analysed.
– The precision of types — even in a strongly typed setting — could be further improved as for success and failure behaviour of strategies, but also regarding differentiated types of traversal scheme parameters to express different kinds of control.

Acknowledgement The idiom of strategic programming has taken shape over several years of research at several academic institutions and has involved many people other than the authors. Bas Luttik [34] is the co-inventor of several one-layer traversal primitives. Zino Benaissa and Andrew Tolmach contributed to the Stratego design. Tobias Kuipers contributed to the design of JJForester. The XT bundle relies on important contributions by Merijn de Jonge. Thanks are due to Otto Skrove Bagge, Martin Bravenboer, Arne de Bruijn, Eelco Dolstra, Patricia Johann, Dick Kieburtz, Karina Olmos, and Hedzer Westra for their applications of Stratego/XT and other contributions. We would like to thank Dave Clarke, Jan Heering, Jan Kort and Karl Lieberherr for discussions on the subject of the paper and for feedback on an initial version of the paper.

References

1. O. S. Bagge, M. Haveraaen, and E. Visser. CodeBoost: A Framework for the Transformation of C++ Programs. Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University, 2001.
2. P. Borovansky, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting. In Meseguer [36].
3. M. Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, CWI, July 2001.
4. M. Brand, M. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997.
5. M. v. d. Brand, H. d. Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software—Practice & Experience*, 30(3):259–291, Mar. 2000.
6. M. Bravenboer and E. Visser. Rewriting Strategies for Instruction Selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *LNCS*, pages 237–251, Copenhagen, Denmark, July 2002. Springer-Verlag.
7. D. Clarke and A. Löh. Generic Haskell, Specifically. In J. Gibbons and J. Jeuring, editors, *Proc. of the IFIP TC2 Working Conference on Generic Programming*. Kluwer Academic Publishers, 2002. To appear.
8. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [36].
9. A. v. Deursen and J. Visser. Building Program Understanding Tools using Visitor Combinators. In *Proceedings 10th Int. Workshop on Program Comprehension, IWPC 2002*. IEEE Computer Society, 2002.
10. E. Dolstra and E. Visser. First-class rules and generic traversal. Technical report, Universiteit Utrecht, UU-CS-2001-38, 2001.
11. T. Elrad and R. Filman, editors. Special issue on aspect-oriented programming. *Communications of the ACM*, 44(10), Oct. 2001.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
13. J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

14. R. Hinze, J. Jeuring, and A. Löh. Type-Indexed Data Types. In *Proc. of the Sixth International Conference on Mathematics of Program Construction (MPC 2002)*, volume 2386 of *LNCS*, pages 148–174. Springer-Verlag, 2002.
15. J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, Berlin, 1996.
16. P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):1–34, 2000.
17. P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 57. Elsevier Science Publishers, 2001.
18. M. Jonge, E. Visser, and J. Visser. XT: a Bundle of Program Transformation Tools. In M. van den Brand and D. Parigot, editors, *ENTCS*, volume 44. Elsevier Science Publishers, 2001.
19. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001.
20. G. Kiczales, J. Lamping, et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, number 1241 in LNCS. Springer Verlag, 1997.
21. J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. In M. van den Brand and R. Lämmel, editors, *Second Workshop on Language Descriptions, Tools and Applications (LDTA 2002), Grenoble, France*, volume 65 of *ENTCS*. Elsevier Science Publishers, Apr.13 2002.
22. R. A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, July 1979.
23. R. Lämmel. The Sketch of a Polymorphic Symphony. In B. Gramlich and S. Lucas, editors, *Proc. of International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, ENTCS. Elsevier Science Publishers, July 2002. To appear.
24. R. Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, Oct.5 2002. ACM Press. 14 pages.
25. R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies, 2002. 85 pages; To appear in the Journal of Logic and Algebraic Programming; First published as CWI Technical Report SEN-R0122 in August 2001.
26. R. Lämmel and W. Lohmann. Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.
27. R. Lämmel and G. Riedewald. Prological Language Processing. In M. Brand and D. Parigot, editors, *Proc. LDTA'01*, volume 44 of *ENTCS*. Elsevier Science, Apr. 2001.
28. R. Lämmel and J. Visser. Design Patterns for Functional Strategic Programming. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, Oct.5 2002. ACM Press. 14 pages.
29. R. Lämmel and J. Visser. Strategic polymorphism requires just two combinators! In *Preproceedings of IFL 2002, Implementation of Functional Languages*, Sept. 2002. 18 pages.
30. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In S. Krishnamurthi and C. Ramakrishnan, editors, *Proc. of PADL 2002, Portland, OR, USA*, volume 2257 of *LNCS*. Springer-Verlag, Jan. 2002.
31. R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.
32. K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

33. K. Lieberherr. Connections between Demeter/Adaptive Programming and Aspect-Oriented Programming, 2002. Retrieved from `http://www.ccs.neu.edu/home/lieber/`.
34. B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.
35. E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. of FPCA'91*, volume 523 of *LNCS*. Springer-Verlag, 1991.
36. J. Meseguer, editor. *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, RWLW'96, (Asilomar, Pacific Grove, CA, USA)*, volume 4 of *ENTCS*, Sept. 1996.
37. D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer-Verlag. 8 pages.
38. J. Ovlinger and M. Wand. A language for specifying recursive traversals of object structures. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 70–81, 1999.
39. L. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, Aug. 1983.
40. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.
41. E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP'00)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.
42. E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.
43. E. Visser. A survey of strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 57. Elsevier Science Publishers, 2001.
44. J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices, OOPSLA 2001 Conference Proceedings*, 36(11):270–282, Nov. 2001.
45. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
46. H. Westra. Configurable transformations for high-quality automatic program improvement. CobolX: a case study. Master's thesis, Utrecht University, The Netherlands, February 2002.
47. XSL Transformations (XSLT) Version 1.0, Nov. 1999. `http://www.w3.org/TR/xslt`.