Tutorial On
Strategic Programming Across Programming Paradigms[1]

Joost Visser
joost.visser@di.uminho.pt

João Saraiva
jas@di.uminho.pt

Department of Informatics
University of Minho
Portugal
http://www.di.uminho.pt/

[1]Partially funded by Fundação para a Ciência e Tecnologia, grants No. SFRH/BPD/11609/2002 and POSI/CHS/44304/2002.

Contents

Chapter 1

Introduction

Strategic programming is a form of generic programming that combines the notions of *one-step traversal* and *dynamic nominal type case* into a powerful combinatorial style of traversal construction. Strategic programming allows novel forms of abstraction and modularization that are useful for program construction in general. In particular when large heterogeneous data structures are involved (e.g. in document and language processing), strategic programming techniques enable a high level of conciseness, composability, structure-shyness, and traversal control [Vis03b].

The basic concepts of strategic programming are independent of the programming paradigm. The tutorial will include an explanation of these concepts in a language-independent way. For various specific programming languages, *incarnations* of strategic programming have been realized in various ways. The tutorial will cover incarnations in several main paradigms, among which an object-oriented incarnation in Java [Vis01b], a functional incarnation in Haskell [LV02b, LV03], and an attribute-grammar incarnation in LRC [Sar99, KS98]. For each of these incarnations, the mapping of basic strategic programming concepts to specific language constructs will be discussed, as well as a quick start introduction to the supporting tools and libraries.

Since strategic programming enriches the programmer's repertoire of abstraction and modularization techniques, it has given rise to a body of programming experience. This experience is laid down in design patterns and programming idioms that guide the programmer in his programmming activities [LV02a, Vis01a]. The tutorial includes a presentation of these design patterns and idioms involving typical examples and sample code.

Strategic programming has been used in various non-trivial applications, including program understanding tools, refactoring tools, and incremental document processors [DV02, LV03, TR03, LRT03, SS03]. The tutorial will include a cursory treatment of the strategic internals of one or more show-case applications. The design trade-offs and lessons-learned from these applications will be shared with the participants. Also, these applications will be used to demonstrate practical implications of strategic programming in the large.

Acknowledgements. We would like to thank Ralf Lämmel and Eelco Visser for the collaboration they provided in research projects related to strategic programming.

1.1 Topics and Schedule

- **Tutorial length**: 3 hours
- Motivation (30 minutes)
  - What is Strategic Programming?
  - What is Strategic Programming good for?
  - Series of motivating examples from different paradigms.
- Basic concepts of Strategic Programming (30 minutes)
  - Expressiveness: one-layer traversal and dynamic type-case
  - Methodology: brew your own in three steps
  - Characteristics: an abstract definition of 'strategies'
  - Algebra: a set of basic strategy combinators
  - Compound strategy combinators and growing combinator libraries
- Incarnations of Strategic Programming across paradigms (45 minutes)
  - Term Rewriting (the Stratego system)
  - Functional Programming (Haskell: the Strafunski library)
  - Object-Oriented (Java: the JJTraveler library and JJForester generator)
  - Attribute Grammars (Strategic LRC)
  - Document Processing (XML: HaQuery)
- Design patterns and programming idioms (45 minutes)
  - Traversal idioms: fix-points, propagation, accumulation, backtracking, ...
  - Transformation idioms: cascading, staging, nesting, ...
  - Variation points: completeness, order, cut-offs, ...
  - Object-oriented: circularity, sharing, state, ...
  - Functional: use of monads, meta-schemes, type guards, ...
- Strategic Programming in the large (30 minutes)
  - General considerations.
  - *Ex*: Cobol Reverse Engineering (developed at CWI - SIG)
  - *Ex*: HaRe, A Haskell Refactoring Tool (developed at Kent University)
  - *Ex*: HaQuery: A Query Language for XML (developed at Minho University)

Tutorial's Organization: This tutorial covers all aspects of Strategic Programming: fundamental concepts (paradigm-independent), incarnations in representatives of specific programming paradigms (OOP, functional programming, term rewriting, attribute grammars), common and best programming practices, and application in-the-large. At each of these levels, explanations are tuned to helping participants get started with Strategic Programming. Ample use will be made of examples. To gain deeper understanding of particular issues, pointers will be given to background material (see also *References*).

Since no previous knowledge or experience with Strategic Programming is expected, we start with introducing the concept, and providing motivation for it. In particular, we will discuss the potential benefits that Strategic Programming offers in practical programming projects. To appeal to programmers of all persuasions, motivating examples will be presented from various programming paradigms.

The basic concepts of Strategic Programming will be discussed in a paradigm-independent way. In particular, the concepts of one-layer traversal and dynamic type-case will be explained, as well as their interplay [LV02c, LP03]. A three-step methodology for creating strategic programs will be presented. A small set of defining characteristics will be given to delineate the concept of strategies in an abstract, i.e. paradigm-independent way. This abstract definition will be made more concrete by discussing a strategic programming algebra, i.e. a minimal set of strategic programming combinators and their properties. Finally, we will explain by example how compound strategies can be composed from basic ones in order to create libraries of reusable strategy combinators [SAS99].

In the second hour, specific incarnations of Strategic Programming in well-known programming paradigms will be presented. Depending on the interest of the participants, some paradigms may be given more emphasis than others. A very elegant and easy-to-grasp incarnation of Strategic Programming is the Stratego term-rewriting system [Vis03a, VBT99]. A functional programming incarnation is available in the form of the Strafunski bundle for generic functional programming [LV02b]. An object-oriented incarnation is realized by the Java-based JJTraveler library, in combination with the JJForester visitor generator [KV01]. The embedding of strategies in the attribute grammar formalism will be discussed and the strategic extension of the LRC attribute grammar based system will be presented [KS98, Sar02]. Finally, we will discuss HaQuery, a DSL for strategic processing of XML documents [SS03].

Since strategic programming enriches the programmer's repertoire of abstraction and modularization techniques, it has given rise to a body of programming experience. This experience is laid down in design patterns and programming idioms that guide the programmer in his program construction activities. The tutorial includes a presentation of these design patterns and idioms involving typical examples and sample code. These will be taken from various programming paradigms, and may be tuned to the interests and wishes of the participants of the tutorial. The objective is to bring the participants to a point where they are ready to start programming strategically in their preferred languages.

Finally, we will discusse the use of strategic programming in the real world. We will discuss several large projects in which essential use has been made of strategic programming techniques. One of them is the developement of Cobol reverse engineering support

with object-oriented strategic programming techniques [DV02]. Another is the Haskell refactoring tool, HaRe [TR03, LRT03], developed with functional strategic programming techniques. Also, the application of strategic programming for XML document processing will be explained, using the HaQuery language. For each of these applications we will point out how strategic programming techniques have been deployed, why they were beneficial for the success of the projects, and what strategic programming lessons were learned in the course of them.

1.2 Curriculum Vitæ: Joost Visser

Joost Visser is a post-doctoral fellow at the University of Minho, Portugal. Strategic Programming is one of his main research topics. Joost carried out his Ph.D. research at the CWI in Amsterdam on the topic of *generic traversal over typed source code representations*. He is co-designer and co-developer of Haskell-based and Java based tools for strategic programming. As senior architect and consultant at the Software Improvement Group, The Netherlands, he has experience with applying strategic programming techniques in industrial settings, in particular for the tool-based analysis of large legacy software systems.

1.3 Curriculum Vitæ: João Saraiva

João Saraiva is an Auxiliar Professor of Computer Science at University of Minho. His research is focused on programming language design and implementation, and functional programming. João finished a Ph.D. program at Utrecht University, The Netherlands, in December 1999 where he worked on purely functional implementation of attribute grammars. During his Ph.D. and now as part of his academic activities (both research and teaching) his work has been concerned with the LRC system: a generator of purely functional and incremental language-based tools.

Chapter 2

Motivation

Since no previous knowledge or experience with Strategic Programming is expected, we start with introducing the concept, and providing motivation for it. In particular, we will discuss the potential benefits that Strategic Programming offers in practical programming projects. To appeal to programmers of all persuasions, motivating examples will be presented from various programming paradigms.

2.1 What is Strategic Programming?

Strategic programming is a form of generic programming that combines the notions of *one-step traversal* and *dynamic nominal type case* into a powerful combinatorial style of traversal construction.

2.2 What is Strategic Programming good for?

Strategic programming allows novel forms of abstraction and modularization that are useful for program construction in general. In particular when large heterogeneous data structures are involved (e.g. in document and language processing), strategic programming techniques enable a high level of conciseness, composability, structure-shyness, and traversal control [Vis03b].

2.3 Examples from different paradigms.

Let's consider some examples of using a typeful programming approach to solve traversal problems.

Suppose the source code representation at hand is the AST of a syntax definition formalism, say EBNF, and among the operations we want to implement are (i) collecting all non-terminals, and (ii) normalizing optional symbols (replace all regular expressions of the form $[R]$ with expressions of the form $R|\epsilon$). Figure 2.1 shows an abstract syntax

$$
\begin{array}{lcl}
\mathit{Grammar} & := & \mathit{Grammar}(\mathit{NonTerminal}, \mathit{Prod}^*) \\
\mathit{Prod} & := & \mathit{Prod}(\mathit{NonTerminal}, \mathit{RegExp}) \\
\mathit{RegExp} & := & T(\mathit{Terminal}) \\
 & | & N(\mathit{NonTerminal}) \\
 & | & \mathit{Empty} \\
 & | & \mathit{Star}(\mathit{RegExp}) \\
 & | & \mathit{Plus}(\mathit{RegExp}) \\
 & | & \mathit{Opt}(\mathit{RegExp}) \\
 & | & \mathit{Seq}(\mathit{RegExp}, \mathit{RegExp}) \\
 & | & \mathit{Alt}(\mathit{RegExp}, \mathit{RegExp})
\end{array}
$$

Terminal and *NonTerminal* are the set of terminal symbols, and the set of non-terminal symbols.

Figure 2.1: Abstract syntax of EBNF.

$$
\begin{array}{lcl}
\mathit{Grammar} & : & \mathit{NonTerminal} * \mathit{Prods} \rightarrow \mathit{Grammar} \\
\mathit{ProdsNil} & : & \mathit{Prods} \\
\mathit{ProdsCons} & : & \mathit{Prod} * \mathit{Prods} \rightarrow \mathit{Prods} \\
\mathit{Prod} & : & \mathit{NonTerminal} * \mathit{RegExp} \rightarrow \mathit{Prod} \\
T & : & \mathit{Terminal} \rightarrow \mathit{RegExp} \\
N & : & \mathit{NonTerminal} \rightarrow \mathit{RegExp} \\
\mathit{Empty} & : & \mathit{RegExp} \\
\mathit{Star} & : & \mathit{RegExp} \rightarrow \mathit{RegExp} \\
\mathit{Plus} & : & \mathit{RegExp} \rightarrow \mathit{RegExp} \\
\mathit{Opt} & : & \mathit{RegExp} \rightarrow \mathit{RegExp} \\
\mathit{Seq} & : & \mathit{RegExp} * \mathit{RegExp} \rightarrow \mathit{RegExp} \\
\mathit{Alt} & : & \mathit{RegExp} * \mathit{RegExp} \rightarrow \mathit{RegExp}
\end{array}
$$

Figure 2.2: First-order signature that represents the abstract syntax of EBNF.

for EBNF (in the form of a tree grammar) that consists of 5 sorts (node types) and 10 productions (node constructors). Let's sketch the 'text-book' approach to solving these problems in various strongly typed programming language paradigms.

Term rewriting

In term rewriting the abstract syntax of EBNF can be represented with a first-order signature, as shown in Figure 2.2. The main difference with the tree grammar of Figure 2.1 is that the iteration of productions (Prod^*) has been expanded into the sort *Prods*. Solutions to our two example problems are shown in Figure 2.3. We will now explain these solutions.

To solve the collection problem (i) in a term rewriting system, we need to introduce a new function symbol coll_S of type $S \rightarrow \mathit{NonTermSet}$ for each (non-lexical) sort S. Here we assume that a sort *NonTermSet* for sets of non-terminals has been previously

Collection of non-terminals in 'functional' rewriting style:

$$
\begin{array}{lcl}
coll_{Grammar} & : & Grammar \rightarrow NonTermSet \\
coll_{Prods} & : & Prods \rightarrow NonTermSet \\
coll_{Prod} & : & Prod \rightarrow NonTermSet \\
coll_{RegExp} & : & RegExp \rightarrow NonTermSet
\end{array}
$$

$$
\begin{array}{lcl}
coll_{Grammar}(Grammar(nt, ps)) & \rightsquigarrow & \{nt\} \cup coll_{Prods}(ps) \\
coll_{Prods}(ProdsNil) & \rightsquigarrow & \emptyset \\
coll_{Prods}(ProdsCons(p, ps)) & \rightsquigarrow & coll_{Prod}(p) \cup coll_{Prods}(ps) \\
coll_{Prod}(Prod(nt, re)) & \rightsquigarrow & \{nt\} \cup coll_{RegExp}(re) \\
coll_{RegExp}(T(t)) & \rightsquigarrow & \emptyset \\
coll_{RegExp}(N(nt)) & \rightsquigarrow & \{nt\} \\
coll_{RegExp}(Empty) & \rightsquigarrow & \emptyset \\
coll_{RegExp}(Star(re)) & \rightsquigarrow & coll_{RegExp}(re) \\
coll_{RegExp}(Plus(re)) & \rightsquigarrow & coll_{RegExp}(re) \\
coll_{RegExp}(Opt(re)) & \rightsquigarrow & coll_{RegExp}(re) \\
coll_{RegExp}(Seq(re_1, re_2)) & \rightsquigarrow & coll_{RegExp}(re_1) \cup coll_{RegExp}(re_2) \\
coll_{RegExp}(Alt(re_1, re_2)) & \rightsquigarrow & coll_{RegExp}(re_1) \cup coll_{RegExp}(re_2)
\end{array}
$$

Normalization of optionals in 'pure' rewriting style.

$$Opt(re) \quad \rightsquigarrow \quad Alt(re, Empty)$$

Normalization of optionals in 'functional' rewriting style.

$$
\begin{array}{lcl}
norm_{Grammar} & : & Grammar \rightarrow Grammar \\
norm_{Prods} & : & Prods \rightarrow Prods \\
norm_{Prod} & : & Prod \rightarrow Prod \\
norm_{RegExp} & : & RegExp \rightarrow RegExp
\end{array}
$$

$$
\begin{array}{lcl}
norm_{Grammar}(Grammar(nt, ps)) & \rightsquigarrow & Grammar(nt, norm_{Prods}(ps)) \\
norm_{Prods}(ProdsNil) & \rightsquigarrow & ProdsNil \\
norm_{Prods}(ProdsCons(p, ps)) & \rightsquigarrow & ProdsCons(norm_{Prod}(p), norm_{Prods}(ps)) \\
norm_{Prod}(Prod(nt, re)) & \rightsquigarrow & Prod(nt, norm_{RegExp}(re)) \\
norm_{RegExp}(T(t)) & \rightsquigarrow & T(t) \\
norm_{RegExp}(N(nt)) & \rightsquigarrow & N(nt) \\
norm_{RegExp}(Empty) & \rightsquigarrow & Empty \\
norm_{RegExp}(Star(re)) & \rightsquigarrow & Star(norm_{RegExp}(re)) \\
norm_{RegExp}(Plus(re)) & \rightsquigarrow & Plus(norm_{RegExp}(re)) \\
norm_{RegExp}(Opt(re)) & \rightsquigarrow & Alt(norm_{RegExp}(re), Empty) \\
norm_{RegExp}(Seq(re_1, re_2)) & \rightsquigarrow & Seq(norm_{RegExp}(re_1), norm_{RegExp}(re_2)) \\
norm_{RegExp}(Alt(re_1, re_2)) & \rightsquigarrow & Alt(norm_{RegExp}(re_1), norm_{RegExp}(re_2))
\end{array}
$$

Figure 2.3: Implementations of EBNF operations in term rewriting.

defined together with appropriate operations on them. Furthermore, for all these additional function symbols, a rewrite rule must be added for each production of the argument sort. These rules perform recursive calls on all subterms except those of type *NonTerminal*. The results of the recursive calls are concatenated with each other and with singleton sets that contain the encountered non-terminals. This style of rewriting can be called the 'functional style' in view of the pervasive use of additional function symbols.

To solve the normalization problem (ii), two alternative avenues can be taken. Firstly, one can refrain from introducing additional function symbols and solve the problem in a 'pure' rewriting style. To this end, a single rewrite rule is added which simply rewrites $Opt(re)$ to $Alt(re, Empty)$. This solution is very concise, but problematic when more traversals need to be implemented in a single rewrite system. The lack of function symbols results in a lack of control over the scheduling of traversals and to which subterms they are applied. If, for instance, our application needs to return not only the normalized grammar, but must also report which expressions have been eliminated, this is impossible, simply because we can not prevent the eliminated expressions from being normalized as well. Also, if we want to implement the *introduction* rule for optional expressions in the same system, we will immediately obtain a non-terminating rewrite system.

The second avenue to solve the normalization problem is to again use the functional style of rewriting. This time, function symbols $norm_S : S \rightarrow S$ are introduced for all sorts S. For $norm_N(Opt(re))$, a rule is added that reduces to $Alt(norm_N(re), Empty)$. For all other productions, a rule is added that recursively calls the appropriate normalization functions on all subterms, and reconstructs the term with the results as subterms. Here, conciseness is lost, but traversal control is regained. For instance, traversal can be cut off by omitting a recursive call, and traversals can be sequenced by applying functions in a particular order.

Functional programming

In functional programming, the abstract syntax of EBNF would be represented by a set of algebraic datatypes. This is shown in Figure 2.4. Both operations (i) and (ii) can then be implemented in a fashion quite similar to the functional style of rewriting discussed above. These are shown in Figure 2.5. Apart from syntax, the differences are minor and not relevant for our particular problem (e.g. the iteration of productions $Prod^*$ is represented by a list $[Prod]$ which is processed with the polymorphic *map* function rather than by a dedicated function; also, lists are used to represent sets of non-terminals).

In contrast to first-order term rewriting languages, functional programming languages support parametric polymorphism and higher-order types. We can make use of these features to implement our EBNF operations with *generalized folds* [MFP91]. We would start by defining a function $fold_S$ for every datatype S, as shown in Figure 2.6. These functions take as many arguments as there are data constructor functions in our set of datatypes, i.e. as there are productions in the abstract grammar. These arguments can be grouped into a fold *algebra*, which is modeled in Haskell by a record Alg_{EBNF}. The type of each argument (record member) reflects the type of the constructor function to which

```
data Grammar     = Grammar NonTerminal [Prod]
data Prod        = Prod NonTerminal RegExp
data RegExp      = T Terminal
                 | N NonTerminal
                 | Empty
                 | Star RegExp
                 | Plus RegExp
                 | Opt RegExp
                 | Seq RegExp RegExp
                 | Alt RegExp RegExp
type Terminal    = String
type NonTerminal = String
```

Figure 2.4: Haskell datatypes that represent the abstract syntax of EBNF.

it corresponds. For instance, the constructor $Opt : RegExp \rightarrow RegExp$ is represented by an argument of type $re \rightarrow re$, where re is a type variable that represents occurrences of *RegExp*. Together, the fold functions capture the scheme of primitive recursion over our set of datatypes. By supplying appropriate functions as arguments to the function $fold_{Grammar}$, the EBNF operations can be reconstructed, as shown in Figure 2.7. For collection, these arguments are empty lists or repeated list concatenations for most cases, and a singleton construction function for the argument that corresponds to *NonTerminal*. For normalization, all arguments are instantiated to the constructor functions to which they correspond, except for the argument corresponding to *Opt*, which is instantiated to the function $\lambda re \rightarrow Alt\ re\ Empty$.

Thus, by using folds we are able to reuse the recursion scheme between various operations on the same source code representation, as long as they can be solved with primitive recursion. Note that the use of (generalized) folds has been advocated mainly to facilitate reasoning about programs and optimizing them on the basis of the mathematical properties of folds. The possibility of using them to improve reuse is largely unexplored (but see [LVK00a]).

Object-oriented programming

In class-based object-oriented programming, the abstract syntax of EBNF can be represented with a class hierarchy, as shown in Figure 2.8. The most straightforward approach to implementing operations (i) and (ii) is by adding corresponding methods to each of the classes in the hierarchy. For each class C, the methods have signatures $coll(Set) : void$ and $norm() : C$. The bodies of these methods are implemented in a way quite similar to the functional and rewriting implementations. They mostly make recursive method calls on their components, and only the bodies of $N.coll()$ and $Opt.norm()$ implement 'interesting' behavior. Figure 2.8 shows the implementation of these two methods in Java. Here,

Collection of non-terminals:

$$
\begin{array}{lcl}
coll_{Grammar} & :: & Grammar \rightarrow [NonTerminal] \\
coll_{Grammar}\ (Grammar\ nt\ ps) & = & [nt] +\!\!+ (concat\ (map\ coll_{Prod}\ ps)) \\
coll_{Prod} & :: & Prod \rightarrow [NonTerminal] \\
coll_{Prod}\ (Prod\ nt\ re) & = & [nt] +\!\!+ (coll_{RegExp}\ re) \\
coll_{RegExp} & :: & RegExp \rightarrow [NonTerminal] \\
coll_{RegExp}\ (T\ t) & = & [\,] \\
coll_{RegExp}\ (N\ nt) & = & [nt] \\
coll_{RegExp}\ Empty & = & [\,] \\
coll_{RegExp}\ (Star\ re) & = & coll_{RegExp}\ re \\
coll_{RegExp}\ (Plus\ re) & = & coll_{RegExp}\ re \\
coll_{RegExp}\ (Opt\ re) & = & coll_{RegExp}\ re \\
coll_{RegExp}\ (Seq\ re\_1\ re\_2) & = & (coll_{RegExp}\ re\_1) +\!\!+ (coll_{RegExp}\ re\_2) \\
coll_{RegExp}\ (Alt\ re\_1\ re\_2) & = & (coll_{RegExp}\ re\_1) +\!\!+ (coll_{RegExp}\ re\_2)
\end{array}
$$

Normalization of optionals:

$$
\begin{array}{lcl}
norm_{Grammar} & :: & Grammar \rightarrow Grammar \\
norm_{Grammar}\ (Grammar\ nt\ ps) & = & Grammar\ nt\ (map\ norm_{Prod}\ ps) \\
norm_{Prod} & :: & Prod \rightarrow Prod \\
norm_{Prod}\ (Prod\ nt\ re) & = & Prod\ nt\ (norm_{RegExp}\ re) \\
norm_{RegExp} & :: & RegExp \rightarrow RegExp \\
norm_{RegExp}\ (T\ t) & = & T\ t \\
norm_{RegExp}\ (N\ nt) & = & N\ nt \\
norm_{RegExp}\ Empty & = & Empty \\
norm_{RegExp}\ (Star\ re) & = & Star\ (norm_{RegExp}\ re) \\
norm_{RegExp}\ (Plus\ re) & = & Plus\ (norm_{RegExp}\ re) \\
norm_{RegExp}\ (Opt\ re) & = & Alt\ (norm_{RegExp}\ re)\ Empty \\
norm_{RegExp}\ (Seq\ re\_1\ re\_2) & = & Seq\ (norm_{RegExp}\ re\_1)\ (norm_{RegExp}\ re\_2) \\
norm_{RegExp}\ (Alt\ re\_1\ re\_2) & = & Alt\ (norm_{RegExp}\ re\_1)\ (norm_{RegExp}\ re\_2)
\end{array}
$$

We use the following standard functions on lists for appending, mapping, and concatenation:

$$
\begin{array}{lcl}
(+\!\!+) & :: & [a] \rightarrow [a] \rightarrow [a] \\
map & :: & (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
concat & :: & [[a]] \rightarrow [a]
\end{array}
$$

Figure 2.5: Haskell implementation of example problems.

Fold algebra for EBNF:

$$
\begin{array}{lll}
\textbf{data}\ Alg_{EBNF}\ g\ ps\ p\ re & & \\
\quad = Alg_{EBNF}\{\ grammar & :: & NonTerminal \to ps \to g, \\
\qquad\qquad prodsnil & :: & ps, \\
\qquad\qquad prodscons & :: & p \to ps \to ps, \\
\qquad\qquad prod & :: & NonTerminal \to re \to p, \\
\qquad\qquad t & :: & Terminal \to re, \\
\qquad\qquad n & :: & NonTerminal \to re, \\
\qquad\qquad e & :: & re, \\
\qquad\qquad star & :: & re \to re, \\
\qquad\qquad plus & :: & re \to re, \\
\qquad\qquad opt & :: & re \to re, \\
\qquad\qquad seq & :: & re \to re \to re, \\
\qquad\qquad alt & :: & re \to re \to re\ \}
\end{array}
$$

The fold algebra is modeled as a Haskell *record* with one member for each constuctor in the EBNF abstract syntax. The types of these members are derived from the types of the constructors by replacing the constant types $Grammar$, $[Prod]$, $Prod$, and $RegExp$ that stand for non-terminals by type variables g, ps, p, and re. The fold algebra is parameterized with these variables.

Fold functions for EBNF:

$$
\begin{array}{lcl}
fold_{Grammar} & :: & Alg_{EBNF}\ g\ ps\ p\ re \to Grammar \to g \\
fold_{Grammar}\ a\ (Grammar\ nt\ ps) & = & grammar\ a\ nt\ (fold_{Prods}\ a\ ps) \\
fold_{Prods} & :: & Alg_{EBNF}\ g\ ps\ p\ re \to [Prod] \to ps \\
fold_{Prods}\ a\ [\,] & = & prodsnil\ a \\
fold_{Prods}\ a\ (p : ps) & = & prodscons\ a\ (fold_{Prod}\ a\ p)\ (fold_{Prods}\ a\ ps) \\
fold_{Prod} & :: & Alg_{EBNF}\ g\ ps\ p\ re \to Prod \to p \\
fold_{Prod}\ a\ (Prod\ nt\ re) & = & prod\ a\ nt\ (fold_{RegExp}\ a\ re) \\
fold_{RegExp} & :: & Alg_{EBNF}\ g\ ps\ p\ re \to RegExp \to re \\
fold_{RegExp}\ a\ (T\ x) & = & t\ a\ x \\
fold_{RegExp}\ a\ (N\ x) & = & n\ a\ x \\
fold_{RegExp}\ a\ Empty & = & e\ a \\
fold_{RegExp}\ a\ (Star\ re) & = & star\ a\ (fold_{RegExp}\ a\ re) \\
fold_{RegExp}\ a\ (Plus\ re) & = & plus\ a\ (fold_{RegExp}\ a\ re) \\
fold_{RegExp}\ a\ (Opt\ re) & = & opt\ a\ (fold_{RegExp}\ a\ re) \\
fold_{RegExp}\ a\ (Seq\ re_1\ re_2) & = & seq\ a\ (fold_{RegExp}\ a\ re_1)\ (fold_{RegExp}\ a\ re_2) \\
fold_{RegExp}\ a\ (Alt\ re_1\ re_2) & = & alt\ a\ (fold_{RegExp}\ a\ re_1)\ (fold_{RegExp}\ a\ re_2)
\end{array}
$$

Each fold function replaces the application of a constructor C by the application of the corresponding algebra member c to the recursive applications of the fold function to the arguments of the constructor. The selection of member c from algebra a is written simply as $c\ a$.

Figure 2.6: Haskell implementation of the generalized fold for EBNF.

Collection of non-terminals:

$$
\begin{aligned}
&coll :: Grammar \rightarrow [NonTerminal] \\
&coll = fold_{Grammar}\ alg_{coll} \\
&alg_{coll} :: Alg_{EBNF}\ [NonTerminal]\ [NonTerminal]\ [NonTerminal]\ [NonTerminal] \\
&alg_{coll} = Alg_{EBNF}\{\ grammar \ = \ \lambda nt\ ps \rightarrow [nt] +\!\!+ ps, \\
&\qquad\qquad\quad prodsnil \ = \ [\,], \\
&\qquad\qquad\quad prodscons \ = \ \lambda p\ ps \rightarrow p +\!\!+ ps, \\
&\qquad\qquad\quad prod \ = \ \lambda nt\ re \rightarrow [nt] +\!\!+ re, \\
&\qquad\qquad\quad t \ = \ \lambda t \rightarrow [\,], \\
&\qquad\qquad\quad n \ = \ \lambda nt \rightarrow [nt], \\
&\qquad\qquad\quad e \ = \ [\,], \\
&\qquad\qquad\quad star \ = \ \lambda re \rightarrow re, \\
&\qquad\qquad\quad plus \ = \ \lambda re \rightarrow re, \\
&\qquad\qquad\quad opt \ = \ \lambda re \rightarrow re, \\
&\qquad\qquad\quad seq \ = \ \lambda re_1\ re_2 \rightarrow re_1 +\!\!+ re_2, \\
&\qquad\qquad\quad alt \ = \ \lambda re_1\ re_2 \rightarrow re_1 +\!\!+ re_2\ \}
\end{aligned}
$$

Most algebra members are functions that return empty lists or concatenations of their arguments. Arguments that represent non-terminals are placed in singleton lists.

Normalization of optionals:

$$
\begin{aligned}
&norm :: Grammar \rightarrow Grammar \\
&norm = fold_{Grammar}\ alg_{norm} \\
&alg_{norm} :: Alg_{EBNF}\ Grammar\ [Prod]\ Prod\ RegExp \\
&alg_{norm} = Alg_{EBNF}\{\ grammar \ = \ Grammar, \\
&\qquad\qquad\quad prodsnil \ = \ [\,], \\
&\qquad\qquad\quad prodscons \ = \ (:), \\
&\qquad\qquad\quad prod \ = \ Prod, \\
&\qquad\qquad\quad t \ = \ T, \\
&\qquad\qquad\quad n \ = \ N, \\
&\qquad\qquad\quad e \ = \ Empty, \\
&\qquad\qquad\quad star \ = \ Star, \\
&\qquad\qquad\quad plus \ = \ Plus, \\
&\qquad\qquad\quad opt \ = \ \lambda re \rightarrow Alt\ re\ Empty, \\
&\qquad\qquad\quad seq \ = \ Seq, \\
&\qquad\qquad\quad alt \ = \ Alt\ \}
\end{aligned}
$$

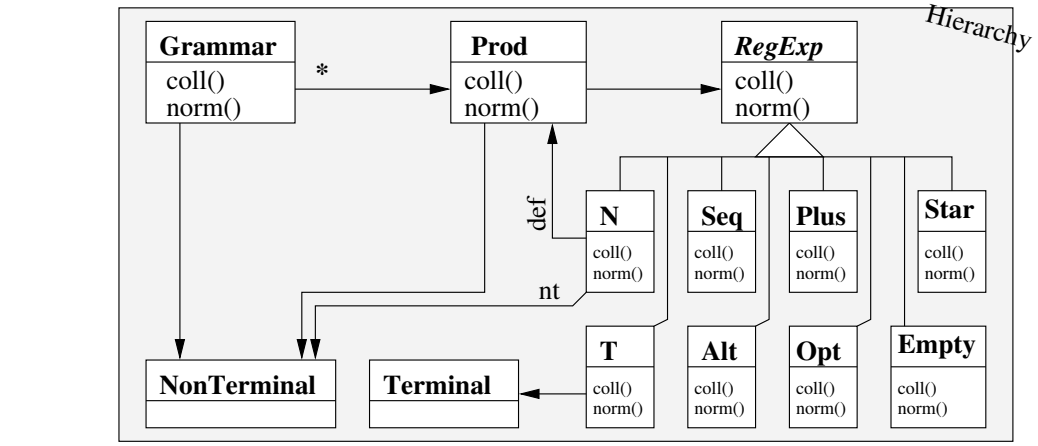
Most algebra members are the constructor functions to which they correspond. The member *opt* is a function that returns a term constructed with *Alt* and *Empty*, instead of *Opt*.

Figure 2.7: Haskell implementation of example problems, using folds.

the parameter *result* is a reference to a *Set* of non-terminals. With the *add* method, the nonterminal *nt* referred to by an object of type *NonTerminal* is added to this set.

Alternatively, one could implement the EBNF operations in accordance with the *Visitor* design pattern [GHJV94]. This is illustrated in Figure 2.9. In this approach, an $accept(Visitor)$ method is added to every class in the hierarchy, where the interface *Visitor* contains a method $visitC(C) : A$ for each concrete class C with abstract superclass A. Here, we assume *returning* visitors, i.e. visitors with visit methods that have their input type as result type, instead of *void*. Now, operations on the hierarchy can be implemented by providing implementations of the *Visitor* interface. A common approach is to first implement a default visitor that performs a top-down traversal over the object graph. Then, this top-down visitor can be specialized to implement our example problems (i) and (ii) by redefining the *visitN* and *visitOpt* methods, respectively. This is shown in the figure. In the case of collection (i), an additional field *result* needs to be added to the specialization of *Visitor* to hold the result of the collection, i.e. a set of *NonTerminal* objects. In case of normalization (ii), the component *re* of the argument *opt* is selected and used in the construction of a new *Alt* object.

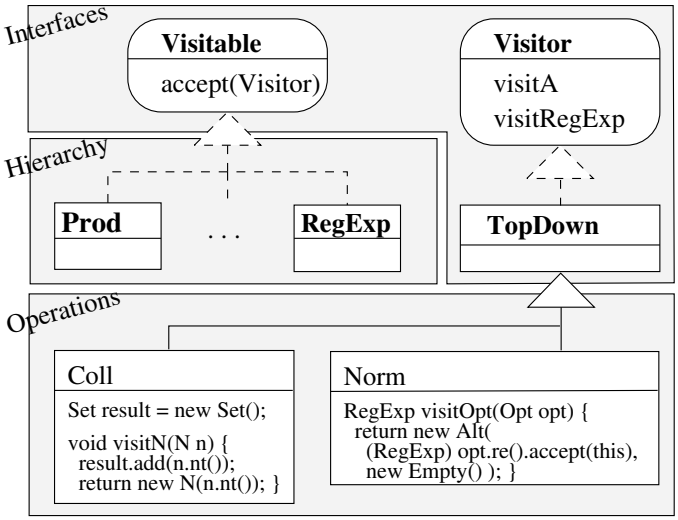
The visitor approach is somewhat similar to the fold approach in functional programming, in the sense that the recursion behavior is factored out and can be reused to implement a range of particular traversals.




```
class N extends RegExp {
  ...
  void coll(Set results) {
    results.add(nt);
  }
  ...
}
```

```
class Opt extends RegExp {
  ...
  RegExp norm() {
    return new Alt(re.norm(),
                   new Empty());
  }
  ...
}
```

Figure 2.8: UML diagram of the class hierarchy for the EBNF syntax. The Java implementation of the methods *coll*() and *norm*() are shown only for the 'interesting' cases.




```
class N extends RegExp implements Visitable {
  ...
  Visitable accept(Visitor v) {
    return v.visitN(this); }
  ...
}
```

```
class Opt extends RegExp implements Visitable {
  ...
  Visitable accept(Visitor v) {
    return v.visitOpt(this); }
  ...
}
```

```
class TopDown extends Visitor {
  ...
  public RegExp visitN(N n) {
    return new N(n.nt()); }
  ...
  public RegExp visitOpt(Opt opt) {
    return new Opt((RegExp) opt.re().accept(this)); }
  ...
  public RegExp visitAlt(Alt alt) {
    return new Alt((RegExp) alt.re1().accept(this),
                   (RegExp) alt.re2().accept(this)); }
}
```

Figure 2.9: Implementation of the example problems, using the Visitor pattern. The code excerpts show the implementation of the *Visitable* interface by the concrete classes *N* and *Opt*, as well as fragments of the default *TopDown* visitor. The UML diagram shows the specific visitors required to solve the example problems.

Lack of genericity in traditional typeful approaches

Thus, in each of the sketched typeful approaches to our little EBNF example problems, we observe that traversal of the AST is dealt with in a non-generic manner. The traversal behavior is implemented separately for each specific node type, where access to and iteration over the immediate subtrees is dealt with in a type-specific way.

Though we have intentionally constructed our examples to bear out the consequences of a typeful approach to traversal, the situation is not atypical. In traversal problems where the proportion of 'interesting' nodes is larger, where the tree needs to be traversed only partially, or in a different order, where traversals must be nested or sequenced, where side-effects or environment propagation are needed, or where other considerations add to the complexity, the bottom line remains: each type needs to be dealt with in a type-specific way, regardless of the conceptual genericity of the required behavior.

Challenges

Given the scenarios sketched above, and the general assessment that adding types leads to non-generic implementation of traversal behavior, we can now articulate a number of concrete disadvantages of using a typeful approach to traversals. As this tutorial will make clear, strategic programming techniques neutralize these disadvantages.

Conciseness

The most obvious casualty in our example scenarios is *conciseness*. Note that our example traversal problems (i) and (ii) only require 'interesting' behavior for nodes of a single type. For all the other nodes, only straightforward recursion behavior is needed. Though this recursion behavior is conceptually the same for all types, it needs to be implemented over and over again for each type. The reason is that when the data structure is heterogeneous, access to and traversal over its subelement requires dealing with many specific types in specific ways. None of the mentioned programming languages offers constructs or idioms to perform such access and traversal in a generic manner. As a result, lengthy traversal code is needed. As we will explain, strategic programming realizes conciseness also for typed traversals, which significantly reduces the effort needed to develop and maintain traversal implementations.

Composability

In all of the sketched approaches, composability of traversals is limited. Imagine, for example, one would implement a traversal that collects all terminals, in addition to the one that collects non-terminals. Could we compose the functionality of these two traversal into a single traversal that collects both terminals and non-terminals? Another form of desired composability would be to instantiate different traversal schemes with the same node action. For instance, would it be possible to reuse the node action of non-terminal

collection for a traversal that selects a single non-terminal from the AST? None of the sketched approaches allows such form of composition.

As we will show, strategic programming allows a high degree of composability where new traversals can conveniently be composed by combining and refining given functionality in a combinatorial style. This enables a high degree of reuse within applications.

Traversal control

In each of the sketched traversal approaches, the possibilities for control over the traversal are unsatisfactory. By traversal control, we mean the ability to determine which parts of the representation are visited in which order, and under which conditions.

In the functional style of rewriting, functional programming without folds, and object-oriented programming without visitors, the traversal strategy is hard-wired into the traversal itself. Traversal control can be implemented by adding parameters to the various functions or methods that implement the traversal, but this requires entangling the control mechanism with the basic functionality of the traversal throughout the code. In the fold approach, the traversal scheme is fixed in the fold function. Control is absent. In the visitor approach, the default visitor implements the basic traversal scenario. The visit method redefinitions in subclasses of this default visitor have the responsibility of iterating over the subelements of a type, and by changing the iteration behavior, some traversal control can be exerted. Here, tangling is again an issue, and control can only be implemented per node type.

Strategic programming offers powerful means of traversal control, where programmers can concisely construct the traversal strategies that their applications require.

Robustness

The traversal approaches sketched above are fragile with respect to changes in the underlying source code representation. If, for instance, a change would be needed to the representation of iterated symbols, each of the solutions would break. This is especially disappointing because the two example traversals include no 'interesting' behavior for iterated symbols. Ideally, their solutions would never break unless the representation is changed of the types they are specifically intended to deal with: non-terminals, optional symbols, alternatives and epsilon. In the functional rewriting style, the functional approach without folds, and the object-oriented approach without visitors, the implementation of every operation so far defined on the representation will need modification. In the fold approach, the fold function would need to be modified, as well as all instantiations of it. In the visitor approach, the situation is slightly better, since the default visitors must be changed, but their specializations will keep working.

In Strategic Programming, typed traversals are defined in a (largely) generic fashion, making them more robust against changes in source code representations. Furthermore, the non-generic parts can be properly separated form the generic parts, making it possible

to reuse the latter across different source code representations. This opens the door to the construction of libraries of reusable traversal components.

Chapter 3

Basic Concepts

The basic concepts of Strategic Programming will be discussed in a paradigm-independent way. In particular, the concepts of one-layer traversal and dynamic type-case will be explained, as well as their interplay [LV02c, LP03]. A three-step methodology for creating strategic programs will be presented. A small set of defining characteristics will be given to delineate the concept of strategies in an abstract, i.e. paradigm-independent way. This abstract definition will be made more concrete by discussing a strategic programming algebra, i.e. a minimal set of strategic programming combinators and their properties. Finally, we will explain by example how compound strategies can be composed from basic ones in order to create libraries of reusable strategy combinators [SAS99].

3.1 Expressiveness

The key insights underlying strategic programming are schematically represented in Figure 3.1. The first insight is that traversals can be decomposed into a traversal scheme, which captures the traversal behaviour, and a number of basic actions, which capture the functionality to be applied at various nodes. The second insight is that a traversal scheme can further be decomposed into one-step traversal behaviour and recursion. Below we will discuss both steps of the conceptual decomposition in more detail.

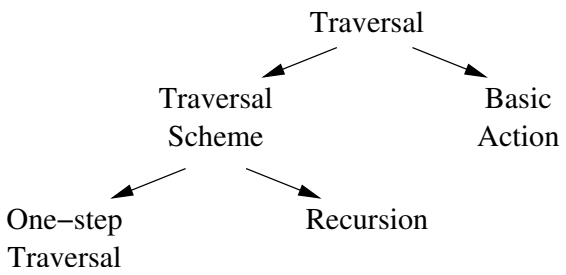


Figure 3.1: The conceptual decomposition underlying strategic programming.

Separating basic actions and traversal schemes

We will illustrate the separation of basic actions and traversals schemes by revisiting the example (ii) of the previous chapter, of normalizing optionals in EBNF expressions. A a tangled Haskell implementation of was shown in Figure 2.5.

In a first phase, we separate out the basic action for normalization, which takes the form of a function *normalizeStep* that models a single rewrite step.

$$
\begin{array}{ll}
\mathit{normalizeStep} :: \mathit{RegExp} \rightarrow \mathit{Maybe\ RegExp} & \\
\mathit{normalizeStep}\ (\mathit{Opt}\ \mathit{exp}) & = \mathit{Just}\ (\mathit{Alt}\ \mathit{exp}\ \mathit{Epsilon}) \\
\mathit{normalizeStep}\ \_ & = \mathit{Nothing}
\end{array}
$$

Here, we use the *Maybe* type constructor in order to express whether any simplification rule triggers or not, i.e. to model partiality of the rewrite rule. Note that this basic action, or rewrite step, is specific to the abstract syntax of regular expressions, and that it does not involve any recursion or traversal.

Thus, in functional programming, basic actions take the form of monomorphic non-recursive functions. In other paradigms, candidates for basic actions are rewrite rules, methods, horn-clauses, and such.

The second step is to select a traversal scheme to which we can feed our basic action to obtain a traversal. In this simple example, we attempt a beginner's favourite: *full_td* (read as full top-down):

$$
\mathit{normalize} = \mathit{full\_td}\ (\mathit{choice}\ \mathit{normalizeStep}\ \mathit{id})
$$

The traversal schemes take care of iterating basic actions all over the tree. In this definition, *full_td* performs a 'full' traversal in top-down fashion: it applies the basic action to the current node, and then recursively applies itself to the child nodes. To succeed, *full_td* requires a basic action as argument that always succeeds. Hence, we combine the basic action *normalizeStep* via the *choice* combinator with the identity *id* to recover from potential failure of *normalizeStep*. In other words, if the rewrite rule fails, the node to which it is applied will be preserved as-is.

The untangling of basic actions and traversal scheme allows the strategic programmer to combine these building blocks in different ways, according to the specific needs of his application.

Separating one-layer traversal and recursion

Apart from the example *full_td*, many more traversal schemes can be imagined and found useful. Rather than providing a finite collection of predefined traversal scheme's, strategic programming allows the composition of traversal schemes from more basic building blocks. This is where the second key insight of strategic programming comes into play: the separation of one-layer traversal from recursion.

The notion of 'one-layer' traversal presupposes that compound data (e.g. terms, objects, document elements) can be decomposed into immediate subcomponents. In the form of

one-layer traversal combinators, strategic programming provides expressiveness for *generic access* to the *immediate* components of heterogeneous data structures. 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 'appropriate' immediate sub component while preserving the overall shape of the datum. Appropriateness can be based on the success/failure behaviour of the argument strategy.

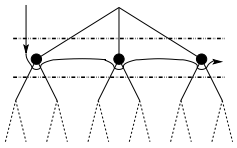
reduce Similar to *all* but the intermediate results of processing the immediate components are reduced by a binary operation.

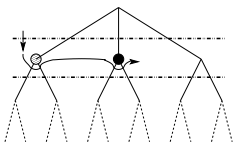
select Similar to *one* but the successfully processed immediate component is returned as the result.

Thus, the one-layer traversal combinators only operate on the immediate subcomponents of a given input datum.

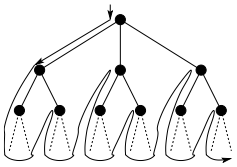
By not anticipating any scheme of recursion, one-layer traversal can still be completed into 'deep' traversal in different ways by plain recursion. Two specific one-layer combinators and derived recursive completions are illustrated in Fig. 3.2.

Below we illustrate two one-layer traversal combinators: **all** to process all immediate components, and **one** to process the left-most 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 left one completes **all** into a full top-down traversal (for short, *full_td*). The right one completes **one** into a single-hit bottom-up traversal (for short, *once_bu*).



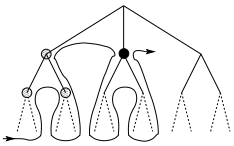


Figure 3.2: Strategic traversal with one-layer combinators

3.2 Methodology

The 'strategic methodology' can be summarised in the following 3 steps for the implementation of a piece of strategic traversal functionality:

[1] The names for these combinators differ in the SP literature. Here, we use the terms favoured in [Läm02].

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

Typically, a reusable traversal scheme is completely generic. Problem-specific actions are anticipated via parameters. The problem-specific building blocks 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.

Thus, by virtue of the conceptual decomposition of Figure 3.1, strategic programming allows the composition of problem-specific traversals from more basic building blocks, some of which are problem-specific (the basic actions) and some of which are generic and reusable (the traversal schemes). Furthermore, the stock of generic and reusable parts is inexhaustible, because the are constructed by combining one-step traversal and recursion in changing configurations.

3.3 Characteristics

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

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.

Compositionality Strategies are composed by putting together basic building blocks in different constellations. 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 characterization 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.

3.4 Algebra

We can make the strategy characterization significantly more concrete by prescribing a set of combinators that must be supported by an incarnation of strategic programming. Fig. 3.3 specifies such a set. We suggest a semi-formal reading of Fig. 3.3. In particular, the given semantic sketch leaves open how to blend with the expressiveness offered by the host paradigm of an eventual incarnation. 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.

Sequential control 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. 3.2. 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, i.e., 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:

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

This definition means that $\mathit{full\_td}(s)$ 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

Combinators

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

Notation

d	data
c	data constructors
$\overline{d}$	data with failure "$\uparrow$"
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

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

Identities

[unit]	s	$\equiv$	$\mathsf{seq}(\mathsf{id}, s)$	$\equiv$	$\mathsf{seq}(s, \mathsf{id})$	$\equiv$	$\mathsf{choice}(\mathsf{fail}, s)$	$\equiv$ $\mathsf{choice}(s, \mathsf{fail})$
[zero]	fail	$\equiv$	$\mathsf{seq}(\mathsf{fail}, s)$	$\equiv$	$\mathsf{seq}(s, \mathsf{fail})$	$\equiv$	$\mathsf{one}(\mathsf{fail})$	
[skip]	id	$\equiv$	$\mathsf{choice}(\mathsf{id}, s)$	$\equiv$	$\mathsf{all}(\mathsf{id})$			

Nested type dispatch

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

Figure 3.3: Specification of a guideline set of basic strategy combinators

datum.

Lifting type-specific actions

In Fig. 3.3, we distinguish *type-specific* actions vs. *generic* actions. 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 which eventually are 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 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 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 for static typechecking.

3.5 Compound combinators and libraries

The power of our small set of basic combinators can best be demonstrated with a few examples. Fig. 3.4 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.

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

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

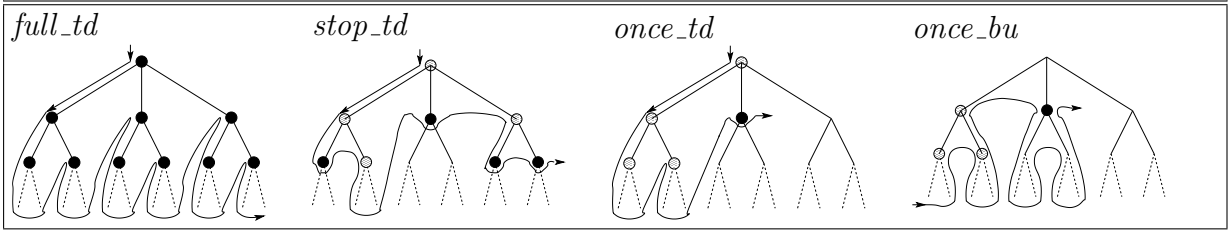


Figure 3.4: Some defined strategy combinators

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. 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.

Chapter 4

Incarnations across Paradigms

In the second hour, specific incarnations of Strategic Programming in well-known programming paradigms will be presented. Depending on the interest of the participants, some paradigms may be given more emphasis than others. A very elegant and easy-to-grasp incarnation of Strategic Programming is the Stratego term-rewriting system [Vis03a, VBT99]. A functional programming incarnation is available in the form of the Strafunski bundle for generic functional programming [LV02b]. An object-oriented incarnation is realized by the Java-based JJTraveler library, in combination with the JJForester visitor generator [KV01]. The embedding of strategies in the attribute grammar formalism will be discussed and the strategic extension of the LRC attribute grammar based system will be presented [KS98, Sar02]. Finally, we will discuss *HaQuery*, an DSL for strategic processing of XML documents [SS03].

4.1 Term Rewriting

Stratego is a language for program transformation based on the paradigm of rewriting strategies. Rewrite rules are a natural formalism to express program transformations. In standard term rewriting, terms are normalised by exhaustive applying rewrite rules to it until no further application is possible. However, in some contexts it may be necessary, for example, to apply a rule in one phase of a transformation only, or to apply a rule only to part of the term under consideration.These restrictions may be necessary to avoid non-termination, for example. To avoid this problem, Stratego makes the the rewriting strategy explicit and programmable, thus allowing the careful control over the application of transformation rules.

A Stratego program defines a transformation on first-order ground terms. Transformation rules define single transformation steps and are combined into transformation strategies by means of strategy combinators. These combinators determine where and in what order rules are applied. Stratego provides basic combinators for the composition of strategies. Strategies can be parameterised with the set of rules, or in general, the transformation to be applied by the strategy. There are six stretegy combinators to compose strategies:

sequential composition ($s1; s2$), *deterministic choice* ($s1 < +s2$; first try $s1$, only if that fails $s2$), non-deterministic choice ($s1 + s2$; same as $<=$, but the order of trying is not defined), *guarded choice* ($s1 < s2 + s3$; if $s1$ succeeds then commit to $s2$ else $s3$), *testing* ($where(s)$; ignores the transformation achieved), *negation* ($not(s)$; succeeds if s fails), and *recursion* ($rec\ x(s)$).

Strategies composed with these combinators can be named using the so-called strategy definitions. For example, the definition

$$try(s) = s < +id$$

defines the combinator *try*, which applies s to the current term. If that fails it applies id, the identity strategy which always secceeds with the original term as result. Similarly, the *repeat* stratagy

$$repeat(s) = try(s; repeat(s))$$

repeats transformation s until it fails.

The above strategy combinators combine strategies which apply transformation rules at the roots of their terms. In order to apply a rule to a subterm , the term must be traversed, obviously. Stratego provides combinators for composing generic traversals. The operator $all(s)$ applies s to each of the direct subterms t_i of a constructor application $C(t_1, \ldots, t_n)$. It succeeds if and only if the application of s to each direct subterms suceeds. In this case the resulting term is the constructor $C(t'_1, \ldots, t'_n)$, where each $T - i'$ is obtained by applying s to t_i. Next we present the definition of the *bottomup* strategy where *all* is used.

$$bottomup(s) = all(bottomup(s)); s$$

This expression defines that s is first applied recursivelly to all direct subterms of the term under consideration. If that succeeds, then s is applied to the resulting term. Similar generic traversal combinators can be defined in this way (*e.g.*, *topdown*, *alltd*, etc).

4.2 Functional Programming

Strafunski is a Haskell-centered software bundle for generic programming and language processing. Strafunski provides programming support for generic traversal as useful for the implementation of program analyses and transformation components of language processors. Strafunski is based on the notion of a functional strategy. These are generic functions that can traverse into terms of any type while mixing type-specific and uniform behaviour. Strafunski offers both a strategy combinator library StrategyLib, including generic traversal combinators, and a generative tool support based on DrIFT to use the library on large systems of data types. There are further ingredients of the Strafunski architecture that support interchange formats and derivation of algebraic datatypes from syntax definitions.

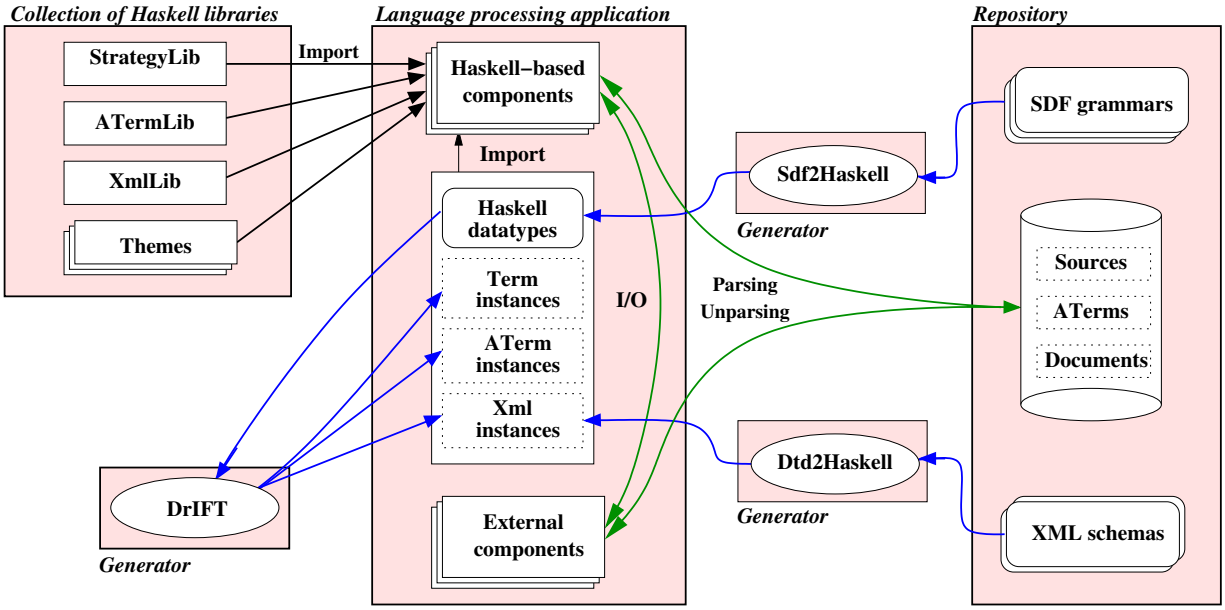


Figure 4.1: Haskell-centred language processing with Strafunski.

"Stra" refers to strategies, "fun" refers to functional programming, and their harmonious composition is a homage to the music of Igor Stravinsky. Strafunski is being developed by Ralf Lämmel and Joost Visser at CWI (Amsterdam), VU (Amsterdam), Universidade do Minho (Braga) since August 2000.

Figure 4.1 illustrates Haskell-centred language processing with Strafunski. The block labelled 'language processing application' emphasises that Haskell-based and external components coexist in an application. The components communicate on the basis of the interchange formats XML and ATerms, or they access a repository with source programs and XML documents. Haskell-based components take advantage of generic programming with functional strategies based on the Haskell library StrategyLib. Here we assume that algebraic datatypes serve for the typed representation of parse trees. Strategic programming in Haskell relies on supportive code per term type. The corresponding instances of a Term class can be generated using the DrIFT preprocessing technology. The algebraic datatypes might be derived from XML schemas (or DTDs) and syntax definitions in SDF; see the generators Sdf2Haskell and Dtd2Haskell. There are further Haskell libraries: XmlLib for XML document processing (contributed by HaXML) and ATermLib for data interchange. DrIFT is also used to generate XML instances and ATerm instances needed as mediators between Haskell terms and the interchange formats. The collection of libraries also encompasses themes for language processing such as name analyses and refactorings.

Packages in the Strafunski bundle

StrategyLib Haskell combinator library for generic programming

DrIFT-Strafunski Generative support for algebraic datatypes

ATermLib Haskell library for I/O for the ATerm interchange format

Sdf2Haskell Derivation of algebraic datatypes from syntax definitions

StrategyLib can be used as a plain Haskell combinator library on its own. The other packages provide additional convenience and support in developing language processors and in strategic programming for large syntaxes.

Themes in the StrategyLib of Strafunski Packages in the Strafunski bundle

TraversalTheme a powerful array of traversal schemes

NameTheme language-parametric name analyses

RefactoringTheme language-parametric refactoring transformations

MetricsTheme combinators for the computation of software metrics

PathTheme adaptive programming with paths on trees

FlowTheme combinators to wire up control- and data-flow

FixpointTheme traversals by fixpoint computation

KeyholeTheme strategies with hidden strategy types

EffectTheme monadic effect handling for strategies

ContainerTheme strategies as heterogeneous containers

In addition, there is support for XML processing, import chasing, monadic programming and others.

4.3 Object-Oriented

An object-oriented incarnation of strategic programming is provided by the notion of *generic visitor combinators*. Visitor combinators were introduced in [Vis01b] as an extension of the regular visitor design pattern. The aim of visitor combinators is to *compose* complex visitors from elementary ones. This is done by simply passing them as arguments to each other. Furthermore, visitor combinators offer full *control* over the traversal strategy and applicability conditions of the constructed visitors.

The use of visitor combinators leads to small, reusable classes, that have little dependence on the actual structure of the concrete objects being traversed. Thus, they are less brittle with respect to changes in the class hierarchy on which they operate. In fact, many combinators (such as the *top-down* or *breadth-first* combinators) are completely *generic*, relying only on a minimal *Visitable* interface. As a result, they can be reused for *any* concrete visitor instantiation.

Visitor combinator programming is supported by JJTraveler: a combination of a framework and library that provides *generic visitor combinators* for Java. We briefly discusses the key elements of JJTraveler.

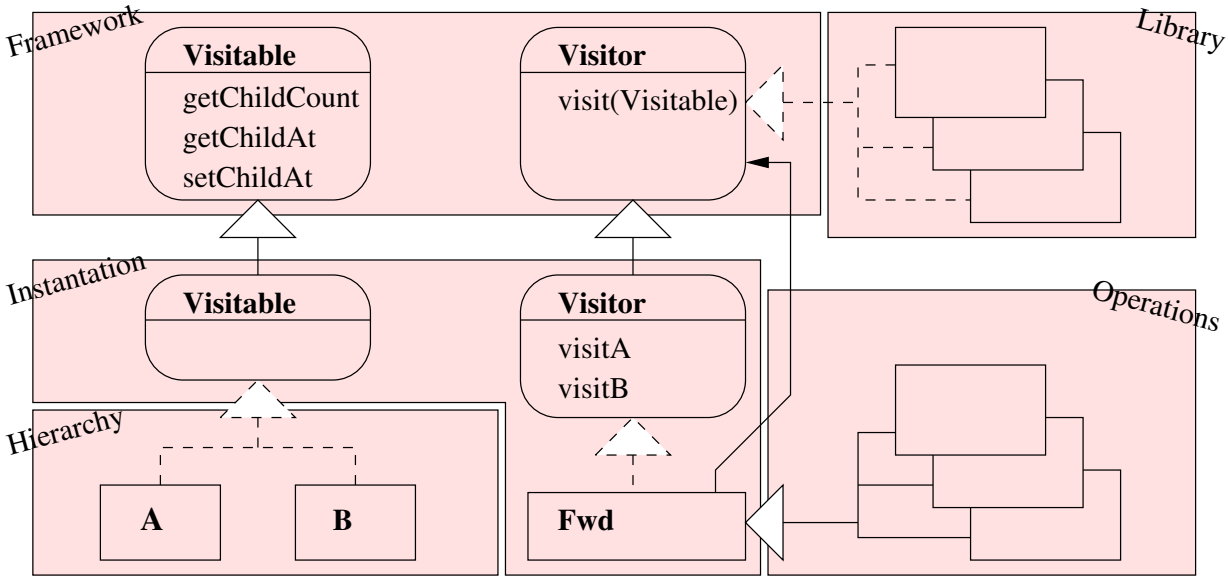


Figure 4.2: The architecture of JJTraveler.

4.3.1 The architecture of JJTraveler

Figure 4.2 shows the architecture of JJTraveler (upper half) and its relationship with an application that uses it (lower half). JJTraveler consists of a *framework* and a *library*. The application consists of a class *hierarchy*, an *instantiation* of JJTraveler's framework for this hierarchy, and the *operations* on the hierarchy implemented as visitors.

Framework The JJTraveler framework offers two generic interfaces, *Visitor* and *Visitable*. The latter provides the minimal interface for nodes that can be visited. Visitable nodes should offer three methods: to get the number of child nodes, to get a child given an index, and to modify a given child. The *Visitor* interface provides a single *visit* method that takes any visitable node as argument. Each visit can *succeed* or *fail*, which can be used to control traversal behavior. Failure is indicated by a *VisitFailure* exception.

Library The library consists of a number of predefined visitor combinators. These rely only on the generic *Visitor* and *Visitable* interfaces, not on any specific underlying class hierarchy. An overview of the library combinators is shown in Figure 4.3. They will be explained in more detail below.

Instantiation To use JJTraveler, one needs to instantiate the framework for the class hierarchy of a particular application. To do this, the hierarchy is turned into a visitable hierarchy by letting every class implement the *Visitable* interface. Also, the generic *Visitor* interface is extended with specific visit methods for each class in the hierarchy. Finally, a single implementation of the extended visitor interface is provided in the form of a visitor combinator *Fwd*. This combinator forwards every specific visit call to a generic default

Name	Args	Description
Identity		Do nothing
Fail		Raise *VisitFailure* exception
Not	v	Fail if v succeeds, and v.v.
Sequence	v_1, v_2	Do v_1, then v_2
Choice	v_1, v_2	Try v_1, if it fails, do v_2
All	v	Apply v to all immediate children
One	v	Apply v to one immediate child
IfThenElse	c,t,f	If c succeeds, do t, otherwise do f
Try	v	*Choice(v,Identity)*
TopDown	v	*Sequence(v,All(TopDown(v)))*
BottomUp	v	*Sequence(All(BottomUp(v)),v)*
OnceTopDown	v	*Choice(v,One(OnceTopDown(v)))*
OnceBottomUp	v	*Choice(One(OnceBottomUp(v)),v)*
AllTopDown	v	*Choice(v,All(AllTopDown(v)))*
AllBottomUp	v	*Choice(All(AllBottomUp(v)),v)*

Figure 4.3: JJTraveler's library (excerpt).

visitor given to it at construction time. Concrete visitors are built by providing *Fwd* with the proper default visitor, and overriding some of its specific visit methods.

Though instantiation of JJTraveler's framework can be done manually, automated support for this is provided by a generator, called JJForester [KV01]. This generator takes a grammar as input. From this grammar, it generates a class hierarchy to represent the parse trees corresponding to the grammar, the hierarchy-specific *Visitor* and *Visitable* interfaces, and the *Fwd* combinator. In addition to framework instantiation, JJForester provides connectivity to a generalized LR parser [BSVV02].

Operations After instantiation, the application programmer can implement operations on the class hierarchy by specializing, composing, and applying visitors.

The starting point of hierarchy-specific visitors is *Fwd*. Typical default visitors provided to *Fwd* are *Identity* and *Fail*. Furthermore, *Fwd* contains a method *visitA* for every class A in the hierarchy, which can be overridden in order to construct specific visitors. As an example, an A-recognizer *IsA* (which only does not fail on A-nodes) can be obtained by an appropriate specialization of method *visitA* of *Fwd(Fail)*.

Visitors are combined by passing them as (constructor) arguments. For example, *All(IsA)* is a visitor which checks that any of the direct child nodes are of class A, and *OnceTopDown(IsA)* is a visitor checking whether a tree contains any A-node. Visitors are applied to visitable objects through the *visit* method, such as *IsA.visit(myA)* (which does nothing), or *IsA.visit(myB)* (which fails).

```
public class Sequence implements Visitor {
  Visitor v1;
  Visitor v2;
  public Sequence(Visitor v1, Visitor v2) {
    this.v1 = v1;
    this.v2 = v2;
  }
  public void visit(Visitable x) {
    v1.visit(x);
    v2.visit(x);
} }
```

Figure 4.4: The *Sequence* combinator.

```
public class Try extends Choice {
  public Try(Visitor v) {
    super(v, new Identity());
} }
```

Figure 4.5: The *Try* combinator.

4.3.2 A library of generic visitor combinators

Figure 4.3 shows high-level descriptions for an excerpt of JJTraveler's library of generic visitor combinators. A full overview of the library can be found in the online documentation of JJTraveler. Two sets of combinators can be distinguished: *basic* combinators and *defined* combinators, which can be described in terms of the basic ones as indicated in the overview. Note that some of these definitions are *recursive*.

Basic combinators Implementation of the generic visitor combinators in Java is straightforward. Figures 4.4 and 4.5 show implementations for the basic combinator *Sequence* and the defined combinator *Try*. The implementation of a basic combinator follows a few simple guidelines. Firstly, each argument of a basic combinator is modeled by a field of type *Visitor*. For *Sequence* there are two such fields. Secondly, a constructor method is provided to initialize these fields. Finally, the generic visit method is implemented in terms of invocations of the visit method of each *Visitor* field. In case of *Sequence*, these invocations are simply performed in sequence.

Defined combinators The guidelines for implementing a defined combinator are as follows. Firstly, the superclass of a defined combinator corresponds to the outermost combinator in its definition. Thus, for the *Try* combinator, the superclass is *Choice*. Secondly, a constructor method is provided that supplies the arguments of the outermost constructor in the definition as arguments to the superclass constructor method (`super`). For *Try*, the first superclass constructor argument is the argument of *Try* itself, and the second is *Identity*. The visit method is simply inherited from the superclass.

```
public class TopDownWhile extends Choice {
  public TopDownWhile(Visitor v1, Visitor v2) {
    super(null,v2);
    setArgument(1,new Sequence(v1,new All(this)));
  }
  public TopDownWhile(Visitor v) {
    this(v,new Identity());
} }
```

Figure 4.6: The $TopDownWhile$ combinator.

Recursive combinators In order to demonstrate how visitor combinators can be used to build recursive visitors with sophisticated traversal behavior, we will develop a new generic visitor combinator $TopDownWhile(v_1, v_2)$.

$$\begin{aligned} &TopDownWhile(v_1, v_2) = \\ &\quad Choice(Sequence(v_1, All(TopDownWhile(v_1, v_2))), v_2) \end{aligned}$$

The first argument v_1 represents the visitor to be applied during traversal in a top-down fashion. When, at a certain node, this visitor v_1 fails, the traversal will not continue into subtrees. Instead, the second argument v_2 will be used to visit the current node. The encoding in Java is given in Figure 4.6. Note that Java does not allow references to `this` until after the `super` constructor has been called. For this reason, the first argument, which contains the recursion, gets its value not via `super`, but via the `setArgument()` method. Note also that the visitor has a second constructor method that provides a shorthand for calling the first constructor with $Identity$ as second argument.

4.4 Attribute Grammars

In the context of the design and implementation of language-based tools, attribute grammars provide powerful properties to improve the productivity of their users, namely, the static scheduling of computations. Indeed, an attribute grammar writer is neither concerned with breaking up her/his algorithm into different traversal functions, nor is she/he concerned in conveying information between traversal functions (*i.e.*, how to pass intermediate values computed in one traversal function and used in following ones). A second important property is that circularities are statically detected. Thus, the existence of cycles, and, as a result, the non-termination of the algorithms, is detected statically. That is to say that for (ordered) attribute grammars the termination of the programs for all possible inputs is statically guaranteed.

An attribute grammar [Knu68, Kas80] consists of a *context-free grammar*, and a set of *attributes* and *attribute equations*. The context-free grammar of a language specifies the (finite) set of symbols of the alphabet, and defines which sequences of those symbols form a syntactically valid sentence. On the other hand, the set of attributes and attribute equations describe semantic properties of the language. The static semantics of the language

is specified by establishing conditions on the attributes. In other words, the context-free grammar defines the structure of the language while the attributes and their equations define the meaning of the language.

In order to present attribute grammars let us analyse in detail the well-known "*repmin*" problem. The formulation of the problem, taken from Bird [Bir84] who originally introduced it, is as follows: consider the problem of transforming a tree into a second tree, identical in shape to the original one, but with all the tip values replaced by the minimum tip value. In listing 2 we show the formulation of the *repmin* in the attribute grammar formalism. We use a standard AG notation, where semantic rules are Haskell expressions. Furthermore, this AG is organised by *aspect*, that is to say, that we structure the grammar according to the different aspects of the repmin problem: computing the minimum value (left) and passing the global minimum down to the leaves and constructing the desired tree.

```
Tree      <↑ min : Int >
Tree  =   Tip     Int
          Tree.min  =  Int
          |  Fork   Tree  Tree
          Tree1.min  =  min Tree2.min  Tree3.min
R         <↑ new : Tree >
R  =      Root   Tree
     Tree.m  =  Tree.min
     R.new   =  Tree.new
```

```
Tree      <↓ m : Int, ↑ new : Tree >
Tree  =   Tip     Int
          Tree.new  =  Tip Tree.m
          |  Fork   Tree  Tree
          Tree1.new  =   Fork   Tree2.new
                                Tree3.new
          Tree2.m    =   Tree1.m
          Tree3.m    =   Tree1.m
```

Fragment 1: The repmin attribute grammar.

From this specification, an attribute grammar based system - the Lrc system in our case [KS98] - first weaves the different aspects, then, computes an evaluation order (testing for circularities induced by the attribute equations), after that it schedules the computations and, finally, produces a functional (or non-functional) implementation [Sar99].

In the context of AG programming, the programmer focus his work in the productions (or constructors) where useful work has to be performed. In the *repmin* problem, for example, this occurs in the production Fork where the semantic function *min* and the constructor Fork is used. However, the programmer has also to include in the specification rules to propagate attributes (*i.e.*, context information) in the tree, by the so-called, copy rules (shadowed equation in 2).

The propagation of information via copy rules results in large and less legible AG specifications. Typical patterns of attribution, however, can be specified in a more concise and comprehensible notation, that makes AG specifications shorter and easier to understand [KW94, Paa95, SAS99]. For example, propagating an attribute in a top-down, left-to-right often occurs in AG specifications. AG systems [GHL+92, SAS99] include special notation for specifying such flow of data. Another technique to eliminate redundant copy rules is to include them as implicit default rules (of the AG-based system). Both of theses approaches have a serious drawback: propagation patterns are fixed and are part of the AG specification language. Thus, the AG programmer can not add his own propagation patterns nor

change the existent ones.

In Strategic Attribute Grammars we propose the use of strategies (tree traversal combinators) to model attribute propagation patterns. The idea is to define the propagation of attributes as strategic semantic functions. As a result, the AG programmer as the full expressiveness of strategic programming and can add or adapt combinators to realise existing or new forms of propagating attributes. So, the user is not restricted to a pre-defined set of propagation patterns. Because in the LRC system we define semantic functions as Haskell-expressions, we can use the Strafunski library to easily embed strategies in an AG specification. Next we show the Strategic version of the *repmin* AG.

$$
\begin{array}{lll}
Tree & <\downarrow m : Int> & \\
R & <\uparrow new : Tree> & \\
R = \textsc{Root}\ Tree & & \\
\quad Tree.m & = & crush\ (0, min, crushAction, Tree) \\
\quad R.new & = & bottomup\ (repAction\ Tree.m, Tree)
\end{array}
$$

Fragment 2: The repmin strategic attribute grammar.

As we can see in the specification, no copy rules are defined in the AG. All the propagation of attributes is done by the *crush* and *bottomup* combinators.

We need to define, however, the function that perform useful work in the nodes of the tree. They are presented next.

```
crushAction = build [] `adhocTU` f
  where f (Tip i) -> [i]
        f  _      -> []

repAction min_in = identity `adhocTP` f
  where f (Tip _) -> [Tip min_in]
        f  x      -> [x]
```

The key idea of strategic attribute grammars is to combine the advantages of both approaches, namely, the static scheduling of the traversal functions of the AG formalism and the expressiveness and genericity of the strategic programming paradigm.

Chapter 5

Design Patterns and Programming Idioms

Since strategic programming enriches the programmer's repertoire of abstraction and modularization techniques, it has given rise to a body of programming experience. This experience is laid down in design patterns and programming idioms that guide the programmer in his program construction activities. The tutorial includes a presentation of these design patterns and idioms involving typical examples and sample code. These will be taken from various programming paradigms, and may be tuned to the interests and wishes of the participants of the tutorial. The objective is to bring the participants to a point where they are ready to start programming strategically in their preferred languages.

5.1 Traversal idioms

Strategic programming is particularly useful for the construction of traversals over many-sorted data structures. Not surprisingly, quite a number of idioms for constructing such traversals can be identified. These idioms touch upon traversal issues such as:

Fix-points Repeating the traversal and / or its basic actions until some fix-point is reached.

propagation Passing information down into the traversed structure.

Accumulation Retrieving information from the traversed structure by adding pieces of information to an accumulator.

Backtracking Switching to a different part of the traverse structure, or to different traversal behavior depending on success and failure of subtraversals.

5.2 Transformation idioms

Transformations are type-preserving traversals that reshape the traversed structure, for instance for normalization, optimization, etc. Example idioms:

Cascading Combining simple node actions into a complex node action. The triggering of one of the simple actions may create opportunities for other simple actions to trigger as well.

Staging Dividing a transformation into separate steps that are executed in sequence.

Nesting Implementing an overal transformation by a traversal strategy that invokes subordinate transformations on particular substructures of the traversed structure.

5.3 Variation points

Traversals can have variation points of various kinds. When the abstraction mechanisms of the host paradigm are sufficiently powerful, these variation points can become actual parameters of library components.

Completeness Traversals can be complete, in the sense that they will visit each and every node of the traversed structure, or they can be incomplete in serveral ways. For instance, they may only walk along a spine of the traversed structure.

Order Traversals can be top-down, bottom-up, breadth-first, etc.

Cut-offs Traversals can be cut-off at certain nodes and under certain conditions. For instance, cut-off can occur at a fixed depth, at success or failure of an ingredient node action, etc.

5.4 Object-oriented

In object-oriented setting, several issues come into play.

Circularity The visited object structures can contain cycles. As a result, special care needs to be taken to ensure termination of traversals. Several solution can be chosen here, such as marking the visited graph, or accumulating history.

Sharing Even when the visited structure is not circular, its structure may not be tree-shaped, but contain shared nodes. To prevent visiting nodes or entire substructures repetitively, special care must be taken.

State The objects that make up a traversal, e.g., its ingredient visitors, can encapsulate particular state information. These can be simple counters, lookup tables, but also complete object graphs representing representing for instance the accumulated result of the traversal.

5.5 Functional

In functional programming, idioms can be identified that exploit the expressiveness of the hosting type system.

Monads Traversals can be monadic such that monadic effects can be exploited. These include state propagation, side-effects, non-determinism, etc.

Meta-schemes When traversal schemes are highly parameterized, for instance with the one-step traversal that they employ or the kind of combination of recursive traversals. In this case, we call them meta-shemes.

Type guards Type guards are basic actions that succeed or fail depending on the type of the node to which they are applied.

Chapter 6

Strategic Programming in the Large

Finally, we will discusse the use of strategic programming in the real world. We will discuss several large projects in which essential use has been made of strategic programming techniques. One of them is the developement of Cobol reverse engineering support with object-oriented strategic programming techniques [DV02]. Another is the Haskell refactoring tool, HaRe [TR03, LRT03], developed with functional strategic programming techniques. Also, the application of strategic programming for XML document processing will be explained, using the *HaQuery* language. For each of these applications we will point out how strategic programming techniques have been deployed, why they were beneficial for the success of the projects, and what strategic programming lessons were learned in the course of them.

6.1 Cobol reverse engineering

Visitor Combinators, as supported by JJForester and JJTraveler, have been used in the implementation of the program comprehension tool ControlCruiser. ControlCruiser is a program comprehension tool that uses Conditional Control Graphs (CCGs) as source model. ControlCruiser constructs such CCGs from program source texts, and subsequently visualizes them. Additionally, ControlCruiser computes a number of metrics by CCG analysis. Control Cruiser was developed by CWI and the Software Improvement Group.

Currently, ControlCruiser has a single front-end: for Cobol. On the basis of PERFORM statements, IF statements, and section and paragraph labels, ControlCruiser reconstructs procedures and their interrelationships from Cobol programs.

We will explain the overall design of ControlCruiser, and zoom in on some of the strategic programming techniques employed in its implementation. In the implementation of ControlCruiser, both tree shaped and graph shaped source models are constructed, analyzed, and manipulated with visitor combinators.

6.2 Haskell refactoring

The functional programming group at Kent University, Canterbury, UK is developing the Haskell refactoring tool HaRe in the context of their Refactoring Functional Programs project [TR03, LRT03]. HaRe's refactoring engine is implemented on top of Programatica's Haskell frontend and Strafunski's generic traversal strategy library. Most of the refactoring's offered are implemented with strategic programming techniques. According to HaRe's developers, the conciseness of strategic programming has been essential in achieving developer productivity and keeping the project manageable.

The project describes a catalogue of Haskell program refectorings, like: *Add or remove an argument*, *Delete/Add a definition*, *Introduce or remove a duplicate definition*, *Generalise or specialise a definition*, *Introduce A New Definition*, *Widen or narrow definition scope*, *Widen or narrow definition scope, with compensation (generalise/specialise)*, *Renaming*, and *Unfolding*.

6.3 XML querying

*HaQuery*is a Haskell-based combinator library for querying *Xml* documents being developed at Minho University. *HaQuery* is closely related to *XQuery*, a typed, functional language for querying XML, currently being designed by the *Xml* Query Working Group of the World-Wide Web Consortium [Dra02, Wad02].

The library consists of a (small) set of functions (*i.e.*, the combinators) that can be combined allowing us to efficiently, easily and concisely express queries over *Xml* documents. The *HaQuery* library itself is constructed in a combinatorial approach, that is to say that its parser is defined through combinator parsers and the implementation of *HaQuery* relies entirely on traversal combinators. The first prototype implementation of *HaQuery* was built on top of *HaXml*, the standard HASKELL combinator library for *Xml*. Currently we are porting *HaQuery* system to tely on the the strafunsky library. Or, to be more precise, we are extending Strafunsky with a new set of combinators: the HaQuery combinators.

To introduce *HaQuery*, let us consider some example queries on a suitable *Xml* document. We consider a document that contains the information describing a particular university course. That is, the document describes the name of the course, the year on the *curriculum*, the information concerning the teachers and the students registered in the course. To list the students registered in the course we have to write the following simple *HaQuery* sentence:

```
/curso/cadeiras/inscritos ?
```

If we wish to list the students attending the third year (or higher) or the students whose surname is *Atento* and are registered as *TE* (working students), then we might write:

```
//aluno[@ano .>=. 3 | (./nome/@sobrenome='Atento'
                       & ~(@estatuto = 'TE'))] ?
```

HaQuery allows us to elaborate more complex queries, like nesting queries and queries that are performed over a list of *Xml* documents and not just on a single one.

In the Strafunsky-based implementation of the *HaQuery* processor, *HaQuery* sentences are Haskell expressions that are defined through combinator functions that use a friendly user-defined syntax to resemble the original *HaQuery* syntax. Such combinator functions model *HaQuery* operators (like, for example, the *HaQuery* operator `/` that defines a parent/child relation, is modelled with a haskell function with exatly that name). These combinator functions are straightforwardly expressed in strafunsky.

Bibliography

[BDPS02] Gilles Barthes, Peter Dybjer, Luis Pinto, and João Saraiva, editors. *International Summer School on Applied Semantics*, volume 2395 of *LNCS Tutorial*. Springer-Verlag, August 2002.

[Bir84] R. S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, (21):239–250, January 1984.

[BSVV02] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[Dra02] W3C Working Draft. *XQuery 1.0: An XML Query Language*, April 2002.

[DV02] A. van 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.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GHL+92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM,*, 35(2):121–131, February 1992.

[GL01] B. Gramlich and S. Lucas, editors. *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume SPUPV 2359, Utrecht, The Netherlands, May 2001. Servicio de Publicaciones - Universidad Politécnica de Valencia.

[Jeu00] J. Jeuring, editor. *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, July 2000.

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[KLV00] J. Kort, R. Lämmel, and J. Visser. Functional Transformation Systems. In *9th International Workshop on Functional and Logic Programming*, Benicassim, Spain, July 2000.

[Knu68] Donald E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[KS98] Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

[KV01] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[KW94] Uwe Kastens and William Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, June 1994.

[Läm01] R. Lämmel. Generic Type-preserving Traversal Strategies. In Gramlich and Lucas [GL01].

[Läm02] R. Lämmel. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming*, 2002. To appear.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[LRT03] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In Johan Jeuring, editor, *ACM SIGPLAN 2003 Haskell Workshop*. Association for Computing Machinery, August 2003. ISBN 1-58113-758-3.

[LV00] R. Lämmel and J. Visser. Type-safe functional strategies. In *Scottish Functional Programming Workshop, Draft Proceedings*, St Andrews, 2000.

[LV02a] R. Lämmel and J. Visser. Design patterns for functional strategic programming. In *Proceedings of the international workshop on rule-based programming (RULE 2002)*, October 2002.

[LV02b] R. Lämmel and J. Visser. Typed combinators for generic traversal. In *PADL 2002: Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2002.

[LV02c] Ralf Lämmel and Joost Visser. Strategic polymorphism requires just two combinators! In *Preproceedings of IFL 2002*, September 2002.

[LV03] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[LVK00a] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In Jeuring [Jeu00], pages 46–59.

[LVK00b] R. Lämmel, J. Visser, and J. Kort. Dealing with large bananas. In Johan Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. FPCA'91*, volume 523 of *LNCS*. Springer-Verlag, 1991.

[Paa95] Jukka Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[Sar99] João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999.

[Sar02] João Saraiva. Component-based Programming for Higher-Order Attribute Grammars. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2002*, volume 2487 of *LNCS*, pages 268–282. Springer-Verlag, October 2002.

[SAS99] Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, September 1999.

[SS03] João Saraiva and Doaitse Swierstra. Generating Spreadsheet-like Tools from Strong Attribute Grammars. In Frank Pfenning and Yannis Smaradakis, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2003*, volume 2830 of *LNCS*, pages 307–323. Springer-Verlag, September 2003.

[SSK00] João Saraiva, Doaitse Swierstra, and Matthijs Kuiper. Functional Incremental Attribute Evaluation. In David Watt, editor, *9th International Conference on*

Compiler Construction, CC/ETAPS2000, volume 1781 of *LNCS*, pages 279–294. Springer-Verlag, March 2000.

[TR03] Simon Thompson and Claus Reinke. A case study in refactoring functional programs. In Roberto Ierusalimschy, Lucilia Figueiredo, and Marcio Tulio Valente, editors, *VII Brazilian Symposium on Programming Languages*, pages 1–16. Sociedade Brasileira de Computacao, May 2003.

[V+] E. Visser et al. The online survey of program transformation. `http://www.program-transformation.org/survey.html`.

[VBT99] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).

[Vis99] E. Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30 – 44. Springer-Verlag, 1999.

[Vis00] E. Visser. Language Independent Traversals for Program Transformation. In Jeuring [Jeu00], pages 86–104.

[Vis01a] E. Visser. A Survey of Strategies in Program Transformation Systems. In Gramlich and Lucas [GL01].

[Vis01b] J. Visser. Visitor combination and traversal control. In *OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 2001.

[Vis03a] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer et al., editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science. Spinger-Verlag, November 2003. (Draft; Accepted for publication).

[Vis03b] Joost Visser. *Generic Traversal over Typed Source Code Representations*. PhD thesis, University of Amsterdam, February 2003.

[Wad02] Philip Wadler. Xquery: a typed functional language for querying XML. In *Fourth Summer School on Advanced Functional Programming, Oxford*, August 2002.

Appendix A

Strategic Programming in a Nutshell

At the heart of Strategic Programming are two programming concepts: dynamic type-case, and one-step traversal. Dynamic type-case allows mixing generic behaviour with type-specific behaviour: depending on the type of input data, either the type-specific behaviour is triggered, or the generic default behaviour. One-step traversal makes generic traversal scheme's programmable: by composing the combinators in different constellations, different traversal scheme's are obtained.

To make things more concrete, consider some of the basic combinators offered by Strategic Programming:

id	return input term unchanged
sequence(f,g)	apply f to the input term, and g to the result of that
all(f)	apply f to all immediate subterms of the input term
fail	react to any input term with *failure*
choice(f,g)	apply f to the input term. If it fails, apply g instead
one(f)	apply f to a single immediate subterm of the input term
adhoc(f,g)	apply g to the input term if it's type matches, otherwise apply f
f◇t	apply strategy f to input term t

Of course, it depends on the hosting programming paradigm which combinators are available exactly, and in which form. The above list shows a paradigm-independent, almost mathematical specification of a subset of such combinators.

Given such combinators, one can for instance define the following generic traversal schemas:

```
bottomup(f)     =  sequence(all(bottomup(f),f))
topdown(f)      =  sequence(f,all(topdown(f))
oncebottomup(f) =  choice(one(oncebottomup(f),f))
innermost(f)    =  sequence(all(innermost(f)),choice(sequence(f,innermost(f)),id))
```

Now, such traversal schemes can be instantiated to particular traversals by supplying a node action as argument. For instance, with the adhoc combinator we can create a node action integerincrement, and pass it as argument to topdown:

$$\begin{array}{rcl} \mathsf{incrementinteger} & = & \mathsf{adhoc}(\mathsf{id}, \lambda x.x{+}1) \\ \mathsf{incrementintegerstopdown} & = & \mathsf{topdown}(\mathsf{incrementinteger}) \end{array}$$

When applied to an integer integerincrement will increment it (see the function $\lambda x.x{+}1$ on integers), but when applied to any other input term it will return it unchanged (see the generic identity function id). When we apply incrementintegerstopdown to input terms of different types, the genericity of strategic programming becomes evident:

$$\begin{array}{rcl} \mathsf{incrementintegerstopdown}\diamond[0,1,2] & \rightsquigarrow & [1,2,3] \\ \mathsf{incrementintegerstopdown}\diamond(True,[0,1],2) & \rightsquigarrow & (True,[1,2],3) \end{array}$$

Thus, the traversal incrementintegerstopdown works on input terms of any type. It walks over the input term in a bottom-up fashion, and increments all integers that it finds on its way.

Thus, these examples give a brief glimpse of the essentials of Strategic Programming. The concepts of dynamic type case (adhoc) and of one step traversal (all and one) play a fundamental role. In the tutorial, these concepts will be explained thoroughly, as well as their realization in different programming languages from various paradigms. Also, the practise of programming with strategies, both in the small (idioms, design patterns) and in the large (applications, tool support) will be covered.