Generic Traversal over Typed Source Code Representations

The work reported in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Generic Traversal over Typed Source Code Representations

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. mr. P. F. van der Heijden
ten overstaan van een door het
college voor promoties ingestelde commissie,
in het openbaar te verdedigen
in de Aula der Universiteit
op vrijdag 14 februari 2003, te 10.00 uur

door

Johannes Michiel Willem Visser
geboren te Middelburg, Nederland

Promotor:	prof. dr P. Klint
Co-promotor:	Dr.-Ing. R. Lämmel
Faculteit:	Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam
Kruislaan 403
1098 SJ Amsterdam

Preface

Environment is of decisive importance to the success or failure of a starting researcher. To get into a productive research and publication mode, one is helped tremendously by the challenges and examples that others set before him. At a time that my research qualities were all but evident, I was very lucky that Paul Klint gave me the opportunity to become part of his software engineering research group at CWI. The thesis that lies before you is a witness to the fact that this group has been a fruitful environment for me.

In my perception, the key axiom pervading Paul's group is that science is pursued not purely as an end in itself, but with relevance to and inspiration from the realities of software users and producers. The work presented in this thesis attempts such a blending of theory and practice. I am grateful to Paul's challenge and his support in meeting it.

Ralf Lämmel has been a great inspirational influence from the moment he arrived at CWI. His relentless drive for productivity and quality is truly amazing. Our collaboration on our first joint paper gave me that push in the right direction which I needed to get into orbit. Since then we have worked together intensively and fruitfully, as witnessed by various of the papers underlying this thesis, and by the success of the Strafunski project. On top of all this, working with Ralf is a lot of fun.

Merijn de Jonge, Leon Moonen, and I started at CWI simultaneously and we shared an office during most of our time there. Both proved to have ample expertise in areas where I was an utter ignoramus, and they were kind enough to show me some ropes. Their continuous strive to genericity (Merijn) and perfection (Leon) were a source of both short-lived frustration and enduring inspiration. Miraculously, Leon and I never published together – an omission that must some day be repaired. Merijn and I worked together on various projects, including XT, workshop organization, 'programmatuurvoeding', and arduous traveling in the interest of science.

At all times, Tobias Kuipers can be counted on for boundless optimism, outspoken opinions, and action. In projects such as HaSdf and JJForester, I was still contemplating possibilities while he just went ahead and started implementing. He

is my favorite remedy against indecisiveness.

I enjoyed working with Arie van Deursen because of his cheerful scrutiny. He never stops to ask critical questions until he is fully satisfied, while always maintaining his natural positive attitude.

Eelco Visser has been a close colleague, even though he was not part of our group at the same time as I was. I enjoyed ‘conspiring’ in the XT project with him and Merijn, and many of the ideas in this thesis have been inspired by his work.

A silent, but extremely stimulating force in the background has been my father, Wim Visser. His unconditional trust and support for whatever I try to achieve has been a factor in actually achieving it that can not be overestimated. My sisters, Esther and Lucie, have helped me keep things within perspective by making fun of me whenever I deserved it. Thanks!

Finally, I thank all members of the reading committee, Claude Kirchner, Erik Meijer, Jan Bergstra, Peter van Emde Boas, and Jan van Eijck, for their willingness to review this thesis.

Hilversum, December 2002

Contents

Chapter 1

Introduction

Languages are at the heart of computing. These include not only programming languages of numerous shapes and sizes (object-oriented, logical, functional, general-purpose, domain-specific, low level, (very) high level), but also command languages, scripting languages, query languages, configuration languages, specification languages, data formats, interface definition languages, and mark-up languages.

Software products are created by writing source code in these languages, and then having this source code processed by appropriate language processing tools, such as compilers, interpreters, configuration managers, database management systems, and code generators. Similarly, secondary software development tasks, such as program comprehension, reverse engineering, quality assessment and software renovation, are supported by language processing tools such as documentation generators, renovation factories, refactoring tools, and testing tools. Thus, computer languages are more than a means of expression and communication for software developers. They also form the interface to the software developer's tools. From the perspective of these tools, the expressions of computer languages are *data* to be processed.

Software development tools are themselves software products that need to be developed. This thesis focuses on providing support for such tool development, in particular for tasks that are common to and at the core of all language processing tools: *creating* representations of source code, and *traversing* these representations to analyze them, modify them, or generate new representations from them. The prime objective of this thesis is to demonstrate that traversal of these representations can be done in a *generic* manner, whilst their well-formedness is guaranteed by a strong type system.

1.1 Areas of language processing

We briefly review some areas of language processing, their scope and aim. We make an inventory of the source code representations employed in these areas, and the typical traversal scenarios that occur in them.

Language implementation

A compiler implements the operational semantics of a programming language by translating source code to expressions in a target language [ASU86]. This target language can be the instruction set of a particular execution platform, or it can be an intermediate language which in its turn needs to be compiled.

In the first phase of compilation, the source code is parsed and turned into an abstract syntax tree (AST). The target code generated in the last phase of a compiler may in turn be represented by an AST. Sometimes, tree-shaped or graph-shaped intermediate representations (IRs) are used between translation steps. Between parsing and code generation, various static checks may be performed, such as type checks and initialization checks. Another phase that may precede translation is desugaring or normalization.

Optimizing compilers perform sophisticated analyses to be able to reduce the number of instructions that are generated, the memory or time consumption of the generated program, or to improve other properties. Such analyses include data flow analysis, control flow analysis, and liveness analysis. Typically, various kinds of dependency graphs are constructed during these analyses. The results of these analyses are often used to steer subsequent transformations, such as inlining and deforestation.

An interpreter, like a compiler, consumes source code, but implements operational semantics in a different way. Not by translation to a target language, but by executing target instructions on the execution platform directly. The parsing, checking, and desugaring phases of a compiler, including the source representations involved in them, may also be found in interpreters. The actual interpretation phase itself is a traversal of a source code representation that is programmed *in* the target language, i.e. in a language that runs on the execution platform.

Reverse Engineering

Reverse engineering [CC90] aims at creating representations of a software system, its components, and their interrelationships at a higher level of abstraction. This includes activities such as decompilation (reconstruct source code from object code), architecture extraction (reconstruct design from implementation), and documentation generation (extract APIs, textual and graphical overviews, indexes). The ultimate goal of reverse engineering can be (interactive) program comprehension, impact analysis, quality assessment, re-implementation, or migration.

Often, reverse engineering only concerns certain aspects of the source code, because the particular higher level model that is to be constructed abstracts over other aspects. For example, in architecture extraction for Java, one is usually not interested in the bodies of methods, but only in their signatures and the call relations among them. As a consequence, reverse engineering tools may not perform a full syntactic analysis, but opt for selective parsing with an island grammar [DK99a, Moo02], or for lexical analysis. In such cases, the initial source representation is not a fully detailed AST, but a rather trimmed-down AST, or simply a table.

Other source code representations employed in reverse engineering include module graphs, conditional call graphs (see Chapter 7), concept lattices [Sne00, DK99b], and document trees.

Generative Programming

The objective of generative programming [CE99] is the construction of programs by automating the construction and configuration of components. Generative programming is, in some sense, directed in the opposite sense of reverse engineering, as it involves generation of actual programs from higher-level specifications of such programs. The effect of using generative programming is that the level of abstraction at which the programmer works is raised from the solution domain to the problem domain.

Three kinds of computer languages play a central role in generative programming: domain-specific languages, template programming languages, and configuration languages.

Domain-specific languages (DSLs) or 'little languages' are executable specification languages that provide expressive power focused on a particular application domain [DKV00]. In generative programming, DSLs are used to give high-level specifications of software components. DSL compilers (also called application generators [Cle88]) generate implementations in general-purpose programming languages from DSL programs. Examples of domains for which DSLs have been developed include digital hardware design [JB99], financial products [B$^+$96], and telecommunications [LR94].

One of the implementation techniques for DSL compilers, commonly used in generative programming is *template programming*. A template language is an extension to a general-purpose programming language that allows the programmers to generate base language code at, or immediately preceding, compilation time. A well-known example is the template programming facility of C++.

Both generated and hand-crafted components must be configured into a final software product. To automate such configuration, configuration languages can be used. These are formats or little languages in which the configuration of a system can be described, usually at the problem level. Examples of configuration languages include the Feature Description Language [DK02], and the autobundle package description language [Jon02b].

Source code representations that may play a role in generative programming are the ASTs of DSL programs and of generated code, as well as representations of configuration spaces and component dependencies. DSL compilers and transformation-based generators implement similar traversal scenarios as compilers for general-purpose languages. Traversal scenarios on representations of configurations and component dependencies include computation of transitive closures or transitive reductions, and normalization.

Software Renovation

The aim of software renovation [DKV99] (also known as re-engineering [DV02a]) is to automatically carry out modifications on a complete software system such that errors are removed (corrective maintenance), or additional or different requirements are met (perfective or adaptive maintenance). Such modifications can range from minor changes (e.g. bug fixes) to structural change (e.g. re-modularization, goto-elimination).

Software renovation bears similarity to reverse engineering in the sense that it is usually only concerned with certain aspects of the source code. A difference is that in software renovation the end product is of the same abstraction level as the initial source, and therefore the aspects of the code that are not *changed* still need to be *preserved*. These include aspects such as comments and layout. The consequence of this is that software renovation, like reverse engineering, may use selective analysis techniques, such as parsing with island grammars, or lexical analysis, but the source code representations that are constructed still need to contain all non-relevant parts of the source code in an unanalyzed form. This means that the ASTs still need to contain 'water', i.e. strings of unparsed code. Another technique is to keep detailed information in the AST about the origin of each node, and to perform the changes, not on the AST itself, but directly on the source. One may also decide to use parse trees (concrete syntax trees) that contain all information about the source (including lexicals, layout, and comments), and to implement traversals on these. To regain space-efficiency, compression techniques such as hash-consing [AG93, BJKO00] may be used, and traversal will take place on compressed trees.

Whether ASTs with water or origins are used, or full parse trees, the traversal scenarios to be implemented are basically the same. The trees must be analyzed to determine which changes need to be made, and subsequently they must be transformed accordingly. Finally, the representation of the renovated source code must be unparsed or pretty-printed in a conservative fashion (i.e. with preservation of layout and comments [BV96, Jon02a]). As in the case of compilation, additional analyses involving dependency graphs may be needed as well.

Document processing

Mark-up languages, most notably HTML and XML [BPSM98], are intended to represent and exchange semi-structured information in documents that contain not only text, but also markers that lend structure to the document. Just like program source text, marked-up documents can be parsed to construct ASTs. Such ASTs may contain large portions of unanalyzed text.

Document processing may be aimed at retrieving information from a document, transforming a document, or translating it to another format. XML documents are typically used to hold information that can be presented in different forms by applying different document processors that translate to HTML. Also, marked-up documents can be used as exchange format in electronic data interchange (EDI).

Representations and traversal scenarios in language processing

Thus, the source code representations that are used throughout these areas of language processing are syntax *trees* (abstract, concrete, with and without portions of unanalyzed text), dependency *graphs* (data flow, control flow, import structure), and *tables* with metrics and other properties.

The traversals over these source code representations can be categorized as *translations* (compilation, reverse engineering, type inference, pretty-printing, flow analysis), *rephrasings* (normalization, desugaring, renovation), and *analyses* (type checking, unparsing, computation of metrics). Here we adopt the terminology of the program transformation taxonomies in [JVV01, V^+], where a translation is a traversal that generates a representation of a different type, a rephrasing is a traversal that produces a modified representation of the same type, and an analysis is a traversal that derives properties or values. Note that, following this taxonomy, traversals such as type inference and flow analysis are categorized as translations rather than analyses, because their results are highly structured and can themselves be viewed as (trimmed-down) source code representations.

1.2 The role of types

As we have seen, the source code representations involved in different areas of language processing are usually highly heterogeneous data structures. An abstract syntax tree, for instance, is a term over the many-sorted signature that corresponds to the abstract syntax of the input language. For widely used languages such as Java and XML, the signature contains about 100 sorts and several hundreds of productions. Grammars for (dialects of) the legacy language Cobol contain about

200 sorts and about 600 productions.[1] For smaller languages and formats, such as DSLs, syntax definition formats, graph representation formats, and island grammars, these numbers are usually lower, but around 20 sorts and several dozens of productions is not uncommon.

Likewise, graph-shaped source code representations, such a data-flow graphs and conditional call graphs, are usually heterogeneous. For instance, Control-Cruiser (to be discussed in Chapter 7) represents Cobol control flow with a conditional call graph that contains 5 concrete and 5 abstract node types. The exchange format FAMIX, used within the FAMOOS re-engineering project for exchange of object-oriented source code, consists of 22 types [DTS99].

In case of document processing, the structure of a document is dictated by a document format. In the special case of XML, a distinction is made between well-formedness and validity. A well-formed document adheres to the general XML format. A valid document additionally adheres to a given document type definition (DTD), or 'schema'. A close correspondence exists between document schemas and many-sorted signatures: roughly, 'elements' correspond to sorts, and their alternatives correspond to productions (see [MLM01] for a more in depth discussion).

When processing heterogeneous data structures, the use of a programming language with a strong type system can bring various benefits. Firstly, by giving strong types to the elements of the data structures, the programs that operate on them are guaranteed to preserve their well-formedness (as far as the expressiveness of the type system goes). Ill-formed input will be rejected, and well-formed output is guaranteed. Secondly, the programs themselves will be guaranteed to be well-formed. Any error in a pattern-match, a data component selection, a data construction, or other manipulation will be discovered and reported at compilation time. Secondly, types abstract over a piece of functionality and therefore can be used to describe its interface. This is useful for encapsulation, program understanding, and it can form the basis for generated documentation. The method headers in Java, for instance, are used to form the interface of a class as well as the API of an entire application, and they are presented in browsable form by the *javadoc* documentation generator.

In various programming language paradigms, heterogeneous data structures, such as source code representations, are given types in different ways. In object-oriented programming, a class-hierarchy provides the types. In term rewriting a first-order many-sorted signature provides the types. In functional programming a set of algebraic datatypes serves this role. When a strongly typed language from one of these paradigms is used for language processing, the abovementioned benefits can be enjoyed.

But, when the aim is to program traversals, strong type systems may also entail some disadvantages, as we will explain below.

[1] These figures are based on the SDF grammars of these languages in the online grammar base [GB].

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

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

Figure 1.1: Abstract syntax of EBNF.

1.3 Traditional typeful approaches to traversal

Let's consider some of the consequences 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 1.1 shows an abstract syntax 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 1.2. The main difference with the tree grammar of Figure 1.1 is that the iteration of productions ($Prod\ ^*$) has been expanded into the sort $Prods$. Solutions to our two example problems are shown in Figure 1.3. 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 $S \rightarrow NonTermSet$ for each (non-lexical) sort S. Here we assume that a sort $NonTermSet$ for sets of non-terminals has been previously 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

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

Figure 1.2: First-order signature that represents the abstract syntax of EBNF.

subterms except those of type $\mathit{NonTerminal}$. 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 $\mathit{Opt}(re)$ to $\mathit{Alt}(re, \mathit{Empty})$. 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 $norm_N(\mathit{Opt}(re))$, a rule is added that reduces to $\mathit{Alt}(norm_N(re), \mathit{Empty})$. 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.

Collection of non-terminals in ‘functional’ rewriting style:

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

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

Normalization of optionals in ‘pure’ rewriting style.

$$Opt(re) \leadsto Alt(re, Empty)$$

Normalization of optionals in ‘functional’ rewriting style.

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

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

Figure 1.3: Implementations of EBNF operations in term rewriting.

```
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 1.4: Haskell datatypes that represent the abstract syntax of EBNF.

Functional programming

In functional programming, the abstract syntax of EBNF would be represented by a set of algebraic datatypes. This is shown in Figure 1.4. 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 1.5. 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 1.6. 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 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 1.7. 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

Collection of non-terminals

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

Normalization of optionals

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

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

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

Figure 1.5: Haskell implementation of example problems.

Fold algebra for EBNF:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathbf{data}\ Alg_{EBNF}\ g\ ps\ p\ re} \\
\quad = Alg_{EBNF}\{ & grammar & :: & NonTerminal \to ps \to g, \\
 & prodsnil & :: & ps, \\
 & prodscons & :: & p \to ps \to ps, \\
 & prod & :: & NonTerminal \to re \to p, \\
 & t & :: & Terminal \to re, \\
 & n & :: & NonTerminal \to re, \\
 & e & :: & re, \\
 & star & :: & re \to re, \\
 & plus & :: & re \to re, \\
 & opt & :: & re \to re, \\
 & seq & :: & re \to re \to re, \\
 & alt & :: & re \to re \to re\}
\end{array}
$$

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 1.6: Haskell implementation of the generalized fold for EBNF.

Collection of non-terminals:

$$
\begin{array}{l}
\mathit{coll} :: \mathit{Grammar} \to [\mathit{NonTerminal}]\\
\mathit{coll} = \mathit{fold}_{\mathit{Grammar}}\ \mathit{alg}_{\mathit{coll}}\\
\mathit{alg}_{\mathit{coll}} :: \mathit{Alg}_{\mathit{EBNF}}\ [\mathit{NonTerminal}]\ [\mathit{NonTerminal}]\ [\mathit{NonTerminal}]\ [\mathit{NonTerminal}]\\
\mathit{alg}_{\mathit{coll}} = \mathit{Alg}_{\mathit{EBNF}}\{ \begin{array}[t]{lcl}
\mathit{grammar} & = & \lambda \mathit{nt}\ \mathit{ps} \to [\mathit{nt}] + \!\!+\ \mathit{ps},\\
\mathit{prodsnil} & = & [\,],\\
\mathit{prodscons} & = & \lambda p\ \mathit{ps} \to p + \!\!+\ \mathit{ps},\\
\mathit{prod} & = & \lambda \mathit{nt}\ \mathit{re} \to [\mathit{nt}] + \!\!+\ \mathit{re},\\
t & = & \lambda t \to [\,],\\
n & = & \lambda \mathit{nt} \to [\mathit{nt}],\\
e & = & [\,],\\
\mathit{star} & = & \lambda \mathit{re} \to \mathit{re},\\
\mathit{plus} & = & \lambda \mathit{re} \to \mathit{re},\\
\mathit{opt} & = & \lambda \mathit{re} \to \mathit{re},\\
\mathit{seq} & = & \lambda \mathit{re}_1\ \mathit{re}_2 \to \mathit{re}_1 + \!\!+\ \mathit{re}_2,\\
\mathit{alt} & = & \lambda \mathit{re}_1\ \mathit{re}_2 \to \mathit{re}_1 + \!\!+\ \mathit{re}_2\}
\end{array}
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{norm} :: \mathit{Grammar} \to \mathit{Grammar}\\
\mathit{norm} = \mathit{fold}_{\mathit{Grammar}}\ \mathit{alg}_{\mathit{norm}}\\
\mathit{alg}_{\mathit{norm}} :: \mathit{Alg}_{\mathit{EBNF}}\ \mathit{Grammar}\ [\mathit{Prod}]\ \mathit{Prod}\ \mathit{RegExp}\\
\mathit{alg}_{\mathit{norm}} = \mathit{Alg}_{\mathit{EBNF}}\{ \begin{array}[t]{lcl}
\mathit{grammar} & = & \mathit{Grammar},\\
\mathit{prodsnil} & = & [\,],\\
\mathit{prodscons} & = & (:),\\
\mathit{prod} & = & \mathit{Prod},\\
t & = & T,\\
n & = & N,\\
e & = & \mathit{Empty},\\
\mathit{star} & = & \mathit{Star},\\
\mathit{plus} & = & \mathit{Plus},\\
\mathit{opt} & = & \lambda \mathit{re} \to \mathit{Alt}\ \mathit{re}\ \mathit{Empty},\\
\mathit{seq} & = & \mathit{Seq},\\
\mathit{alt} & = & \mathit{Alt}\}
\end{array}
\end{array}
$$

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.

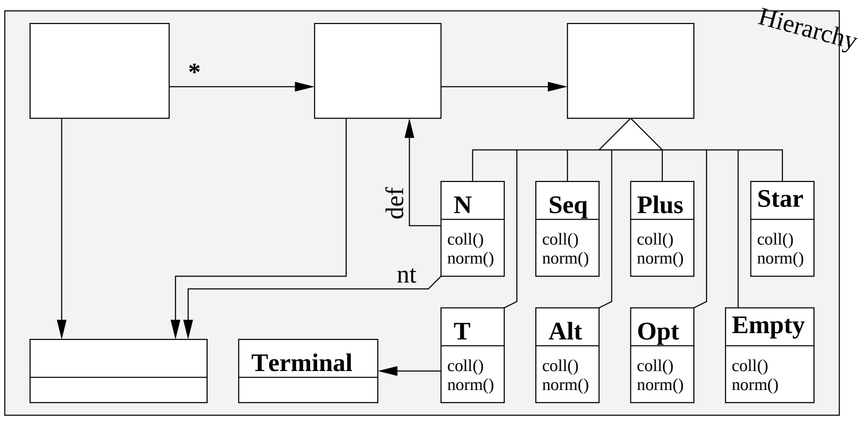
Figure 1.7: Haskell implementation of example problems, using folds.

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 Chapter 3).

Object-oriented programming

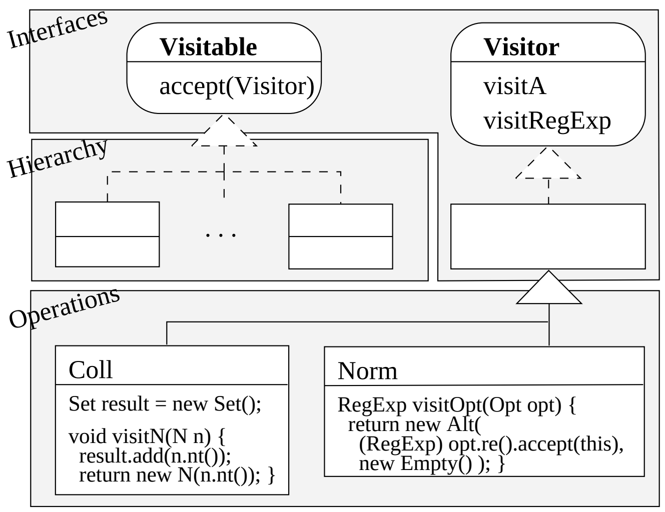
In class-based object-oriented programming, the abstract syntax of EBNF can be represented with a class hierarchy, as shown in Figure 1.8. The most straightforward approach to implementing operations (i) and (ii) is by adding corresponding methods to each of the classes in the hierarchy. For each class 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.



```
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 1.8: 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.



```
Set result = new Set();

void visitN(N n) {
  result.add(n.nt());
  return new N(n.nt()); }
```

```
RegExp visitOpt(Opt opt) {
  return new Alt(
    (RegExp) opt.re().accept(this),
    new Empty() ); }
```

```
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 1.9: 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.

They mostly make recursive method calls on their components, and only the bodies of $N.coll()$ and $Opt.norm()$ implement 'interesting' behavior. Figure 1.8 shows the implementation of these two methods in Java. Here, 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 1.9. 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.

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.

1.4 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. We will take up these disadvantages as challenges to be met by the techniques for typeful generic traversal presented in this thesis.

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.

In the functional style of rewriting, the functional programming approach without folds, and the object-oriented approach without visitors, this means that for each new traversal problem, new function symbols, functions, or methods need to be introduced for all types.

The functional approach with folds allows some reuse between traversals, but without gaining much conciseness. The recursion behavior captured by the fold function is reusable, but needs to be instantiated over and over again with functions for all types. Also, a different lengthy fold function is needed for every system of datatypes.

The visitor approach is an exception. Here, the same lengthy encoding of traversal behavior is needed. But at least this behavior can be encapsulated in a single visitor class, after which particular traversals can be implemented succinctly as subclasses that refine only a limited number of visit methods. However, when different default traversal behavior is needed, or when a different class hierarchy is employed, a new, lengthy visitor class must be constructed again.

If conciseness would be realized also for typed traversals, this would significantly reduce 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? In the functional style of rewriting, the functional approach without folds, and the object-oriented approach without visitors, this is impossible. The new traversal must be implemented from scratch. In the fold approach, the fold function can of course be reused, and the argument functions for collecting terminals and non-terminals can be composed, but the lengthy instantiation of the fold function must be repeated. In the visitor approach, this simple form of composition is possible, but only if multiple inheritance is available. More complex forms of composition can not be realized even with multiple inheritance.

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 composition. In the functional style of rewriting, the functional approach without folds, and the object-oriented approach without visitors there is no separation between traversal schema and node actions at all. In the pure style of rewriting, the node actions are captured in separate rewrite rules, but the navigation behavior is *implicit* in the strategy of the rewrite engine. Therefore, the rules can not be used in separation from the navigation. In the fold approach, the recursion behavior is factored-out, and parameterized with node actions, but these node actions can not be used to instantiate other traversal schemes than the one captured by the fold function. Finally, the visitor approach achieves some separation, but node actions (implemented as visitor refinements) can not be used independently of the traversal visitor they refine.

Ideally, a high degree of composability would be realized where new traversals can conveniently be composed by combining and refining given functionality in a combinatorial style. Thus, we would like to adopt the style of typeful programming available with function combinators and rewrite strategy combinators [KKV95], and apply it to traversal problems. This would allow 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.

It would be desirable to offer powerful means of traversal control, where programmers can concisely construct the traversal strategies that their applications require. An elegant and effective means of achieving this is to take inspiration from the (untyped) Stratego language [VBT99], which deconstructs traversal into one-step traversal combinators and ordinary recursion.

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.

If typed traversals would be defined in a (largely) generic fashion, they would be more robust against changes in source code representations. Furthermore, if the non-generic parts could be properly separated form the generic parts, the latter could be reused across different source code representations. This would open the door to the construction of libraries of reusable traversal components.

Thus, when using traditional approaches, typeful programming of traversals is at odds with conciseness, composability, traversal control, and robustness. Access to and traversal over subelements of typed representations involves dealing with many specific types in specific ways. As a consequence, type-safety comes at the cost of lengthy traversal code, which can not be reused to process different parts of the representation or for differently typed representations, and which breaks with any change in the representation type. This is the dilemma that this thesis seeks to escape.

1.5 Limitations of novel typeful approaches

In various programming paradigms, new techniques have been invented that allow a more generic treatment of traversals. To some extent, these approaches alleviate the problems of typeful traversal. We will discuss them and indicate what is still lacking.

Traversal functions

The term rewriting language ASF has been extended with *traversal functions* [BKV02]. This means that when appropriate annotations are added to a function symbol, the programmer is relieved from providing the tedious implementation of function symbols for all types in the signature. He only needs to provide declarations and rules for the types at which non-default behavior is required.

The traversal functions effectively eliminate the problem of loss of conciseness of the functional style of rewriting. Also, robustness against representation changes is realized. Our two example problems, for instance, can be implemented with traversal functions in just a few lines. Unfortunately, the approach is limited with respect to traversal control and composability. The repertoire of possible traversal schemes is fixed. The programmer is not enabled to construct his own traversal schemes, but is rather forced to encode the desired scheme in terms of the fixed set. For instance, to retrieve only a *single* non-terminal from an EBNF grammar, an accumulating topdown traversal function would need to be used that, when encountering a non-terminal, continues to traverse peer subtrees but ignores any further non-terminals that it might find. This leads again to a loss of conciseness, but also to non-intuitive encodings or unwarranted performance complexity, though recently support for additional directives, such as *break* and *continue*, has been added to alleviate these problems.

As the fold and visitor approaches, the traversal function approach allows traversal schemes to be instantiated with different node actions, but not vice versa. A node action is always implemented as a member of a family of traversal functions that follows a particular traversal strategy. In some cases, part of the traversal strategy is entangled in the node actions, in the form of recursive calls that restart the traversal when needed. Also, traversals can not be composed from reusable traversal ingredients such as one-step traversal combinators.

Polytypic programming

Polytypic programming [Mee96, JJ97a] extends the functional programming paradigm with a means of defining functions by induction over a sums-of-products representation type. Such functions are generic, since any type can be represented by sums of products. For specific types, additional equations can be provided in a polytypic function definition to provide non-generic behavior. At compile-time, a polytypic function definition is expanded to specialized functions for all encountered types.

Polytypic programming makes concise and robust implementation of traversals possible. Unfortunately, the approach is limited with respect to composability. For instance, a traversal *scheme* can not be defined separately from node actions, because polytypic functions are not first-class. One polytypic function can not be passed as argument to another. But the argument of a traversal scheme needs to be a polytypic function to enable application of the node action to more than a single node type.

A recent implementation of polytypic programming, *Generic Haskell* [CL02], complements induction over sums-of-products representation types with the additional notions of *copy lines*, *constructor cases*, and *generic abstractions*. These are inspired by the expressiveness of updatable fold algebras, as will be presented in Chapter 3. Though the composability of polytypic functions is to some extent improved with these additional notions, they still fail to be first-class citizens and hence are limited regarding composability.

Adaptive programming

In the Demeter project [LPS97], a notion of traversal strategy has been introduced for object-oriented programming. This notion of strategy should not be confused with the one from term rewriting in general or Stratego in particular. Demeter's strategies are high-level descriptions of paths through object structures in terms of start nodes, intermediate nodes, and end nodes. From these high-level descriptions, traversal code is generated.

The Demeter project has succeeded in making traversals more robust against changes in the class hierarchy, i.e. in making object-oriented software more adaptive. The approach is limited in composability, traversal control, and reusability. Demeter's strategies are never *fully* generic: though they define traversals in terms of only a *few* types, they do not allow traversals to be defined independent of *any* particular type.

1.6 Research questions

The prime objective of this thesis is to demonstrate that traversal of source code representations can be done in a generic manner, whilst their well-formedness is guaranteed by a strong type system. But this objective is not pursued in the sterile environment of theoretical study. Rather, we take the pragmatic viewpoint that the theoretical solutions need to be brought to a larger audience by proposing worked-out, light-weight, practically viable support for these solutions in mainstream general purpose programming. Only through such efforts can one entertain the hope that the potential benefits of typed generic traversal will actually be realized. The concrete research questions that this thesis aims to answer are:

1. Can traversal over source code representations be both generic and strongly typed?

2. Can typed generic traversal be supported within the context of general-purpose, mainstream programming languages?

3. Can typed generic traversal support be integrated with support for other common language tool development tasks?

Typed generic traversal As we have seen, traditional approaches to typed traversal lack any constructs for generic traversal. As a consequence, lengthy traversal code is needed, composition of complex traversals from smaller building blocks is hardly possible, reuse within applications and across applications is hindered, and traversal code is brittle with respect to changes in source code representations.

In various programming paradigms novel techniques have been proposed that allow some form of typed generic traversal. These approaches regain conciseness of traversal code and robustness, but unfortunately, they fail to address the issues of composability, traversal control, or reuse across source code representations.

Our objective will be to provide support for generic traversal that improves over these approaches in a few essential ways. We aim to take a combinatorial approach to traversal construction, where generic traversal combinators are first-class citizens that allow amalgamation of generic and type-specific behavior. Success of our approach will be measured by the amount of conciseness, composability, traversal control, and robustness that can be achieved with it.

Mainstream programming The need for generic traversal support stems from application areas such as compiler construction, software renovation, reverse engineering, generative programming, and document processing. To build competitive applications in these areas, one may need support for a wide range of technologies, such as database access, interoperability, multi-threading, and graphical user interfaces. For this reason, it is preferable to add generic traversal techniques to existing mainstream general-purpose programming languages, rather than to offer a dedicated niche-language with generic traversal support. That would allow leveraging the expressiveness of these mainstream languages, as well as the libraries and tool support that have been developed for them, and to make use of the deployment expertise gathered by an extensive user community.

We will direct our efforts at appropriate representatives from the object-oriented and functional programming paradigms. In particular, we will attempt adding generic traversal support to the class-based object-oriented programming language Java, and the non-strict strongly-typed functional programming language Haskell.

The mainstream character of Java needs no corroboration. Though no functional language can at the present time be called genuinely mainstream, Haskell comes as close as any other strongly typed functional language (SML would also have been a good candidate). It is supported by several compilers and interpreters, it has a significantly large user community, and libraries and tools are available that address issues such as database access, concurrency, interoperability, graphical users interfaces, and more [Rei02].

Integrated language tool development Traversal of source code representations is only one out of several tasks that are common to all language processing tools. Other essential tasks are *parsing* to create representations, and *pretty-printing* to convert representations back to code. When language tool development is done in a component-based fashion, another important task is *exchange* of source code representations via appropriate exchange formats.

We intend to integrate our support for generic traversal with support for parsing, pretty-printing and exchange of source code representations. Through such integration, generic traversal support should be usable for component-based development of complete language tools. In particular, we aim for integration with the language tool components developed in the context of the ASF+SDF Meta-Environment [BDH+01].

1.7 Road map

Chapter 2 presents a general architecture for language processing tools. In this architecture, SDF grammars are used as contracts between tool components. From these grammars, one can generate parsers, pretty-printers, and traversal support as well as the necessary code for representing and exchanging syntax trees between parsers, traversal components, and pretty-printers. Instantiations of this architecture are sketched for various implementation languages. In the subsequent chapters, the most challenging elements of the architecture instantiations are worked out for representative typed languages from the functional and object-oriented programming paradigms, *viz* Haskell and Java.

In Chapters 3 and 4, generic traversal support is developed for the strongly typed functional programming language Haskell, following two approaches. The first approach is more 'conventional' from the perspective of the functional paradigm, since it is based on the notion of *generalized folds,* which is well-established in this paradigm. We make these folds *updatable* and *composable*. The second approach is more flexible and powerful. It constitutes a realization in Haskell of the strategic programming idiom, of which Stratego [VBT99] and (an extension of) the Rewriting Calculus [CK99] provide earlier, but untyped, incarnations.

In Chapters 5, 6, and 7, generic traversal support is developed for the strongly typed object-oriented language Java. Also, integration is realized of traversal components developed in Java with SDF tools for parsing and pretty printing. In this paradigm, the notion of a *visitor combinator* is introduced to realize the idiom of strategic programming.

Finally, Chapter 8 discusses how our research questions are met by the material of the various chapters.

Origins of the Chapters

Chapter 2, "Grammars as Contracts", was co-authored by Merijn de Jonge. It was published earlier as:

> M. de Jonge and J. Visser. Grammars as Contracts. In *Proceedings of the Second International Conference on Generative and Component-based Software Engineering (GCSE 2000)*. Lecture Notes in Computer Science 2177, pages 85–99. Springer, 2000.

Chapter 3, "Dealing with Large Bananas", was co-authored by Ralf Lämmel and Jan Kort. It was published earlier as:

> R. Lämmel, J. Visser and J. Kort. Dealing with Large Bananas. In *Proceedings of the second Workshop on Generic Programming (WGP 2000)*. Technical Report UU-CS-2000-19, Universiteit Utrecht.

Chapter 4, "Typed Combinators for Generic Traversal", was co-authored by Ralf Lämmel. It was published earlier as:

> R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proceedings of the Fourth International Symposium on Practical Aspects of Declarative Languages (PADL 2002*). In Lecture Notes in Computer Science 2257, pages 137–154. Springer, 2002.

Chapter 5, "Visitor Combination and Traversal Control", was published earlier as:

> J. Visser. Visitor Combination and Traversal Control. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2001)*. ACM SIGPLAN Notices (36)11, pages 270–282. ACM 2001.

Chapter 6, "Object-Oriented Tree Traversal with JJForester", was co-authored by Tobias Kuipers. It was published earlier as:

> T. Kuipers and J. Visser. Object-oriented Tree Traversal with JJ-Forester. In *Proceedings of the First Workshop on Language Descriptions, Tools and Applications 2001 (LDTA'01)*. Electronic Notes in Theoretical Computer Science 44(2). Elsevier Science Publishers, 2001. To appear also in Science of Computer Programming.

Chapter 7, "Building Program Understanding Tools Using Visitor Combinators", was co-authored by Arie van Deursen. It was published earlier as:

> A. van Deursen and J. Visser. Building Program Understanding Tools Using Visitor Combinators. In *Proceedings of the Tenth International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society, pages 137–146.

Chapter 2

Grammars as Contracts

This chapter presents a general architecture for component-based development of language processing tools. It demonstrates how traversal components can be integrated with components for parsing, pretty-printing, and data-exchange. Thus, the architecture of this chapter provides a context for the generic traversal techniques to be presented in the upcoming chapters.

Component-based development of language tools stands in need of meta-tool support. This support can be offered by generation of code – libraries or full-fledged components – from syntax definitions. We develop a comprehensive architecture for such syntax-driven meta-tooling in which grammars serve as contracts between components. This architecture addresses exchange and processing both of full parse trees and of abstract syntax trees, and it caters for the integration of generated parse and pretty-print components with tree processing components.

We discuss an instantiation of the architecture for the syntax definition formalism SDF, integrating both existing and newly developed meta-tools that support SDF. The ATerm format is adopted as exchange format. This instantiation gives special attention to adaptability, scalability, reusability, and maintainability issues surrounding language tool development.

This chapter is based on [JV00].

2.1 Introduction

A need exists for meta-tools supporting component-based construction of language tools. Language-oriented software engineering areas such as development of domain-specific languages (DSLs), language engineering, and automatic software renovation (ASR) pose challenges to tool-developers with respect to adaptability, scalability, and maintainability of the tool development process. These

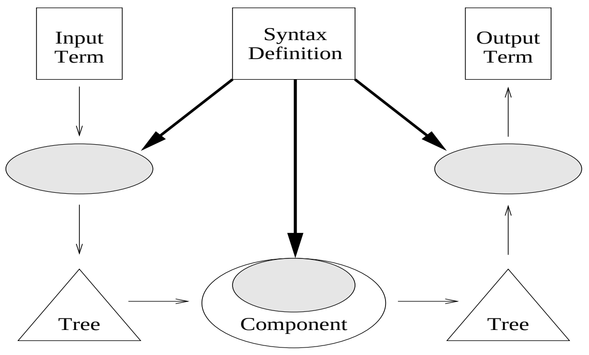


Figure 2.1: Architecture for meta-tool support for component based language tool development. Bold arrows are meta-tools. Grey ellipses are generated code.

challenges call for methods and tools that facilitate reuse. One such method is component-based construction of language tools, and this method needs to be supported by appropriate meta-tooling to be viable.

Component-based construction of language tools can be supported by meta-tools that generate code – subroutine libraries or full-fledged components – from syntax definitions. Figure 2.1 shows a global architecture for such meta-tooling. The bold arrows depict meta-tools, and the grey ellipses depict generated code. From a syntax definition, a parse component and a pretty-print component are generated that take input terms into trees and vice versa. From the same syntax definition a library is generated for each supported programming language, which is imported by components that operate on these trees. One such component is depicted at the bottom of the picture (more would clutter the picture). Several of these components, possibly developed in different programming languages can interoperate seamlessly, since the imported exchange code is generated from the same syntax definition.

In this chapter, we will refine the global architecture of Figure 2.1 into a comprehensive architecture for syntax-driven meta-tooling. This architecture embodies the idea that grammars can serve as contracts governing all exchange of syntax trees between components and that representation and exchange of these trees should be supported by a common exchange format. An instantiation of this architecture is available as part of the Transformation Tools package XT [JVV01]. The architecture is also instantiated by the tool JJForester, which will be the subject of Chapter 6.

The chapter is structured as follows. In Sections 2.2, 2.3, and 2.4 we will develop several perspectives on the architecture. For each perspective we will make an inventory of meta-languages and meta-tools and formulate requirements on these languages and tools. We will discuss how we instantiated this architec-

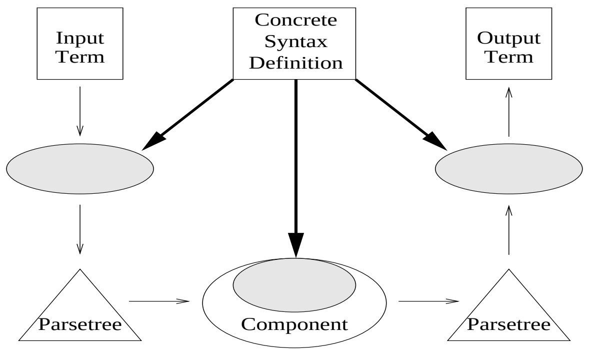


Figure 2.2: Architecture for *concrete syntax* meta-tools. The concrete syntax definition serves as contract between components. Components that import generated library code interoperate with each other and with generated parsers and pretty-printers by exchanging parse trees adhering to the contractual grammar.

ture: by adopting or developing specific languages and tools meeting these requirements. In Section 2.5 we will combine the various perspectives thus developed into a comprehensive architecture. Applications of the presented meta-tooling will be described in Section 2.6. Sections 2.7, and 2.8 contain a discussion of related work and a summary of our contributions.

2.2 Concrete syntax definition and meta-tooling

One aspect of meta-tooling for component based language tool development concerns the generation of code from *concrete* syntax definitions (grammars). Figure 2.2 shows the basic architecture of such tooling. Given a concrete syntax definition, parse and pretty-print components are generated by a parser generator and a pretty-printer generator, respectively. Furthermore, library code is generated, which is imported by tool components (Figure 2.2 shows no more than a single component to prevent clutter). These components use the generated library code to represent parse trees (i.e. *concrete* syntax trees), read, process, and write them. Thus, the grammar serves as an interface description for these components, since it describes the form of the trees that are exchanged.

A key feature of this approach is that meta-tools such as pretty-printer and parser generators are assumed to operate on the same input grammar. The reason for this is that having multiple grammars for these purposes introduces enormous maintenance costs in application areas with large, rapidly changing grammars. A grammar serving as interface definition enables smooth interoperation between parse components, pretty-print components and tree processing components. In fact, we want grammars to serve as contracts governing all exchange of trees between components, and having several contracts specifying the same agreement is a recipe for disagreement.

Note that our architecture deviates from existing meta-tools in the respect that we assume full parse trees can be produced by parsers and consumed by pretty-printers, not just abstract syntax trees (ASTs). These parse trees contain not only semantically relevant information, as do ASTs, but they additionally contain nodes representing literals, layout, and comments. The reason for allowing such concrete syntax information in trees is that many applications, e.g. software renovation, require preservation of layout and comments during tree transformation.

2.2.1 Concrete syntax definition

In order to satisfy our adaptability, scalability and maintainability demands, the concrete syntax definition formalism must satisfy a number of criteria. The syntax definition formalism must have powerful support for modularity and reuse. It must be possible to extend languages without changing the grammar for the base language. This is essential, because each change to a grammar on which tooling is based potentially leads to a modification avalanche[1]. Also, the syntax definition language must be purely declarative. If not, its reusability for different purposes is compromised.

In our instantiation of the meta-tool architecture, the central role of concrete syntax definition language is fulfilled by the Syntax Definition Formalism SDF [HHKR89]. Figure 2.3 shows an example of an SDF grammar. This example definition contains lexical and context-free syntax definitions distributed over a number of modules. Note that the orientation of productions is flipped with respect to BNF notation.

SDF offers powerful modularization features. Notably, it allows modules to be mutually dependent, and it allows alternatives of the same non-terminal to be spread across multiple modules. For instance, the syntax of a kernel language and the syntaxes of its extensions can be defined in separate modules. Also, mutually dependent non-terminals can be defined in separate modules. Renamings and parameterized modules further facilitate syntax reuse.

SDF is a highly expressive syntax definition formalism. Apart from symbol iteration constructors, with or without separators, it provides notation for optional symbols, sequences of symbols, optional symbols, and more. These notations for building compound symbols can be arbitrarily nested. SDF is not limited to a subclass of context-free grammars, such as LR or LL grammars. Since the full class of context-free syntaxes, as opposed to any of its proper subclasses, is closed under composition (combining two context-free grammars will always produce a

[1]The generic traversal techniques to be presented in the upcoming chapters alleviate the dependence of tools on grammars, but generally do not quite eliminate it.

```
definition
module Exp
exports
  context-free syntax
    Identifier                         → Exp {cons(var)}
    Identifier "(" {Exp ","}* ")"      → Exp {cons(fcall)}
    "(" Exp ")"                        → Exp {bracket}

module Let
exports
  context-free syntax
    let Defs in Exp  → Exp {cons(let)}
    Exp where Defs   → Exp {cons(where)}

module Def
exports
  aliases
    {( Identifier "=" Exp ) ","}+ → Defs

module Main
imports Exp Let Def

exports
  sorts Exp
  lexical syntax
    [\␣\t\n] → LAYOUT
  context-free restrictions
    LAYOUT?      -/- [\␣\t\n]
```

Figure 2.3: An example SDF grammar.

grammar that is context-free as well), this absence of restrictions is essential to obtain true modular syntax definition, and "as-is" syntax reuse.

SDF offers disambiguation constructs, such as associativity annotations and relative production priorities, that are decoupled from constructs for syntax definition itself. As a result, disambiguation and syntax definition are not tangled in grammars. This is beneficial for syntax definition reuse. Also, SDF grammars are purely declarative, ensuring their reusability for other purposes besides parsing (e.g. code generation, pretty-printing).

SDF offers the ability to control the shape of parse trees. The alias construct (see module *Def* in Figure 2.3) allows auxiliary names for complex sorts to be introduced without affecting the shape of parse trees or abstract syntax trees. Aliases are resolved by a normalization phase during parser generation, and they do not introduce auxiliary nodes.

2.2.2 Concrete meta-tooling

Parsing SDF is supported by *generalized* LR parser generation [Rek92]. In contrast to plain LR parsing, generalized LR parsing is able to deal with (local) ambiguities and thereby removes any restrictions on the context-free grammars. A detailed argument that explains how the properties of GLR parsing contribute to meeting the scalability and maintainability demands of language-centered application areas can be found in [BSV98]. The meta-tooling used for parsing in our architecture consist of a parse table generator `pgen`, and a generic parse component, called `sglr`, which parses terms using these tables, and generates parse trees [Vis97].

Parse tree representation In our architecture instantiation, the parse trees produced from generated parsers are represented in the SDF parse tree format, called AsFix [Vis97]. AsFix trees contain all information about the parsed term, including layout and comments. As a consequence, the exact input term can always be reconstructed, and during tree processing layout and comments can be preserved. This is essential in the application area of software renovation.

Full AsFix trees rapidly grow large and become inefficient to represent and exchange. It is therefore of vital importance to have an efficient representation for AsFix trees available. Moreover, component based software development requires a uniform exchange format to share data (including parse trees) between components. The ATerm format is a term representation suitable as exchange format for which an efficient representation exists. Therefore AsFix trees are encoded as ATerms to obtain space efficient exchangeable parse trees ([BJKO00] reports compression rates of over 90 percent). In Section 2.3.2 we will discuss tree representation using ATerms in more detail.

Pretty-printing We use GPP, a generic pretty-printing toolset that has been defined in [Jon00]. This set of meta-tools provides the generation of customizable pretty-printers for arbitrary languages defined in SDF. The layout of a language is expressed in terms of pretty-print rules which are defined in an ordered sequence of pretty-print tables. The ordering of tables allows customization by overruling existing formatting rules.

GPP contains a formatter which operates on AsFix parse trees and supports comment and layout preservation. An additional formatter which operates on ASTs is also part of GPP.

Since GPP is an open system which can be extended and adapted easily, support for new output formats (in addition to plain text, $\LaTeX$, and HTML which are supported by default) and language specific formatters can be incorporated with little effort.

2.3 Abstract syntax definition and meta-tooling

A second aspect of meta-tooling for component based language tool development concerns the generation of code from *abstract* syntax definitions. Figure 2.4 shows the architecture of such tooling. Given an abstract syntax definition, library code is generated, which is used to represent and manipulate ASTs. The abstract syntax definition language serves as an interface description language for AST components. In other words, abstract syntax definitions serve as tree type definitions (analogous to XML's document type definitions).

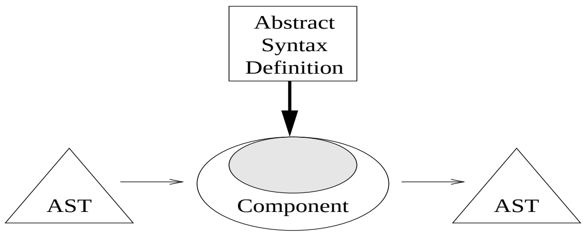


Figure 2.4: Architecture for *abstract syntax* meta-tools. The abstract syntax definition, prescribing tree structure, serves as a contract between tree processing components.

2.3.1 Abstract syntax definition

For the specification of abstract syntax we have defined a subset of SDF, which we call AbstractSDF. AbstractSDF was obtained from SDF simply by omitting all constructs specific to the definition of *concrete* syntax. Thus, AbstractSDF allows only productions specifying prefix syntax, and it contains no disambiguation constructs or constructs for specifying lexical syntax. AbstractSDF inherits the powerful modularity features of SDF, as well as the high expressiveness concerning arbitrarily nested compound sorts. Figure 2.5 shows an example of an AbstractSDF definition.

The need to define separate concrete syntax and abstract syntax definitions would cause a maintenance problem. Therefore, the concrete syntax definition can be annotated with abstract syntax directives from which an AbstractSDF definition can be generated (see Section 2.3.3 below). These abstract syntax directives consist of optional constructor annotations for context-free productions (the “cons” attributes in Figure 2.3) which specify the names of the corresponding abstract syntax productions.

2.3.2 Abstract syntax tree representation

In order to meet our scalability demands, we will require a tree representation format that provides the possibility of efficient storage and exchange. However, we do not want a tree format that has an efficient binary instantiation only, since this makes all tooling necessarily dependent on routines for binary encoding. Having a human readable instantiation keeps the system open to the accommodation of components for which such routines are not (yet) available. Finally, we want the typing of trees to be *optional*, in order not to preempt integration with typeless generic components. For instance, a generic tree viewer should be able to read the intermediate trees without explicit knowledge of their types.

ASTs are therefore represented in the ATerm format, which is a generic format

```
definition
module Exp
exports
  syntax
    "var" ( Identifier )           → Exp
    "fcall" ( Identifier, Exp* ) → Exp

module Let
exports
  syntax
    "let" ( Defs, Exp )       → Exp
    "where" ( Exp, Defs ) → Exp

module Def
exports
  aliases
    ( Identifier Exp )+ → Defs

module Main
imports Exp Let Def
```

Figure 2.5: Generated AbstractSDF definition.

for representing annotated trees. In [BJKO00] a 2-level API is defined for ATerms. This API hides a space efficient binary representation of ATerms (BAF) behind interface functions for building, traversing and inspecting ATerms. The binary representation format is based on maximal subtree sharing. Apart from the binary representation, a plain, human-readable representation is available.

AbstractSDF definitions can be used as type definitions for ATerms by language tool components. In particular, the AbstractSDF definition of the parse tree formalism AsFix serves as a type definition for parse trees (See Section 2.2). The AbstractSDF definition of Figure 2.5 defines the type of ASTs representing expressions. Thus, the ATerm format provides a generic (type-less) tree format, on which AbstractSDF provides a typed view.

2.3.3 Abstract from concrete syntax

The connection between the abstract syntax meta-tooling and the concrete syntax meta-tooling can be provided by three meta-tools, which are depicted in Figure 2.6. Central in this picture is a meta-tool that derives an abstract syntax definition from a concrete syntax definition. The two accompanying meta-tools generate tools for converting full parse trees into ASTs and vice versa. Evidently, these ASTs should correspond to the abstract syntax definition which has been generated from the concrete syntax definition to which the parse trees correspond.

An abstract syntax definition is obtained from a grammar in two steps. Firstly, concrete syntax productions are optionally annotated with prefix constructor names. To derive these constructor names automatically, the meta-tool `sdfcons` has been implemented. This tool basically collects keywords and non-terminal names from productions and applies some heuristics to synthesize nice names from these. Non-unique constructors are made unique by adding primes or qualifying with non-terminal names. By manually supplying some seed constructor names, users can steer the operation of `sdfcons`, which is useful for languages which sparsely

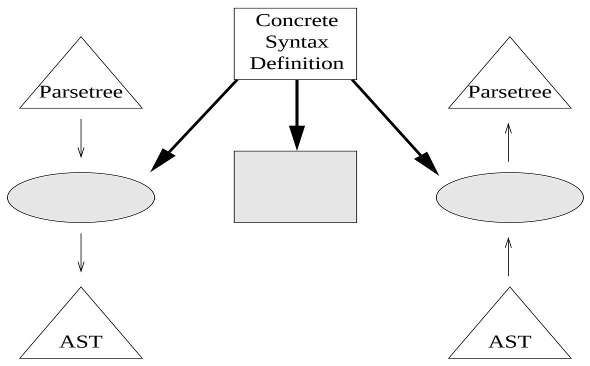


Figure 2.6: Architecture for meta-tools linking abstract to concrete syntax. The abstract syntax definition is now generated from the concrete syntax definition.

contain keywords.

Secondly, the annotated grammar is fed into the meta-tool `sdf2asdf`, yielding an AbstractSDF definition. For instance, the AbstractSDF definition in Figure 2.5 was obtained from the SDF definition in Figure 2.3. This transformation basically throws out literals, and replaces mixfix productions by prefix productions, using the associated constructor name.

Together with the abstract syntax definition, the converters `parsetree2ast` and `ast2parsetree` which translate between parse trees and ASTs are generated. Note that the first converter removes layout and comment information, while the second inserts *empty* layout and comments.

Note that the high expressiveness of SDF and AbstractSDF, and their close correspondence are key factors for the feasibility of generating abstract from concrete syntax. In fact, SDF was originally designed with such generation in mind [HHKR89]. Standard, Yacc-like concrete syntax definition languages are not satisfactory in this respect. Since their expressiveness is low, and LR restrictions require non-natural language descriptions, generating abstract syntax from these languages would result in awkwardly structured ASTs, which burden the component programmers.

2.4 Generating library code

In this section we will discuss the generation of library code (see Figures 2.2 and 2.4). Our language tool development architecture contains code generators for several languages and consequently allows components to be developed in different languages. Since ATerms are used as uniform exchange format, components implemented in different programming languages can be connected to each other.

2.4.1 Targeting C

For the programming language C an efficient ATerm implementation exists as a separate library. This implementation consists of an API which hides the efficient binary representation of ATerms based on maximal sharing and provides functions to access, manipulate, traverse, and exchange ATerms.

The availability of the ATerm library allows generic language components to be implemented in C which can perform low-level operations on arbitrary parse trees as well as on abstract syntax trees.

A more high-level access to parse trees is provided by the code generator `asdf2c` which, when passed an abstract syntax definition, produces a library of match and build functions.[2] These functions allow easy manipulation of parse trees without having to know the exact structure of parse trees. These high-level functions are type-preserving with respect to the AbstractSDF definition.

2.4.2 Targeting Java

In Chapters 5 and 6 the tool support for targeting Java will be discussed in detail. For the Java programming language, as for C, an implementation of the ATerm API exists which allows Java programs to operate on parse trees and abstract syntax trees. The code generator JJForester has been developed to provide high level access and traversals of trees similar to the other supported programming languages. Here, syntax trees are represented as object trees, and tree traversals are supported by instantiation of the visitor combinator framework JJTraveler.

2.4.3 Targeting Stratego

Our initial interest was to apply our meta-tooling to program transformation problems, such as automatic software renovation. For this reason we selected the transformational programming language Stratego [Vis99] as the first target of code generation. Stratego offers powerful tree traversal primitives, as well as advanced features such as separation of pattern-matching and scope, which allows pattern-matching at arbitrary tree depths. Furthermore, Stratego has built-in support for reading and writing ATerms. Stratego also offers a notion of pseudo-constructors, called *overlays*, that can be used to operate on full parse trees using a simple AST interface.

Two meta-tools support the generation of Stratego libraries from syntax descriptions. The library for AST processing is generated by `asdf2stratego` from an AbstractSDF definition. The library for combined parse tree and AST processing is generated by `sdf2stratego` from an SDF grammar. The latter library subsumes the former[3].

[2]The `asdf2c` has been subsumed by ApiGen [JO02].

[3]Code generation for Stratego has further been elaborated and applied in [Wes02].

The Stratego code generation allows programming on parse trees as if they were ASTs. Underneath such AST-style manipulations, parse trees are processed in which hidden layout and literal information is preserved during transformation. This style of programming can be mixed freely with programming directly on parse trees. Since Stratego has native ATerm support, there is no need for generating library code for reading and writing trees.

2.4.4 Targeting Haskell

In Chapters 3 and 4, the support for targeting Haskell as available in Tabaluga and Strafunski will be discussed. Code generated in this case is of various kinds. Firstly, the meta-tool `sdf2haskell` generates datatypes to represent parse trees and ASTs. These datatypes are quite similar to the signatures generated for Stratego. Secondly, an extended version of the DrIFT code generator can be used to generate exchange and traversal code from these datatypes. The generated exchange code allows reading ATerm representations into the generated Haskell datatypes and writing them to ATerms. The generated traversal code allows composition of analyses and traversals from either updatable fold combinators or functional strategy combinators. We developed the *Haskell ATerm Library* to support input and output of ATerms from Haskell types.

Note that not only general purpose programming languages of various paradigms can be fitted into our architecture, but also more specialized, possibly very high-level languages. An attribute grammar system, for instance, would be a convenient tool to program certain tree transformation components.

2.5 A comprehensive architecture

Combining the partial architectures of the foregoing subsections leads to the complete architecture in Figure 2.7. This figure can be viewed as a refinement of our first general architecture in Figure 2.1, which does not differentiate between concrete and abstract syntax, or between parse trees and ASTs.

The refined picture shows that all generated code (libraries and components), and the abstract syntax definition stem from the same source: the grammar. Thus, this grammar serves as the single contract that governs the structure of all trees that are exchanged. In other words, all component interfaces are defined in a single location: the grammar. (When several languages are involved, there are of course equally many grammars.) This single contract approach eliminates many maintenance headaches during component-based development. Of course, careful grammar version management is needed when maintenance due to language changes is not carried out for all components at once.

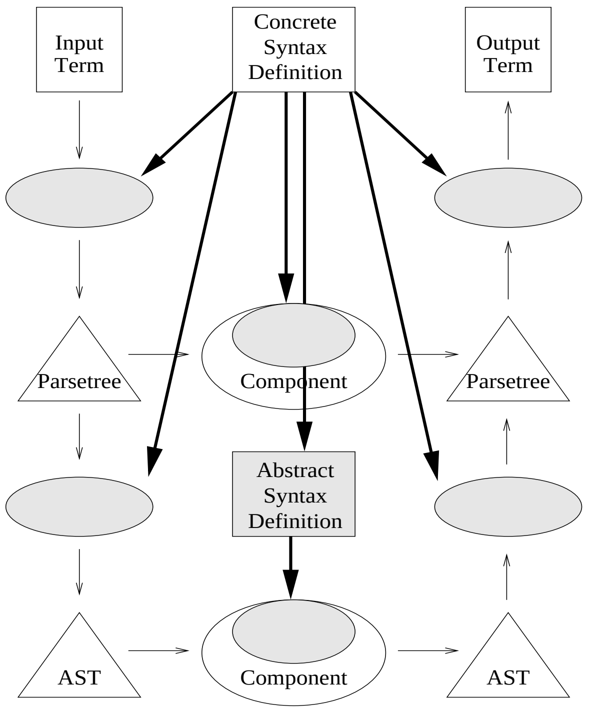


Figure 2.7: Complete meta-tooling architecture. The grammar serves as the contract governing all tree exchange.

2.5.1 Grammar version management

Any change to a grammar, no matter how small, potentially breaks all tools that depend on it. Thus, sharing grammars between tools or between tool components, which is a crucial feature of our architecture, is potentially at odds with grammar *change*. To pacify grammar change and grammar sharing, grammar management is needed.

To facilitate grammar version management, we established a *Grammar Base*, in which grammars are stored. Furthermore, we subjected the stored grammars to simple schemes of grammar version numbers and grammar maturity levels.

To allow tool builders to unequivocally identify the grammars they are building their tool on, each grammar in the Grammar Base is given a name and a version number. To give tool builders an indication of the maturity of the grammars they are using to build their tools upon, all grammars in the Grammar Base are labeled with a maturity level. We distinguish the following levels:

volatile	The grammar is still under development.
stable	The grammar will only be subject to minor changes due to bug fixing.
immutable	The grammar will never change.

Normally, a grammar will begin its life cycle at maturity level *volatile*. To build extensive tooling on such a grammar is unwise, since grammar changes are to be expected that will break this tooling. Once confidence in the correctness of the grammar has grown, usually through a combination of testing, bench-marking, and code inspection, it becomes eligible for maturity level *stable*. At this point, only very local changes are still allowed on the grammar, usually to fix minor bugs. Tool-builders can safely rely on stable grammars without risking that their tools will break due to grammar changes. Only a few grammars will make it to level *immutable*. This happens for instance when a grammar is published, and thus becomes a fixed point of reference. If the need for changes arises in grammars that are stable or immutable, a *new* grammar (possibly the same grammar with a new version number) will be initiated instead of changing the grammar itself.

2.5.2 Connecting components

The connectivity to different programming languages allows components to be developed in the programming language of choice. The use of ATerms for the representation of data allows easy and efficient exchange of data between different components and it enables the composition of new and existing components to form advanced language tools.

Exchange between components and the composition of components is supported in several ways. First, components can be combined using standard scripting techniques and data can be exchanged by means of files. Secondly, the uniform data representation allows for a sequential composition of components in which Unix pipes are used to exchange data from one component to another. Finally, the ToolBus [BK96] architecture can be used to connect components and define the communication between them. This architecture resembles a hardware communication bus to which individual components can be connected. Communication between components only takes place over the bus and is formalized in terms of Process Algebra [BW90].

2.6 Applications

Extensive experience is available about actually applying the meta-tooling presented in the previous sections. We will present a selection of such experiences.

To start with, the meta-tooling has been applied for its own development, and for the development of some other meta-tools that it is bundled with in the Transformation Tools package XT. These bootstrap flavored applications include the generation of an abstract syntax definition for the parse tree format AsFix from the grammar of SDF. From this abstract syntax definition, a modular Stratego library for transforming AsFix trees was generated and used for the implementation of some AsFix normalization components. Also, the tools `sdf2stratego`, `sdfcons`, `asdf2stratego`, `sdf2asdf`, and many more meta-tools were implemented by parsing, AST processing in one or more components, and pretty-printing.

Apart from SDF and AbstractSDF, the domain specific languages BOX (for generic formatting), and BENCH (for generating benchmark reports), have been implemented with syntax-driven meta-tooling. In the BOX implementation, a grammar for pretty-print tables was built by reusing the SDF grammar and the BOX grammar. New BOX components were implemented in Stratego and connected to existing BOX components programmed in other languages.

The generated transformation frameworks for Haskell have been applied to software renovation problems. In [KLV00], a Cobol renovation application is reported. It involves parsing according to a Cobol grammar, applying a number of function transformers to solve a data expansion problem, and unparsing the transformed parse trees. The functional transformers have been constructed by refining a transformation framework generated from the Cobol grammar.

The Stratego meta-tools have been elaborated and applied in the CobolX project [Wes02]. Transformations implemented in this project include goto-elimination, and data field expansion with preservation of layout and comments.

In the upcoming chapters, further applications will be described. Chapter 4 describes the implementation of Java refactoring. Chapter 5 describes analysis of GraphXML, where the roots and sinks are extracted from a graph document. Chapter 6 contains a case study in which communication graphs are generated from Toolbus scripts. Chapter 7 describes procedure reconstruction for Cobol for program understanding purposes.

2.7 Related work

Syntax-driven meta-tools for language tool development are ubiquitous, but rarely do they address a combination of features such as those addressed in this chapter. We will briefly discuss a selection of approaches some of which attain a degree of integration of various features.

- Parser generators such as Yacc [Joh75] and JavaCC are meta-tools that generate parsers from syntax definitions. Compared with SDF with its supporting tools `pgen` and `sglr`, they offer poor support for *modular* syntax definition, their input languages are not sufficiently declarative to be reusable for the generation of other components than parsers, and they do not generally target more than a single programming language.

- The language SYN [Bou96] combines notations for specifying parsers, pretty-printers and abstract syntax in a single language. However, the underlying parser generator is limited to LALR(1), in order to have both parse trees and ASTs, users need to construct two grammars, and code the mapping between trees by hand. Moreover, the expressiveness of the language is much smaller than the expressiveness of SDF, and the language is not modular. Consequently, SYN and its underlying system can not meet our adaptability, scalability and maintainability requirements.

- Wile [Wil97] describes derivation of abstract syntax from concrete syntax. Like us he uses a syntax description formalism more expressive than Yacc's BNF notation in order to avoid warped ASTs. Additionally, he provides a procedure for transforming a Yacc-style grammar into a more "tasteful" grammar. His BNF extension allows annotations that steer the mapping with the same effect as SDF's aliases. He does not discuss automatic name synthesis.

- AsdlGen [WAKS97] provides the most comprehensive approach we are aware of to syntax-driven support of component-based language tools. It generates library code for various programming languages from abstract syntax definitions. It offers ASDL as abstract syntax definition formalism, and *pickles* as space-efficient exchange format. It offers no support for dealing with concrete syntax and full parse trees.

  AsdlGen targets more languages than our architecture instantiation does at the moment. The choice of target languages, including C and Java, has presumably motivated some restrictions on the expressiveness of ASDL. ASDL lacks certain modularity features, compared to AbstractSDF: no mutually dependent modules, and all alternatives for a non-terminal must be grouped together. Furthermore, ASDL is much less expressive. It does not allow nesting of complex symbols, it has a very limited range of symbol constructors, and it does not provide module renamings or parameterized modules.

  Unlike ATerms, the exchange format that comes with ASDL is always typed, thus obstructing integration with typeless generic components. In fact, the compression scheme of ASDL relies on the typedness of the trees. The rate of compression is significantly smaller than for ATerms [BJKO00]. Furthermore, pickles have a binary form only.

- The DTD notation of XML [BPSM98] is an alternative formalism in which abstract syntax can be defined. Tools such as HaXML [WR99] generate code from DTDs. HaXML offers support both for type-based and for generic transformations on XML documents, using Haskell as programming language. Other languages are not targeted. Concrete syntax support is not integrated.

XML is originally intended as mark-up language, not to represent abstract syntax. As a result, the language contains a number of inappropriate constructs, and some awkward irregularities from an abstract syntax point of view. XML also has some desirable features, currently not offered by AbstractSDF, such as annotations, and inclusion of DTDs (abstract syntax definitions) in documents (abstract terms).

- Many elements of our instantiation of the architecture for syntax-driven component-based language tool development were originally developed in the context of the ASF+SDF *Meta-Environment* [BHK89, HHKR89, DHK96, BDH$^+$01]. This is an integrated language development environment which offers SDF as syntax definition formalism and the term rewriting language ASF as programming language. Programming takes place directly on concrete syntax, thus hiding parse trees from the programmers view. Programming, debugging, parsing, rewriting and pretty-printing functionality are all offered via a single interactive user interface. Meta-tooling has been developed to generate ASF-modules for term traversal from SDF definitions [BSV97].

  The ASF+SDF Meta-Environment is an interactive environment for component-based development of language tools. It offers a single programming language (ASF), and programming on abstract syntax is not supported.

  To provide support for component-based tool development, we have adopted the SDF, AsFix, and ATerm formats from the ASF+SDF Meta-Environment as well as the parse table generator for SDF, the parser `sglr`, and the ATerm library. To these we have added the meta-tooling required to complete the instantiation of the architecture of Figure 2.7. In future, some of these meta-tools might be integrated into the Meta-Environment.

2.8 Contributions

We have presented a comprehensive architecture for syntax-driven meta-tooling that supports component based language tool development. This architecture embodies the vision that grammars can serve as contracts between components under the condition that the syntax definition formalism is sufficiently expressive and the meta-tools supporting this formalism are sufficiently powerful. We have presented our instantiation of such an architecture based on the syntax formalism SDF. SDF and the tools supporting it have agreeable properties with respect to modularity, expressiveness, and efficiency, which allow them to meet scalability and maintainability demands of application areas such as software renovation and domain-specific language implementation. We have shown how abstract syntax definitions can be obtained from grammars. We discussed the meta-tooling which generates library code for a variety of programming languages from concrete and abstract syntax definitions. Components that are constructed with these libraries can interoperate by exchanging ATerms that represent trees.

Chapter 3

Dealing with Large Bananas

This chapter presents techniques for generic traversal in functional programming, based on an elaboration of the established notion of generalized folds. We make these folds *updatable* and *composable*.

Many problems call for a mixture of generic and specific programming techniques. We propose a generic programming approach based on generalized (monadic) folds where a separation is made between basic fold algebras that model generic behavior and updates on these algebras that model specific behavior. We identify particular basic algebras as well as some algebra combinators, and we show how these facilitate structured programming with updatable fold algebras. This blend of genericity and specificity allows programming with folds to scale up to applications involving large systems of mutually recursive datatypes. Finally, we address the possibility of providing generic definitions for the functions, algebras, and combinators that we propose.

This chapter is based on [LVK00].

3.1 Introduction

Polytypic programming [JJ97a, Hin00, CL02] aims at relieving the programmer from repeatedly writing functions of similar functionality for different user-defined datatypes. For example, for any datatype parametric in α, 'crushing' the values of type α in a given structure can be defined *fully* generically [Mee96, Hin99]. Such a generic function abstracts from constructors. It is defined by induction on the structure of datatypes in terms of sums, products and others.

Many problems rather call for a *mixture* of generic and specific programming techniques. Think of a program transformation. On the one hand, it must implement specific behavior for particular constructs of the language at hand. On the other hand, it acts on the remaining constructs in a completely generic way: it preserves them. Or think of a program analysis. It often follows a completely generic scheme such as accumulation or reduction, while usually only a few patterns require specific functionality. This interplay of genericity and specificity has also been observed by others (e.g., [Vis00a]).

To address this mixture of genericity and specificity, we propose a polytypic programming approach based on generalized [Fok92, MFP91] and monadic [Fok94, MJ95] folds for systems of mutually recursive datatypes. It is generally accepted that programming with folds (or, more generally, with morphisms) is desirable because it imposes 'structured programming', it facilitates (optimizing) program transformation, it untangles traversal schemes from traversal-specific ingredients, and it facilitates reasoning about programs. Programming with folds offers a restricted form of generic programming, in the sense that traversal schemes such as fold functions can be defined generically for large classes of datatypes. Recent research has focused on extending the class of permitted datatypes, and on identifying the various traversal schemes and their properties [Mee92, FSS92, SF93, MH95, BP99].

Yet, programming with generalized folds is not truly generic because actual programming means to pass algebras to the fold function. These algebras provide the ingredients of the actual traversal, and their structure depends on the actual datatype. Thus, while the traversal schemes might be generic, their instantiations are obtained through non-generic programming.

We propose to separate constructing fold algebras into (i) obtaining a generic fold algebra through polytypic programming and/or reuse from a library of basic fold algebras and algebra combinators, and (ii) updating the generic algebra with specific behavior for particular constructors. This separates the places where one wants to be generic from the places where one needs to be specific. Since both algebras and updates on them are regarded as first-class citizens, structured programming with them is facilitated. In particular, we identify some generic functions for calculating with monadic folds.

Our approach can be used, for example, for the development of program transformations and analyses in the context of legacy system renovation [CC90, BSV00], where one is concerned with the adaptation of large software applications, for example written in Cobol. The sheer size of the underlying languages in this area makes some sort of generic programming indispensable; defining traversals on the language's syntax non-generically is simply not feasible. Yet, for particular constructors specific behavior must be specified. Programming with folds scales up to these kinds of problems when a functional language is used that provides, as we propose, generalized folds for mutually recursive datatypes and a combinator language for fold algebras, including a mechanism for updating generic algebras with specific behavior.

Section 3.2 briefly recapitulates the various elements involved in existing methods of programming with folds. Section 3.3 explains the separation of generic algebras and algebra updates, which is the key to scalable programming with folds. Section 3.4 extrapolates this separation to monadic folds. Throughout these sections, a running Haskell example, adapted from [MJ95], is used to identify and illustrate the required elements for programming with updatable folds. Section 3.5 provides a more abstract formulation of our approach, including polytypic definitions of some elements.

3.2 Programming with folds

Using an example adapted from [MJ95], we will quickly recapitulate the various elements involved in existing methods of programming with folds. Moreover, we will explain the lack of scalability of these methods.

Remarks We use Haskell examples throughout the chapter. In particular, we use Haskell 98 extended with multi-parameter type classes, which are supported by the main Haskell implementations. We use classes to overload functions merely for convenience — our treatment does not rely on them. We chose not to use 'functional dependencies' [Jon99] in class headers (as in: $\mathbf{class}\ \mathit{Fold}\ \mathit{alg}\ t\ a \mid \mathit{alg}\ t \to a\ \mathbf{where}\ldots$). This would make more accurate overloading resolution possible, allowing the user to write fewer explicit types, but it is currently not supported by all Haskell implementations. In Section 3.4 on monadic folds, we make use of stackable monads from Andy Gill's Monad Template Library, to be found via `http://www.haskell.org`.

3.2.1 An example

When using folds, a programmer writes functions consuming values of a datatype D in terms of a fold function which captures the recursive traversal scheme for D. The fold function is parameterized by a *fold algebra*, which holds as many functions as there are constructors in the datatype. These functions are meant to replace the constructors in the traversal.

Example 3.2.1 *Assume for example the following system of datatypes, which represents the abstract syntax of a simple functional language:*

```
data Type   =  TVar String
            |  Arrow Type Type
data Expr   =  Var String
            |  Apply Expr Expr
            |  Lambda (String, Type) Expr
```

The type of the algebras that parameterize folds over this system of datatypes is the following:

```
data Cata a b  =  Cata{ tvar :: String → a
                      , arrow :: a → a → a
                      , var :: String → b
                      , apply :: b → b → b
                      , lambda :: (String, a) → b → b }
```

The algebra type is named $Cata$ because the corresponding fold functions capture the catamorphic *scheme of recursion. We will comment on paramorphisms [Mee92] in Section 3.5. We use a flat Haskell record to model an algebra for usability reasons. There are other possible encodings. Some of them will be discussed in Section 3.6.*

The family of fold functions for the system of datatypes can be represented by the following class and instance declarations:

```
class Fold alg t a where
  fold        ::  alg → t → a

instance Fold (Cata a b) Type a where
  fold alg (TVar x)           =  (tvar alg) x
  fold alg (Arrow s t)        =  (arrow alg) (fold alg s) (fold alg t)
instance Fold (Cata a b) Expr b where
  fold alg (Var x)            =  (var alg) x
  fold alg (Apply f a)        =  (apply alg) (fold alg f) (fold alg a)
  fold alg (Lambda (x, t) b)  =  (lambda alg) (x, (fold alg t)) (fold alg b)
```

Note that in general the fold functions can be mutually recursive just like the system of datatypes. Given these definitions, a programmer can begin to write functions consuming values of one of the datatypes, by passing appropriate algebra values to one of the fold functions. For instance, a function for constant function elimination can be written as follows:

```
cfe       ::  Expr → Expr
cfe       =   fold cfeAlg
cfeAlg    ::  Cata Type Expr
cfeAlg    =   Cata{ tvar = TVar
                  , arrow = Arrow
                  , var = Var
                  , apply = λf a → case f of
                                   (Lambda (x, t) b) → if ¬ (elem x (freevars b))
                                                       then b
                                                       else (Apply f a)
                                   _ → (Apply f a)
                  , lambda = Lambda}
```

The function $freevars$, which collects free variables from a given expression, can be programmed in the same style, as we will show in Section 3.3.3. ⊠

3.2.2 Scalability problems

Imagine using the technique of programming with folds, not for the toy language of Example 3.2.1, which has a syntax definition with two nonterminals (types) and five productions (constructors), but for Cobol, which has a syntax definition with several hundreds of nonterminals and productions. This occurs in the application areas of program analysis and transformation such as legacy system renovation [CC90, BSV00]. There are several problems with respect to scalability:

Initial effort Before programming with folds can begin, the algebra type and the fold functions need to be defined. Since both the number of field declarations in the algebra type, and the number of function equations are equal to the number of constructors, the effort involved is proportional to the size of the syntax definition.

Repeated effort Instantiating a fold function with an algebra almost requires as much effort as writing a traversal from scratch. The number of field definitions in the fold algebra is again equal to the number of constructors. So, no matter how small the problem to be solved by a traversal, the size of the algebra to be written is proportional to the size of the syntax definition.

In principle, the first problem can be solved by generating folds (refer, e.g., to [BB85, She91]), offering them as language primitives (as, e.g., in Charity [CS92]), or providing polytypic definitions for them (as, e.g., in PolyP [JJ97a] or Generic Haskell [CL02]). However, there are some problems with the existing approaches regarding systems of mutually recursive datatypes and the kind of algebra notion supported by them. In Section 3.5, we attempt to improve on these existing approaches. To solve the second problem, this chapter proposes to separate generic fold algebras from language-specific updates on them. This is explained in Sections 3.3 and 3.4.

3.3 Programming with updatable fold algebras

We propose to separate the construction of fold algebras into (i) obtaining a basic algebra, and (ii) updating the algebra. This separates the places where one wants to be generic from the places where one needs to be specific. In this section, we will explain how programming with updatable fold algebras proceeds, and we will identify some useful basic fold algebras. In Section 3.4, some sophistication is added to the technique of programming with folds by accommodating monads and (monadic) fold algebra combinators. Finally, in Section 3.5, the generic structures involved in programming with updatable fold algebras are given generic definitions.

3.3.1 Updating algebras

To explain the separation into basic algebras and updates, we revisit Example 3.2.1.

Example 3.3.1 *The algebra $cfeAlg$ can be constructed by applying a fold algebra update to a basic fold algebra. In this particular case, the basic fold algebra $idmap$ is appropriate:*

```
idmap     ::  Cata Type Expr
idmap     =   Cata{ tvar = TVar
                  , arrow = Arrow
                  , var = Var
                  , apply = Apply
                  , lambda = Lambda }
```

The generic behavior captured by this algebra is to traverse a term without changing it, i.e., $fold$ applied to $idmap$ is the identity function. This holds because constructors are replaced by themselves. This is a law each fold should satisfy [MJ95]. In order to obtain $cfeAlg$ from $idmap$, we apply the update $cfeUpd$:

```
cfeAlg = cfeUpd idmap
cfeUpd alg = alg{ apply = λf a → case f of
                  (Lambda (x, t) b) → if ¬ (elem x (freevars b))
                                       then b
                                       else (Apply f a)
                  _ → Apply f a}
```

Here we make use of the Haskell syntax $r\{a_1 = x_1, \ldots, a_n = x_n\}$ for record update. ⊠

The separation of a basic fold algebra and an update on it, is the key to making programming with folds scalable. The basic fold algebra, which is proportional to the size of the language's syntax definition, can be derived automatically or defined polytypically (see Section 3.5). The update needs to contain problem-specific functionality only, and is provided by the programmer.

3.3.2 Type-preserving and type-unifying

The basic fold algebra $idmap$ and all algebras obtained by updating it are *type-preserving* in the sense that when folding with them a $Type$ is mapped to a $Type$, and an $Expr$ is mapped to an $Expr$. This is captured by the following type synonym:

```
type Preserve  =  Cata Type Expr
```

Type-preserving algebras are useful for programming (program) transformations. Another important class of algebras are the *type-unifying* ones. These map both $Expr$ and $Type$ onto the same result type. This is captured as follows:

```
type Unify a  =  Cata a a
```

The next subsection features such type-unifying algebras. As will become clear, type-unifying algebras are useful for programming (program) analyses.

3.3.3 Crushing

We start our discussion of type-unifying basic fold algebras with the parameterized basic fold algebra *crush*.

```
crush        ::  a → (a → a → a) → Unify a
crush e o    =   Cata{ tvar = λx → e
                     , arrow = λa b → a ‘o‘ b
                     , var = λx → e
                     , apply = λa b → a ‘o‘ b
                     , lambda = λ(x, t) b → t ‘o‘ b}
```

The parameters of this algebra, i.e., the value e and the binary operator o, are assumed to form a monoid. Alternatively, a type class *Monoid* could have been used here. Instantiation of *crush* would then proceed by type specialization instead of passing parameters explicitly. The name *crush* is inspired by the related concept of polytypic crushing on parameterized datatypes [Mee96, Hin99]. Polytypic crushing means to collect and to reduce all values of type α in a datatype parametric in α. In contrast, our *crush* has to be updated before it collects values in a given data structure at all. The basic algebra just defines the reduction of intermediate results. Given a term t, the expression *fold* (*crush e o*) t will be evaluated to e, if we assume the monoid unit laws. This type of reduction does not depend on the parameterization of a datatype.

Example 3.3.2 *To demonstrate the use of the type-unifying parameterized algebra crush, we will define a program analysis that collects free variables. First we instantiate crush to obtain a basic fold algebra collect:*

```
collect      =   crush [] (++)
```

Then we define a collector of variables:

```
vars         ::  Expr → [String]
vars         =   fold (varsUpd collect)
varsUpd alg  =   alg{ var = λx → [x]}
```

And we derive a collector of free variables as needed in Example 3.2.1:

```
freevars     ::  Expr → [String]
freevars     =   fold (fvUpd collect)
fvUpd alg    =   (varsUpd alg){ lambda = λ(x, t) b → filter (x ≢) b}
```

This two-step update illustrates the modularization of algebra updates. In Section 3.4 another technique is discussed. Of course, collect could have been updated in two points (i.e., constructors) at once. ☒

Example 3.3.3 *Another use of crush is to build a basic fold algebra count for counting:*

```
count        =   crush 0 (+)
```

A counter of variables is constructed by updating count:

```
countvars      :: Expr → Integer
countvars      =  fold (cvUpd count)
cvUpd alg      =  alg{var = λx → 1}
```

⊠

3.4 Merging monads and updatable folds

There are several reasons for using monads in combination with updatable fold algebras. Firstly, monadic effects can be used to address issues such as context propagation (environment monad), side-effects (I/O and state monad), and failure (error monad). Secondly, monadic updatable folds can be used to elegantly modularize programs.

3.4.1 Monadic folds

Monadic folds are explained in [Fok94, MJ95]. Some variants are discussed in [MBJ99]. The monadic algebra type and type synonyms for type-preserving and type-unifying monadic algebras for our example language are as follows:

```
data Monad m ⇒ MCata m a b
                        =  MCata{ mtvar :: String → m a
                                , marrow :: a → a → m a
                                , mvar :: String → m b
                                , mapply :: b → b → m b
                                , mlambda :: (String, a) → b → m b }
type MPreserve m        =  MCata m Type Expr
type MUnify m a         =  MCata m a a
```

For brevity we give the monadic fold function only for *Type*; for *Expr* it is similar:

```
instance Monad m ⇒ Fold (MCata m a b) Type (m a) where
 fold alg (TVar x)      =  (mtvar alg) x
 fold alg (Arrow s t)   =  do {s' ← fold alg s; t' ← fold alg t; marrow alg s' t'}
```

Note that the traversal scheme modeled by the monadic fold function explicitly sequences the computations of the recursive calls.

3.4.2 Lifting fold algebras

Monadic algebras can be constructed via two routes. Either directly, by updating a monadic basic fold algebra, or indirectly, by updating an ordinary algebra and *lifting* it to a monadic one. The lifting operator *unit* is straightforwardly defined as follows:

```
unit           :: Monad m ⇒ Cata a b → MCata m a b
unit alg       =  MCata{ mtvar = λx → return ((tvar alg) x)
                       , marrow = λa b → return ((arrow alg) a b)
```

```
                           , mvar = λx → return ((var alg) x)
                           , mapply = λf a → return ((apply alg) f a)
                           , mlambda = λxt b → return ((lambda alg) xt b)}
```

Of course, if the programmer wishes to use monadic effects in particular updates, only the direct route is available. As we will show, indirectly constructed updates and directly constructed ones can be composed, so the programmer is not forced to deal with monads where he does not use them.

3.4.3 Fold algebra composition

The algebra update *cfeUpd* of Example 3.3.1 is not quite suitable to be merged (by function composition) with other updates, because the fall-through arm of the **case** and the **else** branch of the conditional explicitly rebuild the original term. It would override the functionality specified by previous updates for all application nodes, not just for constant function applications. To prepare this update for modular composition, it could instead refer to the the algebra *alg* that is being updated (substitute *apply alg* for *Apply*). There is another technique which facilitates merging of updates. It is based on an algebra combinator *plus* and a neutral algebra *zero*.

```
plus      :: MonadPlus m ⇒ MCata m a b → MCata m a b → MCata m a b
plus s s' =  MCata{ mtvar = λx → ((mtvar s) x) ‘mplus‘ ((mtvar s') x)
                  , marrow = λa b → ((marrow s) a b) ‘mplus‘ ((marrow s') a b)
                  , mvar = λx → ((mvar s) x) ‘mplus‘ ((mvar s') x)
                  , mapply = λf a → ((mapply s) f a) ‘mplus‘ ((mapply s') f a)
                  , mlambda = λ(x, t) b → ((mlambda s) (x, t) b)
                                    ‘mplus‘ ((mlambda s') (x, t) b)}
zero      :: MonadPlus m ⇒ MCata m a b
zero      =  MCata{ mtvar = λx → mzero
                  , marrow = λa b → mzero
                  , mvar = λx → mzero
                  , mapply = λf a → mzero
                  , mlambda = λxt e → mzero}
```

These employ a monad with plus and zero (backtracking or error monad) to model the success or failure of algebra members. For convenience we additionally define an algebra combinator *try*, which tries to apply a type-preserving algebra and resorts to *idmap* when it fails:

```
try      :: MonadPlus m ⇒ MPreserve m → MPreserve m
try s    =  s ‘plus‘ (unit idmap)
```

Note that in this definition *idmap* is lifted to obtain a monadic *idmap*.

Example 3.4.1 *The function cfe can now be reformulated.*

```
cfe      :: Expr → Maybe Expr
cfe      =  fold (try cfeAlg :: MPreserve Maybe)
cfeAlg   :: MonadPlus m ⇒ MPreserve m
cfeAlg      =  zero{ mapply = λf a → case f of
                                       (Lambda (x, t) b)
                                         → do guard (¬ (elem x (freevars b)))
                                              return b
                                       _ → mzero }
```

⧄

Algebras formulated as updates on *zero* can freely be combined with other (appropriately typed) algebras by means of the combinators *plus* and *try*. This will be illustrated below.

3.4.4 Carried monads

It is well known that monadic folds are not expressive enough for all effects in traversals [MJ95]. The reason for this is that the sequencing of recursive calls which is weaved into the monadic fold function sometimes needs to be modified. In these cases, monads can be used in a different way, which we call *carried* (vs. *weaved-in*). We introduce the following type synonyms for carried monadic fold algebras:

```
type PreserveM m  =  Cata (m Type) (m Expr)
type UnifyM m a   =  Unify (m a)
```

Note that in carried monadic fold algebras, the sequencing of recursive calls needs to be done explicitly by the programmer. We can define *unit*, *zero*, *plus* and *try* for carried monadic algebras too. As in the weaved-in case, carried monadic algebras can be constructed directly or by lifting ordinary algebras. We will postfix names with M to indicate that carried monads are involved.

3.4.5 Casting weaved-in to carried monadic fold algebras

For some effects, carried monads are necessary, but in general they are more cumbersome than weaved-in monads, because the programmer is burdened with sequencing. Also, the restricted expressiveness of weaved-in monads yields more theorems for free. Fortunately, we can define a function *carried* that casts a weaved-in monadic algebra to a carried one.

```
carried      ::  Monad m ⇒ MCata m t e → Cata (m t) (m e)
carried alg  =   Cata{ tvar = λx → mtvar alg x
                     , arrow = λma mb → do { a ← ma; b ← mb; marrow alg a b }
                     , var = λx → mvar alg x
                     , apply = λmf ma → do {f ← mf; a ← ma; mapply alg f a }
                     , lambda = λ(x, mt) mb → do t ← mt
                                                 b ← mb
                                                 mlambda alg (x, t) b} }
```

The following example shows how *carried* can be used to resort to carried monads only for effects that need them.

Example 3.4.2 *We define an algebra for performing substitutions. An environment monad is used to propagate a context of type $Subst = [(String, Expr)]$. A state monad is used to generate new variable names, which are needed to prevent variable capture.*

```
lookupAlg   ::  (MonadPlus m, MonadReader Subst m) ⇒ MPreserve m
lookupAlg   =   zero{ mvar = λx → mlookup x }
restoreAlg  ::  (MonadPlus m, MonadReader Subst m, MonadState Int m)
            ⇒   PreserveM m
restoreAlg  =   zeroM{ lambda = λ(x, mt) mb →
                                          do env ← ask
                                             x′ ← new_name
                                             t ← mt
                                             b ← restore ((x, Var x′) : env) mb
                                             return (Lambda (x′, t) b)}
substAlg    ::  (MonadPlus m, MonadReader Subst m, MonadState Int m)
            ⇒   PreserveM m
substAlg    =   tryM (carried lookupAlg ‘plusM‘ restoreAlg)
```

The algebra $lookupAlg$ takes care of the actual substitution of a variable. It is defined as a weaved-in monadic algebra. The algebra $restoreAlg$ takes care of adding a renaming of a bound variable to the context before *processing the body of a lambda abstraction. Here, a* carried *monad is needed. In the algebra $substAlg$, these two algebras are combined into a carried algebra, by first casting the weaved-in monadic algebra to a carried one, and then applying $plusM$.* ⊠

3.5 Generic bananas

In the foregoing sections, we gave Haskell definitions of the ingredients for programming with (monadic) updatable folds: the fold algebra type, the fold functions, the basic fold algebras $idmap$, $crush$ and $zero$, the fold combinators $unit$ and $plus$, and the casting function $carried$. These definitions were *specific* to our example system of datatypes.

Of course, to truly enable *generic* programming, programmers should not be burdened with repeatedly supplying such definitions for all systems of datatypes that come up. In this section, we will demonstrate that generic definitions of the ingredients of programming with updatable folds can be given. These definitions can be implemented by a program generator (see Section 3.6), or by supplying them as language primitives in a functional language. This would allow generic programming *with* updatable folds. Alternatively, a generic programming language which allows these definitions to be expressed, would additionally enable programming *of* updatable folds.

3.5.1 Systems of datatypes

In the polytypic definitions to come, we use a flavor of polytypic programming [JJ97a, Hin00]. We will perform induction over the structure of systems of (mutually recursive) datatypes. This structure is given by the following grammar:

S	$::=$	$\emptyset \mid N = D \mid S \cup S$	-- systems of datatypes
D	$::=$	$C\,T \mid D + D$	-- datatype definitions
T	$::=$	$1 \mid T \times T \mid N$	-- type expressions
N			-- names of datatypes
C			-- constructor names

We use s, d, t, n, and c, possibly subscripted or primed, to range over respectively S, D, T, N, and C. For convenience, we introduce the notation $c(s)$ to denote the type of the constructor c in the system s, i.e., if $n = \cdots + c\ t + \cdots \in s$, then $c(s) = t \rightarrow n$.

As the grammar details, a system of datatypes s is a set of equations, a datatype definition d is a sum of types, labeled with constructor names, and a type expression t is a product over names of datatypes. Three features of this grammar are noteworthy. Firstly, constructor names are not suppressed in the representation of datatype definitions. Indeed, constructor names are indispensable when generic programming is to be mixed with specific programming. Secondly, the grammar explicitly distinguishes datatypes from type expressions. If they would be merged into a single nonterminal, that allows both sums and products, unintended expressions would be generated, e.g., sums not qualified with constructors, or constructors occurring inside products. Finally, though constructors are usually typed in a curried fashion, we use products for the parameters of constructors. This allows a more homogeneous treatment as common in polytypic programming. We only consider complete and non-extensible systems of datatypes in this chapter. For the moment being, we limit ourselves to non-parameterized datatypes without function types and nested sums involved. At the end of the section we will discuss whether these limitations can be lifted.

3.5.2 Fold algebras

We need to define the fold algebra type induced by a system s of datatypes. This is a generalization of the algebra type for a single datatype, which is well understood. Since we want to abstract from the concrete structure of algebras (whether they are records or tuples, flat or nested), we will provide a (semi-formal) axiomatization of fold algebras. The Haskell approach of the previous sections should be regarded as one model of this axiomatization.

Intuitively, the algebra type of a datatype system s is obtained as a collection of function types derived from all the constructor types in s by consistently replacing names of datatypes by distinct type variables. To accommodate type variables, we define type schemes $TS \supset T$, i.e., type expressions which may contain type variables. Type schemes are defined according to the following grammar:

$$\begin{array}{lcll} TS & ::= & 1 \mid TS \times TS \mid N \mid X & \text{-- type schemes} \\ X & & & \text{-- type variables} \end{array}$$

We use τ and α to range respectively over TS and X. Now we can proceed to define s-fold algebras. $\mathcal{A}$ is an s-fold algebra for a system s of datatype definitions if:

1. For each equation $n = d$ in s there is a type scheme $\overline{n}(\mathcal{A})$ called *result type (scheme)* for n.

2. We lift the $\overline{n}(\mathcal{A})$ from data names to $\overline{t}(\mathcal{A})$ for type expressions t:

$$\begin{array}{rcl} \overline{1}(\mathcal{A}) & = & 1 \\ \overline{t_1 \times t_2}(\mathcal{A}) & = & (\overline{t_1}(\mathcal{A}) \times \overline{t_2}(\mathcal{A})) \end{array}$$

3. For each constructor c in s there is an *algebra member* $\mathcal{A}.c$ of type $\overline{t}(\mathcal{A}) \rightarrow \overline{n}(\mathcal{A})$, where $c(s) = t \rightarrow n$.

We consider the set of all s-fold algebras as the fold algebra type for the system s.

3.5.3 Fold functions

Generalized folding for systems s of datatypes can be defined by induction on T. In an application $\mathit{fold}\langle t\rangle\ \mathcal{A}\ x$, we require that $\mathcal{A}$ is an s-fold algebra, and x is of type t. The result type of folding is, of course, $\overline{t}(\mathcal{A})$.

$$\begin{array}{rcl} \mathit{fold}\langle 1\rangle\ \mathcal{A}\ () & = & () \\ \mathit{fold}\langle t_1 \times t_2\rangle\ \mathcal{A}\ (x_1, x_2) & = & (\mathit{fold}\langle t_1\rangle\ \mathcal{A}\ x_1, \mathit{fold}\langle t_2\rangle\ \mathcal{A}\ x_2) \\ \mathit{fold}\langle n\rangle\ \mathcal{A}\ (c\ x) & = & \mathcal{A}.c\ (\mathit{fold}\langle t\rangle\ \mathcal{A}\ x) \text{ where } c(s) = t \rightarrow n \end{array}$$

The definition of paramorphic fold functions [Mee92, SF93] and monadic fold functions [Fok94, MJ95] (see also Section 3.4) requires just a modest elaboration of the scheme above. Although we did not illustrate paramorphisms in this chapter, we should mention that the recursion scheme underlying paramorphisms is very desirable for traversals where the structure of subterms needs to be observed. Paramorphisms can be encoded as catamorphisms by a tupling technique, but this is very inconvenient in actual programming.

3.5.4 Basic algebras

Let us now define the basic fold algebras $idmap$, $crush$ and $zero$ induced by a system s. For all c:

$$\begin{array}{rcl} idmap.c & = & c \\ crush.c & = & \lambda x.crush'\langle t\rangle\ x \text{ where } c(s) = t \to n \\ zero.c & = & \lambda x.mzero \end{array}$$

The definition of $idmap$ is immediately clear. For $crush$, we need to define a generic function $crush'$ which performs crushing for parameters of constructors. The definition of this function (and thereby crushing) assumes a monoid $\langle \alpha, e, \circ \rangle$, where α is a type variable, e denotes the neutral element, and $\circ$ denotes the associative operation:

$$\begin{array}{rcl} crush'\langle 1\rangle\ () & = & e \\ crush'\langle t_1 \times t_2\rangle\ (x_1, x_2) & = & (crush'\langle t_1\rangle\ x_1) \circ (crush'\langle t_2\rangle\ x_2) \\ crush'\langle n\rangle\ x & = & x \text{ for all } n \text{ in } s \end{array}$$

For $zero$, we assume a monad with zero, that is a structure $\langle M, return, \gg\!=, mzero\rangle$.

In Section 3.3, we introduced the terms *type-preserving* and *type-unifying* to describe the classes of algebras of which respectively $idmap$ and $crush$ are representatives. We can now characterize these classes by the result types of the algebras. For a type-preserving algebra $\mathcal{A}$, $\overline{n}(\mathcal{A}) = n$ for all n in s. For a type-unifying algebra $\mathcal{A}$, $\overline{n}(\mathcal{A}) = \tau$ for all n in s, i.e., there is common result type τ independent of the type index. The basic fold algebra $zero$ (or any algebra of the same type) is not restricted to either of these classes. The result types are of the form $\overline{n}(zero) = M\ \alpha$ (with different α for different n), i.e., the result types for the various n in s are only constrained to be monadic.

3.5.5 Algebra combinators

Sections 3.3 and 3.4 featured a number of operators on fold algebras. Algebra $update$ is the most important of these operators. The combinators $unit$, $plus$, and $carried$ were introduced for monadic fold algebras. The definitions of these monadic combinators are similar to those for the basic algebras above. The definition of updating is more involved.

If the datatype system s contains the constructor name c, i.e., if $c(s)$ is defined, $\mathcal{A}[c/f]$ denotes the update of an s-algebra $\mathcal{A}$ at c by a function f. Initially, we require the type of f to be equal to the type of $\mathcal{A}.c$. Then, updating can be defined as follows:

$$\mathcal{A}[c/f].c' = \begin{cases} f, & \text{if } c = c' \\ \mathcal{A}.c', & \text{otherwise} \end{cases}$$

It is easy to verify that the resulting structure is indeed a proper s-algebra with $\overline{n}(\mathcal{A}[c/f]) = \overline{n}(\mathcal{A})$ for all n in s. The condition that the type of f is equal to the type of $\mathcal{A}.c$ is not too restrictive in the presence of an operator for type specialization. We will use $\mathcal{A}[\overline{n}/\tau]$ to denote the instantiation of the result type for n in $\mathcal{A}$ to the type τ. The axiomatization is omitted for brevity. Type specialization is allowed under the condition that τ is more specific than $\overline{n}(\mathcal{A})$, i.e., if there is a substitution to replace type variables by type schemes in $\overline{n}(\mathcal{A})$ such that it becomes equal to τ. Recall that in the Haskell model, fold algebra types are parameterized datatypes (record types), algebra updating is record updating, and type specialization is type parameter instantiation.

3.5.6 Extensions

So far we have restricted ourselves to closed systems of non-parameterized datatypes. For many purposes this is quite sufficient. In the application areas we have in mind, systems of datatypes are derived from syntax definitions, and the class of systems considered so far covers simple BNF notation. Nonetheless, we will now discuss some possibilities for extending our approach to richer classes of datatypes. As will become apparent, such extensions conjure up a wealth of design choices.

Primitive types The system of datatypes of our running example uses the primitive type $String$. Actually, $String$ is not quite primitive in Haskell, but defined as list of $Char$. In fact, for pragmatic reasons one may choose to regard any predefined type as primitive. Our approach can be easily extended to handle primitive types. We extend our grammar as follows:

T	::=	$\cdots \mid P$	-- additional form of type expression
TS	::=	$\cdots \mid P$	-- maintain $TS \supset T$
P			-- primitive types

The axiomatization of algebras can be extended to provide result types and algebra members for primitive types. This allows to write updates for primitive types. There is an alternative way to cover primitive types, where the axiomatization of algebras is not affected. The values of primitive types are just preserved during folding as modeled by the following additional case in the inductive definition of $fold$:

$$fold\langle p\rangle\ \mathcal{A}\, x \quad = \quad x$$

Here, p ranges over P. For values of primitive types, $fold$ acts like the identity. In Haskell, this is done by having instances of the fold function for primitive types, or as in Example 3.2.1, where $fold$ simply does not recurse into $String$.

Parameterized datatypes Covering systems of parameterized datatypes is more challenging. Let us stick to uniform recursion of parameters in the sense of regular datatypes. From an application perspective, such an extension allows us to cover extended BNF notation including optionals (maybe type), iteration (lists), nested alternatives (binary sums, $Either$). Note that nested concatenation is already covered by the products of our basic approach.

Let us first extend our grammar to cope with regular datatypes. The syntactical domain S is extended by a form for definitions of regular datatypes, and a form of type expression is added to represent the application of parameterized regular datatypes.

S	::=	$\cdots \mid R = F$	-- definition of regular datatypes
T	::=	$\cdots \mid R@T$	-- application of regular datatypes
TS	::=	$\cdots \mid R@T$	-- maintain $TS \supset T$
R			-- names of regular datatypes
F			-- regular datatypes (functors)

We assume that F is the syntactical domain for regular datatypes (or their functors).

Parameterized datatypes can be handled in essentially the same manner as non-parameterized ones, i.e., by defining additional result types and algebra members for the fold algebra. However, this extension is not straightforward. The types of the algebra members get more involved. To uniformly handle all instantiations of a particular parameterized datatype in a single algebra member, such members ultimately need to be polytypic functions themselves. Furthermore, it should be possible to enforce specific behavior for particular applications of a regular datatype.

As for primitive types, there is also a way to cope with parameterized datatypes that does not affect the axiomatization of algebras. Parameterized datatypes are folded in a homogeneous way based on the polytypic map function ($pmap$ in [JJ97a]). Consequently, the inductive definition of $fold$ is extended as follows:

$$fold\langle r@t\rangle\ \mathcal{A}\ x \quad = \quad pmap\ (fold\langle t\rangle\ \mathcal{A})\ x$$

Here, r ranges over R. This approach is much easier to formalize. But it is restricted in the sense that updating can not be performed for (constructors of) regular datatypes.

Nested and function types An elaboration to cover nested (rather than just regular) datatypes [BM98, BP99] is not needed for our intended application areas. Nestedness does not commonly occur among large bananas. For similar reasons, function types [MH95] are not considered.

3.6 Concluding remarks

Contributions The advantages of programming with folds (as opposed to general recursion) are well known. We have presented an elaboration for generalized (monadic) folds on systems of mutually recursive datatypes where a separation is made between fold algebras that capture generic functionality and fold algebra updates that implement problem-specific functionality. This separation provides a combination of generic and specific programming which is crucial to make programming with folds as scalable as possible. We identified a number of particular generic fold algebras as well as some algebra combinators for calculating with monadic folds. Furthermore, we showed that generic definitions can be given of these algebras and combinators, and of the other ingredients for programming with updatable folds.

Our approach is relatively lightweight in two important dimensions: it is conceptually simple, and easy to implement. The first claim can be justified by the argument that, essentially, mastering the concept of generalized folds is sufficient to use the approach. The second claim holds for a generator-based approach, where Haskell functions and datatypes are generated for programming with folds. Our generator took us about 0.1 man years development effort. It is fully operational and can be used for serious case studies as the one reported in [KLV00]. To provide a thorough semantics for our approach and to fully integrate the concepts in a functional language is more ambitious. The integration issue raises the question if such an integration can be done by recasting the approach to some existing generic framework or language such as Charity [CS92], PolyP [JJ97a], FISh [Jay99], or Generic Haskell [Hin99]. Such a recasting is not obvious because of the (inherent or current) limitations of the respective languages and approaches. For example, polytypic programming systems do not allow induction over datatype *systems*, as was required for our polytypic definitions of algebra types and algebras in Section 3.5.

Related work Polytypic programming [JJ97a, Hin00] allows for general recursive type-indexed (or even kind-indexed) functions. On the other hand, we require type-indexed algebra types, i.e., a kind of polytypic datatype definition. To understand the pros and cons of these variations, more research is needed. We should mention one interesting observation, where the restriction to folds pays back in a surprising manner, that is non-monadic traversals (say algebras) can be turned into monadic ones. For general recursive functions, such a migration is inherently subject to program transformation [Läm00], or to semantically restrictive and non-trivial type systems [Fil99].

In [Jon95, SAS99] it is discussed how to program with catamorphisms in Haskell in an (almost) generic way. A generic $cata$ is easily defined based on a Haskell class $Functor$ whose $fmap$ member, however, needs to be instantiated by the programmer for *each* datatype. As noted in [SAS99], elaborate coding is to be done to cope with mutually recursive datatypes. A new functor class $Functor\_n$ is needed for each number n of datatypes. This is not a theoretical problem, but a result of Haskell's limited genericity. Note that datatype definitions must be written as functors in order to fit into this scheme. On the positive side, this allows for modularization of the datatypes, algebras, and instances of $Functor$.

The tension between genericity and specificity is a recurring theme. Strategies [VBT99, Vis00a] have been proposed for term rewriting so that separation is possible of generic phenomena (such as traversal schemes and reduction) and specific ones (one-step rewrite rules). However, the approach is untyped. In Chapters 4 and 5, we will define *typed* strategy operators in a functional and an object-oriented setting.

Chapter 4

Typed Combinators for Generic Traversal

In this chapter, we develop a second approach to generic traversal in functional programming. While the approach of the previous chapter was based on updatable generalized folds, this second approach is based on the notion of a functional *strategy*. This approach is more powerful and flexible, but also somewhat further removed from standard functional programming techniques.

A functional strategy is a typeful generic function that can not only be applied to terms of any type, but which also allows mixing generic and type-specific behaviour, and generic traversal into subterms. While the basic building blocks of updatable folds are complete (primitive) traversal schemes, functional strategies are constructed from *one-step* traversal combinators and general recursion. Also, fold are updated per data constructor, but strategies can be specialized per type.

We show how strategies are modeled inside a functional language, and we present a combinator library including generic traversal combinators. We illustrate our technique of programming with functional strategies by an implementation of the *extract method* refactoring for Java.

This chapter is based on [LV02b].

4.1 Introduction

Our domain of interest is program transformation in the context of software re-engineering [CC90, ABFP86, BSV00]. Particular problems include automated refactoring (e.g., removal of duplicated code, or goto elimination) and conversion (e.g., Cobol 74 to 85, or Euro conversion). In this context, the bulk of the functionality consists of traversal over the syntax of the involved languages. Most problems call for various different traversal schemes. The involved syntaxes are typically complex (50-2000 grammar productions), and often one has to cope with evolving languages, diverging dialects, and embedded languages. In such a setting, genericity regarding traversal is indispensable [BSV00, KLV00].

By lack of support for generic term traversal, functional programming suffers from a serious and notoriously ignored scalability problem when applied to program transformation problems. To remedy this situation, we introduce functional *strategies*: generic functions that cannot only (i) be applied to terms of any type, but which also (ii) allow generic traversal into subterms, and (iii) may exhibit non-generic (ad-hoc) behavior for particular types.[1] We show how these strategies can be modeled inside the functional language Haskell,[2] and we present a strategy combinator library that includes traversal combinators.

A generic traversal problem Let us consider a simple traversal problem and its solution. Assume we want to accumulate all the variables on use sites in a given abstract syntax tree of a Java program. We envision a traversal which is independent of the Java syntax except that it must be able to identify Java variables on use sites. Here is a little Java fragment:

```
//print details
System.out.println("name:" + _name);
System.out.println("amount" + amount);
```

For this fragment, the traversal should return the list `["_name","amount"]` of variables on use sites.

Using the techniques to be presented in this chapter, the desired traversal can be modeled with a function of the following type:

$$collectUseVars \quad :: \quad TU\ Maybe\ [String]$$

Here, $TU\ Maybe\ [String]$ is the type of *type-unifying* generic functions which map terms of any type to a list of $String$s. The $Maybe$ monad is used to model partiality. In general, a function f of type $TU\ m\ a$ can be applied to a term of *any* type to yield a result of type a (of a monadic type $m\ a$ to be precise). Besides type-unifying strategies, we will later encounter so-called *type-preserving* strategies where input and output type coincide.

The definition of $collectUseVars$ can be based on a simple and completely generic traversal scheme of the following name and type:

$$collect \quad :: \quad MonadPlus\ m \Rightarrow TU\ m\ [a] \rightarrow TU\ m\ [a]$$

[1]We use the term *generic* in the general sense of type- or syntax-independent, not in the stricter senses of parametric polymorphism or polytypism. In fact, the genericity of functional strategies goes beyond these stricter senses.

[2]Throughout the chapter we use Haskell 98 [Has99], unless stated otherwise.

The strategy combinator *collect* maps a type-unifying strategy intended for identification of collectable entities in a node to a type-unifying strategy performing the actual collection over the entire syntax tree. This traversal combinator is included in our library. We can use the combinator in the following manner to collect Java variables on use sites:

```
collectUseVars             :: TU Maybe [String]
collectUseVars              = collect (monoTU useVar)

useVar                     :: Expression → Maybe [String]
useVar (Identifier i)       = Just [i]
useVar _                    = Nothing
```

The non-generic, monomorphic function *useVar* identifies variable names in Java expressions. To make it suitable as an argument to *collect*, it is turned into a type-unifying generic function by feeding it to the combinator *monoTU*. The resulting traversal *collectUseVars* can be applied to any kind of Java program fragment, and it will return the variables identified by *useVar*. Note that the constructor functions *Just* and *Nothing* are used to construct a value of the *Maybe* datatype to represent the list of identified variables.

Generic functional programming Note that the code above does not mention any of Java's syntactical constructs except the syntax of identifiers relevant to the problem. Traversal over the other constructs is accomplished with the fully generic traversal scheme *collect*. As a consequence of this genericity, the solution to our example program is extremely concise and declarative. In general, functional strategies can be employed in a scalable way to construct programs that operate on large syntaxes. In the sequel, we will demonstrate how generic combinators like *collect* are defined and how they are used to construct generic functional programs that solve non-trivial program transformation problems.

Structure of the chapter In Section 4.2 we model strategies with abstract data types (ADTs) to be implemented later, and we explain the primitive and defined strategy combinators offered by our strategy library. In Section 4.3, we illustrate the utility of generic traversal combinators for actual programming by an implementation of an automated program refactoring. In Section 4.4, we study two implementations for the strategy ADTs, namely an implementation based on a universal term representation, and an implementation that relies on rank-2 polymorphism and type case. The chapter is concluded in Section 4.5.

4.2 A strategy library

We present a library for generic programming with strategies. To this end, we introduce ADTs with primitive combinators for strategies (i.e., generic functions).

Strategy types (opaque)		
data $Monad\ m$	$\Rightarrow$	$TP\ m = \ldots\ abstract$
data $Monad\ m$	$\Rightarrow$	$TU\ m\ a = \ldots\ abstract$
Strategy application		
$applyTP :: (Monad\ m, Term\ t)$	$\Rightarrow$	$TP\ m \rightarrow t \rightarrow m\ t$
$applyTU :: (Monad\ m, Term\ t)$	$\Rightarrow$	$TU\ m\ a \rightarrow t \rightarrow m\ a$
Strategy construction		
$polyTP :: Monad\ m$	$\Rightarrow$	$(\forall x.\ x \rightarrow m\ x) \rightarrow TP\ m$
$polyTU :: Monad\ m$	$\Rightarrow$	$(\forall x.\ x \rightarrow m\ a) \rightarrow TU\ m\ a$
$adhocTP :: (Monad\ m, Term\ t)$	$\Rightarrow$	$TP\ m \rightarrow (t \rightarrow m\ t) \rightarrow TP\ m$
$adhocTU :: (Monad\ m, Term\ t)$	$\Rightarrow$	$TU\ m\ a \rightarrow (t \rightarrow m\ a) \rightarrow TU\ m\ a$
Sequential composition		
$seqTP :: Monad\ m$	$\Rightarrow$	$TP\ m \rightarrow TP\ m \rightarrow TP\ m$
$letTP :: Monad\ m$	$\Rightarrow$	$TU\ m\ a \rightarrow (a \rightarrow TP\ m) \rightarrow TP\ m$
$seqTU :: Monad\ m$	$\Rightarrow$	$TP\ m \rightarrow TU\ m\ a \rightarrow TU\ m\ a$
$letTU :: Monad\ m$	$\Rightarrow$	$TU\ m\ a \rightarrow (a \rightarrow TU\ m\ b) \rightarrow TU\ m\ b$
Choice		
$choiceTP :: MonadPlus\ m$	$\Rightarrow$	$TP\ m \rightarrow TP\ m \rightarrow TP\ m$
$choiceTU :: MonadPlus\ m$	$\Rightarrow$	$TU\ m\ a \rightarrow TU\ m\ a \rightarrow TU\ m\ a$
Traversal combinators		
$allTP :: Monad\ m$	$\Rightarrow$	$TP\ m \rightarrow TP\ m$
$oneTP :: MonadPlus\ m$	$\Rightarrow$	$TP\ m \rightarrow TP\ m$
$allTU :: (Monad\ m, Monoid\ a)$	$\Rightarrow$	$TU\ m\ a \rightarrow TU\ m\ a$
$oneTU :: MonadPlus\ m$	$\Rightarrow$	$TU\ m\ a \rightarrow TU\ m\ a$

Figure 4.1: Primitive strategy combinators.

For the moment, we consider the representation of strategies as opaque since different models are possible as we will see in Section 4.4. The primitive combinators cover concepts we are used to for ordinary functions, namely application and sequential composition. There are further important facets of strategies, namely partiality or non-determinism, and access to the immediate subterms of a given term. Especially the latter facet makes clear that strategies go beyond parametric polymorphism. A complete overview of all primitive strategy combinators is shown in Figure 4.1. In the running text we will provide definitions of a number of defined strategies, including some traversal schemes.

4.2.1 Strategy types and application

There are two kinds of strategies. Firstly, the ADT $TP\ m$ models type-preserving strategies where the result of a strategy application to a term of type t is of type $m\ t$. Secondly, the ADT $TU\ m\ a$ models type-unifying strategies where the result of strategy application is always of type $m\ a$ regardless of the type of the input term. These contracts are expressed by the types of the corresponding combinators $applyTP$ and $applyTU$ for strategy application (*cf.* Figure 4.1). In both cases, m is a monad parameter [Wad92] to deal with effects in strategies such as state passing or non-determinism. Also note that we do not apply strategies to arbitrary types but only to instances of the class $Term$ for term types. This is sensible since we ultimately want to traverse into subterms.

The strategy application combinators serve to turn a generic functional strategy into a non-generic function which can be applied to a term of a specific type. Recall that the introductory example is a type-unifying traversal with the result type $[String]$. It can be applied to a given Java class declaration $myClassDecl$ of type $ClassDeclaration$ as follows:

```
applyTU collectUseVars myClassDecl :: Maybe [String]
```

Prerequisite for this code to work is that an instance of the class $Term$ is available for $ClassDeclaration$. This issue will be taken up in Section 4.4.

4.2.2 Strategy construction

There are two ways to construct strategies from ordinary functions. Firstly, one can turn a parametric polymorphic function into a strategy (*cf.* $polyTP$ and $polyTU$ in Figure 4.1). Secondly, one can *update* a strategy to apply a monomorphic function for a given type to achieve type-dependent behaviour (*cf.* $adhocTP$ and $adhocTU$). In other words, one can dynamically provide ad-hoc cases for a strategy. Let us first illustrate the construction of strategies from parametric polymorphic functions:

```
identity :: Monad m ⇒ TP m          build   :: Monad m ⇒ a → TU m a
identity =  polyTP return           build a =  polyTU (const (return a))
```

The type-preserving strategy $identity$ denotes the generic (and monadic) identity function. The type-unifying strategy $build\ a$ denotes the generic function which returns a regardless of the input term. As a consequence of parametricity [Wad89], there are no further ways to inhabit the argument types of $polyTP$ and $polyTU$, unless we rely on a specific instance of m (see $failTU$ below).

The second way of strategy construction, i.e., with the *adhoc* combinators, allows us to go beyond parametric polymorphism. Given a strategy, we can provide an ad-hoc case for a specific type. Here is a simple example:

```
gnot   :: Monad m ⇒ TP m
gnot   =  adhocTP identity (return ∘ not)
```

The strategy $gnot$ is applicable to terms of any type. It will behave like $identity$ most of the time, but it will perform Boolean negation when faced with a Boolean. Such type cases are crucial to assemble traversal strategies that exhibit specific behaviour for certain types of the traversed syntax.

4.2.3 Sequential composition

Since the strategy types are opaque, sequential composition has to be defined as a primitive concept. This is in contrast to ordinary functions where one can define function composition in terms of λ-abstraction and function application. Consider the following parametric polymorphic forms of sequential composition:

$$\begin{array}{lcl} g \circ f & = & \lambda x \rightarrow g\ (f\ x) \\ f\ `mseq`\ g & = & \lambda x \rightarrow f\ x \ggg\!= g \\ f\ `mlet`\ g & = & \lambda x \rightarrow f\ x \ggg\!= \lambda y \rightarrow g\ y\ x \end{array}$$

The first form describes ordinary function composition. The second form describes the monadic variation. The third form can be regarded as a let-expression with a free variable x. An input for x is passed to both f and g, and the result of the first application is fed to the second function. The latter two polymorphic forms of sequential composition serve as prototypes of the strategic combinators for sequential composition. The strategy combinators $seqTP$ and $seqTU$ of Figure 4.1 correspond to $mseq$ lifted to the strategy level. Note that the first strategy is always a type-preserving strategy. The strategy combinators $letTP$ and $letTU$ are obtained by lifting $mlet$. Note that the first strategy is always a type-unifying strategy.

Recall that the $poly$ combinators could be used to lift an ordinary parametric polymorphic function to a strategy. We can not just use $poly$ to lift the prototypes for sequential composition because they are function *combinators*. For this reason, we supply the combinators for sequential composition as primitives of the ADTs, and we postpone their definition to Section 4.4.

Let us illustrate the utility of $letTU$. We want to lift a binary operator o to the level of type-unifying strategies by applying two argument strategies to the same input term and combining their intermediate results by o. Here is the corresponding strategy combinator:

$$\begin{array}{lcl} comb & :: & Monad\ m \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow TU\ m\ a \rightarrow TU\ m\ b \rightarrow TU\ m\ c \\ comb\ o\ s\ s' & = & s\ `letTU`\ \lambda a \rightarrow s'\ `letTU`\ \lambda b \rightarrow build\ (o\ a\ b) \end{array}$$

Thus, the result of the first strategy argument s is bound to the variable a. Then, the result of the second strategy argument s' is bound to b. Finally, a and b are combined with the operator o, and the result is returned by the $build$ combinator which was defined Section 4.2.2.

4.2.4 Partiality and non-determinism

Instead of the simple class $Monad$ we can also consider strategies w.r.t. the extended class $MonadPlus$ with the members $mplus$ and $mzero$. This provides us with means to express partiality and non-determinism. It is often useful to consider strategies which might potentially fail. The following ordinary function combinator is the prototype for the *choice* combinators in Figure 4.1.

```
f `mchoice` g     = λx → (f x) `mplus` (g x)
```

As an illustration let us define three simple strategy combinators which contribute to the construction of the introductory example.

```
failTU      :: MonadPlus m ⇒ TU m x
failTU      =  polyTU (const mzero)

monoTU      :: (Term a, MonadPlus m) ⇒ (t → m a) → TU m a
monoTU f    =  adhocTU failTU f

tryTU       :: (MonadPlus m, Monoid a) ⇒ TU m a → TU m a
tryTU s     =  s `choiceTU` (build mempty)
```

The strategy *failTU* denotes unconditional failure. The combinator *monoTU* updates failure by a monomorphic function f, using the combinator *adhocTU*. That is, the resulting strategy fails for all types other than f's argument type. If f is applicable, then the strategy indeed resorts to f. The combinator *tryTU* allows us to recover from failure in case we can employ a neutral element *mempty* of a monoid.

Recall that the *monoTU* combinator was used in the introductory example to turn the non-generic, monomorphic function *useVar* into a type-unifying strategy. This strategy will fail when applied to any type other than *Expression*.

4.2.5 Traversal combinators

A challenging facet of strategies is that they might descend into terms. In fact, any program transformation or program analysis involves traversal. If we want to employ genericity for traversal, corresponding basic combinators are indispensable. The *all* and *one* combinators in Figure 4.1 process all or just one of the *immediate* subterms of a given term, respectively. The combinators do not just vary with respect to quantification but also for the type-preserving and the type-unifying case. The type-preserving combinators *allTP* and *oneTP* preserve the outermost constructor for the sake of type-preservation. Dually, the type-unifying combinators *allTU* and *oneTU* unwrap the outermost constructor in order to migrate to the unified type. More precisely, *allTU* reduces all pre-processed children by the binary operation *mappend* of a monoid whereas *oneTU* returns the result of processing one child. The *all* and *one* combinators have been adopted from the untyped language Stratego [VBT99] for strategic term rewriting.

We are now in the position to define the traversal scheme *collect* from the introduction. We first define a more parametric strategy *crush* which performs a deep reduction by employing the operators of a monoid parameter. Then, the strategy *collect* is nothing more than a type-specialized version of *crush* where we opt for the list monoid.

```
crush       :: (MonadPlus m, Monoid a) ⇒ TU m a → TU m a
crush s     =  comb mappend (tryTU s) (allTU (crush s))

collect     :: MonadPlus m ⇒ TU m [a] → TU m [a]
collect s   =  crush s
```

Note that the *comb* combinator is used to combine the result of *s* on the current node with the result of crushing the subterms. The *tryTU* combinator is used to recover from possible failure of *s*. In the introductory example, this comes down to recovery from failure of *monoTU useVar* at non-*Expression* nodes, and at nodes of type *Expression* for which *useVar* returns *Nothing*.

4.2.6 Some defined combinators

We can subdivide defined combinators into two categories, one for the control of strategies, and another for traversal schemes. Let us discuss a few examples of defined combinators. Here are some representatives of the category for the control of strategies:

```
repeatTP        :: MonadPlus m ⇒ TP m → TP m
repeatTP s      =  tryTP (seqTP s (repeatTP s))

ifthenTP        :: Monad m ⇒ TP m → TP m → TP m
ifthenTP f g    =  (f `seqTU` (build ())) `letTP` (const g)

notTP           :: MonadPlus m ⇒ TP m → TP m
notTP s         =  ((s `ifthenTU` (build True)) `choiceTU` (build False))
                   `letTP` λb → if b then failTP else identity

afterTU         :: Monad m ⇒ (a → b) → TU m a → TU m b
afterTU f s     =  s `letTU` λa → build (f a)
```

The combinator *repeatTP* applies its argument strategy as often as possible. As an aside, a type-unifying counter-part of this combinator would justly not be typeable. The combinator *ifthenTP* precedes the application of a strategy by a guarding strategy. The guard determines whether the guarded strategy is applied at all. However, the guarded strategy is applied to the original term (as opposed to the result of the guarding strategy). The combinator *notTP* models negation by failure. The combinator *afterTU* adapts the result of a type-unifying traversal by an ordinary function.

Let us also define a few traversal schemes (in addition to *crush* and *collect*):

```
bu              :: Monad m ⇒ TP m → TP m
bu s            =  (allTP (bu s)) `seqTP` s

oncetd          :: MonadPlus m ⇒ TP m → TP m
oncetd s        =  s `choiceTP` (oneTP (oncetd s))

select          :: MonadPlus m ⇒ TU m a → TU m a
select s        =  s `choiceTU` (oneTU (select s))

selectenv       :: MonadPlus m ⇒ e → (e → TU m e)
                →  (e → TU m a) → TU m a
selectenv e s' s = s' e `letTU` λe' →
                   (s e) `choiceTU` (oneTU (selectenv e' s' s))
```

All these schemes deal with recursive traversal. The combinator *bu* serves for unconstrained type-preserving bottom-up traversal. The argument strategy has to succeed for every node if the traversal is to succeed. The combinator *oncetd* serves for type-preserving top-down traversal where the argument strategy is tried until it succeeds once. The traversal fails if the argument strategy fails for all nodes. The type-unifying combinator $select$ searches in top-down manner for a node which can be processed by the argument strategy. Finally, the combinator $selectenv$ is an elaboration of $select$ to accomplish explicit environment passing. The first argument strategy serves for updating the environment before descending into the subterms. As will be demonstrated in the upcoming section, traversal schemes like these can serve as building blocks for program transformations.

4.3 Application: Refactoring

Refactoring [Fow99] is the process of step-wise improving the internal structure of a software system without altering its external behaviour. The *extract method refactoring* [Fow99, p. 110] is a well-known example of a basic refactoring step. To demonstrate the technique of programming with strategy combinators, we will implement the extract method refactoring for Java.

4.3.1 The *extract method* refactoring

In brief, the extract method refactoring is described as follows:

> *Turn a code fragment that can be grouped together into a reusable method whose name explains the purpose of the method.*

For instance, the last two statements in the following method can be grouped into a method called `printDetails`.

```
void printOwning(double amount) {
  printBanner ();
  //print details
  System.out.println("name:" + _name);
  System.out.println("ammount" + amount);
}
```

$$\Downarrow$$

```
void printOwning(double amount) {
  printBanner ();
  printDetails(amount);
}
void printDetails(double amount) {
  System.out.println("name:" + _name);
  System.out.println("amount" + amount);
}
```

Note that the local variable `amount` is turned into a parameter of the new method, while the instance variable `_name` is not. Note also, that the *extract method* refactoring is valid only for a code fragment that does not contain any return statements or assignments to local variables.

4.3.2 Design

To implement the *extract method* refactoring, we need to solve a number of subtasks.

Legality check The focused fragment must be analysed to ascertain that it does not contain any return statements or assignments to local variables. The latter involves detection of variables in the fragment that are defined (assigned into), but not declared (i.e., free *defined* variables).

Generation The new method declaration and invocation need to be generated. To construct their formal and actual parameter lists, we need to collect those variables that are used, but not declared (i.e., free *used* variables) from the focused fragments, with their types.

Transformation The focused fragment must be replaced with the generated method invocation, and the generated method declaration must be inserted in the class body.

These subtasks need to be performed at specific moments during a traversal of the abstract syntax tree. Roughly, our traversal will be structured as follows:

1. Descend to the class declaration in which the method with the focused fragment occurs.

2. Descend into the method with the focused fragment to (i) check the legality of the focused fragment, and (ii) return both the focused fragment and a list of typed free variables that occur in the focus.

3. Descend again to the focus to replace it with the method invocation that can now be constructed from the list of typed free variables.

4.3.3 Implementation with strategies

Our solution is shown in Figures 4.2 through 4.4.

Free variable analysis As noted above, we need to perform two kinds of free variable collection: variables used but not declared, and variables defined but not declared. Furthermore, we need to find the types of these free variables. Using

```
typed_free_vars :: (MonadPlus m, Eq v)
                  ⇒ [(v, t)] → TU m [v] → TU m [(v, t)] → TU m [(v, t)]
typed_free_vars env getvars declvars
  = afterTU (flip appendMap env) (tryTU declvars) ‘letTU‘ λenv′ →
      choiceTU (afterTU (flip selectMap env′) getvars)
               (comb diffMap (allTU (typed_free_vars env′ getvars declvars))
                             (tryTU declvars))
```

Figure 4.2: A generic algorithm for extraction of free variables with their declared types.

```
useVar (Identifier i)       =  return [i]
useVar _                    =  mzero
defVar (Assignment i _)     =  return [i]
declVars                    :: MonadPlus m ⇒ TU m [(Identifier, Type)]
declVars                    =  adhocTU (monoTU declVarsBlock) declVarsMeth
      where declVarsBlock (BlockStatements vds _) = return vds
            declVarsMeth (MethodDecl _ _ (FormalParams fps) _) = return fps
freeUseVars env = afterTU nubMap (typed_free_vars env (monoTU useVar) declVars)
freeDefVars env = afterTU nubMap (typed_free_vars env (monoTU defVar) declVars)
```

Figure 4.3: Instantiations of the generic free variable algorithm for Java.

strategies, we can implement free variable collection in an extremely generic fashion. Figure 4.2 shows a generic free variable collection algorithm. This algorithm was adapted from an untyped rewriting strategy in [Vis00a]. It is parameterized with (i) an initial type environment env, (ii) a strategy $getvars$ which selects any variables that are used in a certain node of the AST, and (iii) a strategy $declvars$ which selects declared variables with their types. Note that no assumptions are made with respect to variables or types, except that equality is defined on variables so they can appear as keys in a map.

The algorithm basically performs a top-down traversal. It is not constructed by reusing one of the defined traversal combinators from our library, but directly in terms of the primitive combinator $allTU$. At a given node, first the incoming type environment is extended with any variables declared at this node. Second, either the variables used at the node are looked-up in the type environment and returned with their types, or, if the node is not a use site, any declared variables are subtracted from the collection of free variables found in the children (cf. $allTU$). Note that the algorithm is typeful, and fully generic. It makes ample use of library combinators, such as $afterTU$, $letTU$ and $comb$.

As shown in Figure 4.3, this generic algorithm can be instantiated to the two kinds of free variable analyses needed for our case. The functions $useVar$, $defVar$, and $declVars$ are the Java-specific ingredients that are needed. They determine the used, defined, and declared variables of a given node, respectively. We use them

```
extractMethod  ::  (Term t, MonadPlus m) ⇒ t → m t
extractMethod prog
               =   applyTP (oncetd (monoTP extrMethFromCls)) prog
extrMethFromCls
               ::  MonadPlus m ⇒ ClassDeclaration → m ClassDeclaration
extrMethFromCls (ClassDecl fin nm sup fs cs ds)
               =   do (pars, body) ← ifLegalGetParsAndBody ds
                      ds′ ← replaceFocus pars (ds ++ [constructMethod pars body])
                      return (ClassDecl fin nm sup fs cs ds′)
ifLegalGetParsAndBody
               ::  (Term t, MonadPlus m) ⇒ t → m ([([Char], Type)], Statement)
ifLegalGetParsAndBody ds
               =   applyTU (selectenv [] appendLocals ifLegalGetParsAndBody1) ds
                   where ifLegalGetParsAndBody1 env
                           = getFocus ‘letTU‘ λs →
                             ifthenTU (isLegal env)
                             (freeUseVars env ‘letTU‘ λpars →
                              build (pars, s))
                         appendLocals env
                           = comb appendMap (tryTU declVars) (build env)
replaceFocus   ::  (Term t, MonadPlus m) ⇒ [(Identifier, Type)] → t → m t
replaceFocus pars ds
               =   applyTP (oncetd (replaceFocus1 pars)) ds
                   where replaceFocus1 pars
                           = getFocus ‘letTP‘ λ_ →
                             monoTP (const (return (constructMethodCall pars)))
isLegal        ::  MonadPlus m ⇒ [([Char], Type)] → TP m
isLegal env    =   freeDefVars env ‘letTP‘ λenv′ →
                   if null env′ then notTU (select getReturn) else failTP
getFocus       ::  MonadPlus m ⇒ TU m Statement
getFocus       =   monoTU (λs → case s of (StatFocus s′) → return s′
                                          _ → mzero)
getReturn      ::  MonadPlus m ⇒ TU m (Maybe Expression)
getReturn      =   monoTU (λs → case s of (ReturnStat x) → return x
                                          _ → mzero)
```

Figure 4.4: Implementation of the *extract method* refactoring.

to instantiate the generic free variable collector to construct *freeUseVars*, and *freeDefVars*.

Method extraction The remainder of the extract method implementation is shown in Figure 4.4. The main strategy *extractMethod* performs a top-down traversal to the class level, where it calls *extrMethFromCls*. This latter function first obtains parameters and body with *ifLegalGetParsAndBody*, and then replaces the focus with *replaceFocus*. Code generation is performed by two functions *constructMethod* and *constructMethodCall*. Their definitions are trivial and not shown here. The extraction of the candidate body and parameters for the new method is performed in the same traversal as the legality check. This is a top-down traversal with environment propagation. During descent, the environment is extended with declared variables. When the focus is reached, the legality check is performed. If it succeeds, the free used variables of the focused fragment are determined. These variables are paired with the focused fragment itself, and returned. The legality check itself is defined in the strategy $\mathit{isLegal}$. It fails when the collection of variables that are defined but not declared is non-empty, or when a return statement is recognized in the focus. The replacement of the focus by a new method invocation is defined by the strategy $\mathit{replaceFocus}$. It performs a top-down traversal. When the focus is found, the new method invocation is generated and the focus is replaced with it.

4.4 Models of strategies

We have explained what strategy combinators are, and we have shown their utility. Let us now change the point of view, and explain some options for the implementation of the strategy ADTs including the primitives. Recall that functional strategies have to meet the following requirements. Firstly, they need to be applicable to values of any term type. Secondly, they have to allow for updating in the sense that type-specific behaviour can be enforced. Thirdly, they have to be able to descend into terms. The first model we discuss uses a universal term representation. The second model employs rank-2 polymorphism with type case.

4.4.1 Strategies as functions on a universal term representation

One way to meet the requirements on functional strategies is to rely on a universal representation of terms of algebraic datatypes. Such a representation can easily be constructed in any functional language in a straightforward manner. The challenge is to hide the employment of the universal representation to rule out inconsistent representations, and to relieve the programmer of the burden to deal explicitly with representations rather than ordinary values and functions.

The following declarations set up a representation type $\mathit{TermRep}$, and the ADTs for strategies are defined as functions on $\mathit{TermRep}$ wrapped by datatype constructors MkTP and MkTU:

```
type TypeId        = String
type ConstrId      = String
data TermRep       = TermRep TypeRep ConstrId [TermRep]
data TypeRep       = TypeRep TypeId [TypeRep]
newtype TP m       = MkTP (TermRep → m TermRep)
newtype TU m a     = MkTU (TermRep → m a)
```

Thus, a universal value consists of a type representation (for a potentially parameterized data type), a constructor identifier, and the list of universal values corresponding to the immediate subterms of the encoded term (if any). The strategy ADTs are made opaque by simply not exporting the constructors $MkTP$ and $MkTU$. To mediate between $TermRep$ and specific term types, we place members for implosion and explosion in a class $Term$.

```
class Term t where
  explode    ::  t → TermRep
  implode    ::  TermRep → t
```

The instances for a given term type follow a trivial scheme, as illustrated by the following two sample equations for Java $Identifier$s.

```
explode (Identifier i) = TermRep (TypeRep "Expr" []) "Identifier" [explode i]
implode (TermRep _ "Identifier" [i]) = Identifier (implode i)
```

In fact, we extended the DrIFT tool [Win97] to generate such instances for us (see Section 4.5). For a faithful universal representation it should hold that explosion can be reversed by implosion. Implosion is potentially a partial operation. One could use the $Maybe$ monad for the result to enable recovery from an implosion problem. By contrast, we rule out failure of implosion in the first place by hiding the representation of strategies behind the primitive combinators defined below. It would be easy to prove that all functions on $TermRep$ which can be defined in terms of the primitive combinators are implosion-safe.

The combinators $polyTP$ and $polyTU$ specialize their polymorphic argument to a function on $TermRep$. Essentially, the combinators for sequential composition and choice are also defined by specialisation of the corresponding prototypes *mseq*, *mlet*, and *mchoice*. In addition, we need to unwrap the constructors $MkTP$ and $MkTU$ from each argument and to re-wrap the result.

```
                         seqTP f g = MkTP ((unTP f) `mseq` (unTP g))
polyTP f = MkTP f        seqTU f g = MkTU ((unTP f) `mseq` (unTU g))
polyTU f = MkTU f        letTP f g = MkTP ((unTU f) `mlet` (λa → unTP (g a)))
unTP (MkTP f) = f        letTU f g = MkTU ((unTU f) `mlet` (λa → unTU (g a)))
unTU (MkTU f) = f        choiceTP f g = MkTP ((unTP f) `mchoice` (unTP g))
                         choiceTU f g = MkTU ((unTU f) `mchoice` (unTU g))
```

The combinators for strategy application and updating are defined as follows:

```
applyTP s t   =  unTP s (explode t) >>= λt' → return (implode t')
applyTU s t   =  unTU s (explode t)
adhocTP s f   =  MkTP (λu → if applicable f u
                               then f (implode u) >>= λt → return (explode t)
                               else unTP s u)
adhocTU s f   =  MkTU (λu → if applicable f u
                               then f (implode u)
                               else unTU s u)
```

As for application, terms are always first exploded to $TermRep$ before the function underlying a strategy can be applied. This is because strategies are functions on $TermRep$. In the case of a type-preserving strategy, the result of the application also needs to be imploded afterwards. As for update, we use a type test (*cf.* $applicable$) to check if the given universal value is of the specific type handled by the update. For brevity, we omit the definition of $applicable$ but it simply compares type representations. If the type test succeeds, the corresponding implosion is performed so that the specific function can be applied. If the type test fails, the generic default strategy is applied.

The primitive traversal combinators are particularly easy to define for this model. Recall that these combinators process in some sense the immediate subterms of a given term. Thus, we can essentially perform list processing. The following code fragment defines a helper to apply a list-processing function on the immediate subterms. We also show the implementation of the primitive $allTP$ which directly employs the standard monadic map function $mapM$.

```
applyOnKidsTP     ::  Monad m ⇒ ([TermRep] → m [TermRep]) → TP m
applyOnKidsTP s   =   MkTP (λ(TermRep sort con ks) →
                             s ks ≫= λks′ → return (TermRep sort con ks′))
allTP s           =   applyOnKidsTP (mapM (unTP s))
```

4.4.2 Strategies as rank-2 polymorphic functions with type case

Instead of defining strategies as functions on a universal representation type, we can also define them as a kind of polymorphic functions being directly applicable to terms of the algebraic datatypes. But, since strategies can be passed as arguments to strategy combinators, we need to make use of *rank-2 polymorphism*.[3] The following declarations define $TP\ \ m$ and $TU\ \ m\ \ a$ in terms of universally quantified components of datatype constructors. This form of wrapping is the Haskell approach to deal with rank-2 polymorphism while retaining decidability of type inference [Jon97].

```
newtype Monad m ⇒ TP m       =  MkTP (∀t. Term t ⇒ t → m t)
newtype Monad m ⇒ TU m a     =  MkTU (∀t. Term t ⇒ t → m a)
```

Note that the functions which model strategies are not simply universally quantified, but the domain is also constrained to be an instance of the class $Term$. The following model-specific term interface provides traversal and ad-hoc primitives to meet the other requirements on strategies.

```
class Update t ⇒ Term t where
 allTP′ :: Monad m                       ⇒  TP m → t → m t
 oneTP′ :: MonadPlus m                   ⇒  TP m → t → m t
 allTU′ :: (Monad m, Monoid a)           ⇒  TU m a → t → m a
 oneTU′ :: MonadPlus m                   ⇒  TU m a → t → m a
 adhocTP′ :: (Monad m, Update t′)        ⇒  (t′ → m t′) → (t → m t) → (t′ → m t′)
 adhocTU′ :: (Monad m, Update t′)        ⇒  (t′ → m a) → (t → m a) → (t′ → m a)
```

[3]Rank-2 polymorphism is not part of Haskell 98, but available in the required form as an extension of the Hugs and GHC implementations.

We use primed names because the members are only rank-1 prototypes which still need to be lifted by wrapping and unwrapping. The term interface is instantiated by defining the primitives for all possible term types.

The definitions of the traversal primitives are as simple as the definitions of the *implode* and *explode* functions for the previous model. They are not shown for brevity. To define $adhocTP'$ and $adhocTU'$ for each datatype, an additional technique is needed: we model strategy update as a type case [DRW95, CWM99]. The instances of the $Update$ class, mentioned in the context of class $Term$, implement this type case via an encoding technique for Haskell inspired by [Wei00]. In essence, this technique involves two members $dUpdTP$ and $dUpdTU$ in the $Update$ class for each datatype d. These members for d select their second argument in the instance for d, and default to their first argument in all other instances.

Given the rank-1 prototypes, the derivation of the actual rank-2 primitive combinators is straightforward:

$$\begin{array}{ll}
applyTP\ s\ t = (unTP\ s)\ t & allTP\ s = MkTP\ (allTP'\ s) \\
applyTU\ s\ t = (unTU\ s)\ t & oneTP\ s = MkTP\ (oneTP'\ s) \\
adhocTP\ s\ f = MkTP\ (adhocTP'\ (unTP\ s)\ f) & allTU\ s = MkTU\ (allTU'\ s) \\
adhocTU\ s\ f = MkTU\ (adhocTU'\ (unTU\ s)\ f) & oneTU\ s = MkTU\ (oneTU'\ s)
\end{array}$$

Note that application does not involve conversion with *implode* and *explode*, as in the previous model, but only unwrapping of the rank-2 polymorphic function. As for sequential composition, choice, and the *poly* combinators, the definitions from the previous model carry over.

4.4.3 Trade-offs and alternatives

The model relying on a universal term representation is simple and does not rely on more than parametric polymorphism and class overloading. It satisfies extensibility in the sense that for each new datatype, one can provide a new instance of $Term$ without invalidating previous instances. The second model is slightly more involved. But it is more appealing in that no conversion is needed, because strategies are simply functions on the datatypes themselves, instead of on a representation of them. However, extensibility is compromised, as the employed coding scheme for type cases involves a closed world assumption. That is, the encoding technique for type case requires a class $Update$ which has members for each datatype. Note that these trade-offs are Haskell-specific. In a different language, e.g., a language with built-in type case, strategies would be supported via different models. In fact, a simple language extension could support strategies directly.

Regardless of the model, it is intuitively clear that a full traversal visiting all nodes should use time linear in the size of the term, assuming a constant node-processing complexity. Both models expose this behaviour. However, if a traversal stops somewhere, no overhead for non-traversed nodes should occur. The described universal representation is problematic is this respect since the non-traversed part below the stop node will have to be imploded before the node can be processed. Thus, we suffer a penalty linear in the number of non-traversed nodes. Similarly, implosion is needed when a strategy is applied which involves an ad-hoc update. This is because a universal representation has to be imploded before a non-generic function can be applied on a node of a specific datatype. Short of switching to the second model, one can remedy these performance problems by adopting a more involved universal representation. The overall idea is to use dynamic typing [ACPP91] and to do stepwise explosion by need, that is, only if the application of a traversal primitive requires it.

4.5 Conclusion

Functional software re-engineering Without appropriate technology large-scale software maintenance projects cannot be done cost-effectively within a reasonable time-span, or not at all [CC90, DKV99, BSV00]. Currently, declarative *re*-technologies are usually based on term rewriting frameworks and attribute grammars. There are hardly (published) attempts to employ functional programming for the development of large-scale program transformation systems. One exception is AnnoDomini [EHM$^+$99] where SML is used for the implementation of a Y2K tool. The traversal part of AnnoDomini is kept to a reasonable size by a specific normalisation that gets rid of all syntax not relevant for this Y2K approach. In general, re-engineering requires generic traversal technology that is applicable to the full syntax of the language at hand [BSV00]. In [KLV00], we describe an architecture for functional transformation systems and a corresponding case study concerned with a data expansion problem. The architecture addresses the important issues of scalable parsing and pretty-printing, and employs an approach to generic traversal based on combinators for updatable generalized folds (see Chapter 3). The functional strategies described in the present chapter provide a more lightweight and more generic solution than folds, and can be used instead.

Of course, our techniques are not only applicable to software re-engineering problems, but generally to all areas of language and document processing where type-safe generic traversal is desirable. For example, our strategy combinators can be used for XML processing where, in contrast to the approaches presented in [WR99], document processors can at once be typed and generic.

Generic functional programming Related forms of genericity have been proposed elsewhere. These approaches are not just more complex than ours, but they are even insufficient for a faithful encoding of the combinators we propose. With intensional and extensional polymorphism [DRW95, CWM99] one can also encode type-parametric functions where the behaviour is defined via a run-time type case. However, as-is the corresponding systems do not cover algebraic data types, but only products, function space, and basic data types. With polytypic programming (*cf*. PolyP and Generic Haskell [JJ97a, Hin99]), one can define functions by induction on types. However, polytypic functions are not first class citizens: due to the restriction that polytypic parameters are quantified at the top level, polytypic *combinators* cannot be defined. Also, in a polytypic definition, though one can provide fixed ad-hoc cases for specific data types, an $adhoc$ combinator is absent. It may be conceivable that polytypic programming is generalized to cover the functionality of our strategies, but the current chapter shows that strategies can be modelled within a language like Haskell without type-system extensions.

The origins of functional strategies The term ‘strategy’ and our conception of generic programming were largely influenced by strategic term rewriting [Pau83, LV97, Bor98, VBT99, Läm02b]. In particular, the overall idea to define traversal schemes in terms of basic generic combinators like *all* and *one* has been adopted from the untyped language Stratego [VBT99] for strategic term rewriting. This idea is equally present in the rewrite strategy language of the ELAN system [Bor98]. Our contribution is that we integrate this idea with typed and higher-order functional programming. In fact, Stratego was not defined with typing in mind. Integration of rewriting and functional programming concepts is also an objective of the Rewriting Calculus [CK99, CKL01, BKKR01], and we hope that our treatment of typed generic traversal will help its further development.

Chapter 5

Visitor Combination and Traversal Control

We now turn to the support of generic traversal in the context of object-oriented programming. This chapter introduces the essential notion of a *visitor combinator*, which can be seen as the object-oriented counterpart of a strategy. In Chapters 6 and 7, the traversal support offered by visitor combinators will be integrated with parsing support, and will be applied in the construction of language processing tools.

The *Visitor* design pattern allows the encapsulation of polymorphic behavior outside the class hierarchy on which it operates. A common application of *Visitor* is the encapsulation of tree traversals. Unfortunately, visitors resist composition and allow little traversal control. To remove these limitations, we introduce visitor *combinators*. These are implementations of the visitor interface that can be used to *compose* new visitors from given ones. The set of combinators we propose includes *traversal* combinators that can be used to obtain full traversal control. A clean separation can be made between the generic parts of the combinator set and the parts that are specific to a particular class hierarchy. The generic parts form a reusable framework. The specific parts can be generated from a (tree) grammar. Due to this separation, programming with visitor combinators becomes a form of *generic programming* with significant reuse of (visitor) code.

This chapter is based on [Vis01b].

5.1 Introduction

Language processing involves tree traversal. For instance, program analysis and transformation require traversal of syntax trees. In the object-oriented paradigm, tree traversal can be performed in accordance with a particular variant of the Visitor design pattern [GHJV94]. In this approach, the tree to be traversed is represented according to the Composite pattern, the actions to be performed at tree nodes are encapsulated in a visitor class, and iteration over the tree is performed either by the visitor or by the accept methods in the tree classes. Several parser and visitor generation tools exist that support tree traversal with visitors (e.g. Java Tree Builder, SableCC [GH98]).

Unfortunately, tree traversal with visitors suffers from two main limitations. The first limitation is lack of traversal control. The tree traversal strategy is either hard-wired into the accept methods, or entangled in the visitor code. In the first case, traversal control is absent, or limited to selecting one out of a few predefined strategies. In the second case, traversal control is limited to overriding the iteration behavior of particular visit methods. For instance, downward traversal can be cut off by omitting the call to the accept method of one or more subtrees of a particular node. In neither case can full traversal control be exerted.

The second limitation of tree traversal with visitors is that visitors resist combination. Visitors can only be specialized. In a language that supports multiple implementation inheritance, visitors may even inherit behavior from different parent visitors. But the only flavor of combination obtained with multiple inheritance is static exclusive disjunction (each visit method is either inherited from the one or from the other parent, and this is decided at compile time), and may require an overwhelming ambiguity resolution effort from the programmer. Less restricted combination of visitors would allow better visitor code reuse.

In this chapter, we propose a solution to both limitations. We introduce a small set of *visitor combinators* that can be used to construct new visitors from given ones. As will become clear, by *combinators* we mean reusable classes capturing basic functionality that can be composed in different constellations to obtain new functionality. The basic visitor combinators to be introduced are summarized in Table 5.1. This set of combinators is inspired by the *strategy* primitives of the term rewriting language *Stratego* [VBT99]. The combinators `All` and `One` are traversal combinators that can be used to obtain full traversal control.

In our explanation of the visitor combinators we will use a tiny tree syntax as running example. Figure 5.1 shows its description in BNF. As can be gleaned from this BNF definition, there are two kinds of nodes in our example tree syntax. An internal node, or *fork*, has two subtrees as children, and *leaf* nodes have an Integer value as child. These two kinds of nodes suffice to capture all relevant variability to be found in non-trivial syntaxes, which generally contain large numbers of sorts (non-terminals) and syntax rules.

Combinator	*Description*
Identity	Do nothing (non-iterating default visitor).
Sequence(v1,v2)	Sequentially perform visitor v2 after v1.
Fail	Raise exception.
Choice(v1,v2)	Try visitor v1. If v1 fails, try v2 (left-biased choice).
All(v)	Apply visitor v sequentially to every immediate subtree.
One(v)	Apply v sequentially to the imm. subtrees until it succeeds.

Table 5.1: The set of basic visitor combinators.

```
Node ::= ‘Fork’ ‘(’ Node ‘,’ Node ‘)’
      |  ‘Leaf’ ‘(’ Integer ‘)’
```

Figure 5.1: BNF definition of the running tree example.

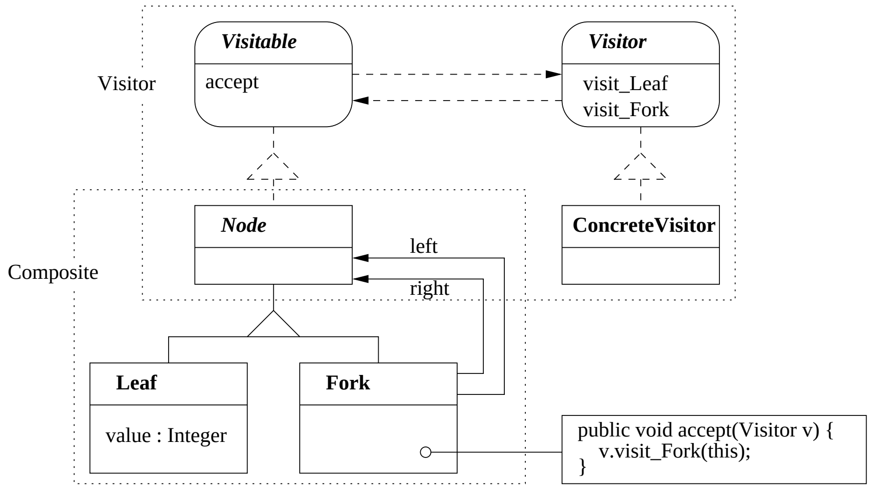


Figure 5.2: Using the Visitor pattern and the Composite pattern for object-oriented tree traversal.

```
interface Visitor {
  public void visit_Leaf(Leaf leaf);
  public void visit_Fork(Fork fork);
}
```

```
class Identity implements Visitor {
  public void visit_Leaf(Leaf leaf) {}
  public void visit_Fork(Fork fork) {}
}
```

Figure 5.3: The `Visitor` interface and the `Identity` combinator.

The use of the Visitor and Composite patterns for object-oriented tree traversal is illustrated for our example syntax in Figure 5.2. The *`Visitor`* interface declares a visit method for each alternative in the grammar. The *`Visitable`* interface declares the `accept` method. This method calls the appropriate visit method from its argument visitor and passes the current top node (`this`) as a parameter. Iteration over a tree can either be implemented in the accept methods, or in default implementations of the visit methods. For the `Fork` node, the Java implementation of the (non-iterating) accept method is shown in a note. Throughout the chapter, we will use Java as implementation language.

The chapter is structured as follows. Sections 5.2 through 5.4 introduce and explain each of the basic visitor combinators of Table 5.1. In Section 5.5 these combinators are refactored into a generic framework, to make them independent of any specific class hierarchy. In Section 5.6 we discuss the support of visitor combinators by our visitor generator JJForester. Finally, Section 5.7 summarizes our contributions, and discusses related work.

5.2 Sequential composition

The first (nullary) combinator in our set of visitor combinators is the traditional non-iterating default visitor, which we call `Identity`. It satisfies the *`Visitor`* interface. The name `Identity` is justified by the behavior of its constituent visit methods: these methods have empty bodies, and therefore preserve the nodes to which they are applied. Also, this default visitor will fulfill the role of an identity element in our set of combinators in an algebraic sense. This will be explained later.

For our example grammar, the *`Visitor`* interface and the `Identity` combinator are shown in Figure 5.3. The *`Visitor`* interface declares a visit method for each kind of node. The visit method for a node takes this node as its argument. The `Identity` visitor implements this interface by providing an empty method body for every visit method. For clarity of presentation, we keep the names of these methods distinct instead of overloading them.

The `Identity` visitor may seem to be useless, because it does nothing, literally. However, in the case of larger tree grammars its usefulness becomes clear. By creating specific visitors as specializations of `Identity`, instead of writing them from scratch, only those methods need to be refined which correspond to nodes at which "special" action needs to be taken. For all other nodes, the default identity behavior is reused. As we will see, its usefulness increases further in the presence of other visitor combinators. Figure 5.4 shows how `Identity` can be refined to a visitor that increments all values in leaf nodes with 1.

Having defined `Identity`, we can proceed to our first "real" combinator: `Sequence`. This is a binary combinator that takes two visitors as arguments, which it sequentially applies to a node. For our example tree syntax, the sequential

```
class AddOne extends Identity {
  public void visit_Leaf(Leaf leaf) {
    leaf.value = leaf.value + 1;
  }
}
```

Figure 5.4: Refinement of the default visitor combinator `Identity`.

```
class Sequence implements Visitor {
  Visitor first;
  Visitor then;
  public Sequence(Visitor first, Visitor then) {
    this.first = first;
    this.then  = then;
  }
  public void visit_Fork(Fork fork) {
    fork.accept(first);
    fork.accept(then);
  }
  public void visit_Leaf(Leaf leaf) {
    leaf.accept(first);
    leaf.accept(then);
  }
}
```

Figure 5.5: The binary visitor combinator `Sequence` applies its argument visitors `first` and `then` one after the other.

visitor combinator is show in Figure 5.5. The `Sequence` combinator is again modeled as a class that implements the *`Visitor`* interface. The arguments of the combinator are modeled as fields `first` and `then` of type *`Visitor`*. The constructor method of `Sequence` initializes these arguments. The visit method for each kind of node is implemented by `Sequence` in the following way. First, the visitor argument stored in field `first` is applied to the node, by calling its `accept` method with this visitor. Then, the same is done with the visitor stored in `then`.

How is the `Sequence` combinator used to create new visitors from given ones? For instance, Figure 5.6 demonstrates how to create a visitor `AddTwo` that applies the `AddOne` visitor twice. First the `Sequence` combinator is refined to the auxiliary combinator `Twice`, which applies its argument visitor twice, sequentially. Then, `Twice` is further refined to use the `AddOne` visitor as argument. In a more concise, mathematical notation these definitions would be written as follows:

$$\begin{array}{lcl} \mathit{Twice(v)} & =_{def} & \mathit{Sequence(v,v)} \\ \mathit{AddTwo} & =_{def} & \mathit{Twice(AddOne)} \end{array}$$

The benefit of creating additional combinators (classes), instead of inlining their definitions, is that this makes them reusable.

```
class Twice extends Sequence {
  public Twice(Visitor v) {
    super(v,v);
  }
}
```

```
class AddTwo extends Twice {
  public AddTwo() {
    super(new AddOne());
  }
}
```

Figure 5.6: Definition of `AddTwo` in terms of `Sequence`, using an auxiliary combinator `Twice`.

```
public class VisitFailure extends Exception { }
```

Figure 5.7: The notion of success and failure of visitors is modeled using a refinement `VisitFailure` of the `Exception` class.

In an algebraic sense, the combinator `Identity` is an identity element for the combinator `Sequence`. This means that the following equalities hold:

$$\begin{aligned} \textit{Sequence(Identity,v)} &= v \\ \textit{Sequence(v,Identity)} &= v \end{aligned}$$

These equations do not necessarily hold for subclasses of `Sequence` and `Identity`, as they might introduce side-effects. Note also that `Twice` applies the *same* argument visitor twice, which means that the same state can be accessed at both applications.

5.3 Alternative composition

We will now introduce a binary visitor combinator `Choice`, which alternatively applies its first or its second argument visitor. More precisely, it will first try one argument visitor, and when this visitor fails, it will try the other. This notion of left-biased alternative composition presupposes a notion of success and failure of visitors.

Success and failure of visitors can be modeled with exceptions. At failure, a `VisitFailure` exception is thrown. Figure 5.7 shows the corresponding refinement of the `Exception` class. The `try` and `catch` constructs are used at choice points.

Given our modeling of visitor failure with exceptions, a nullary visitor combinator `Fail` can be defined, which raises a `VisitFailure` exception for every node. For our example tree syntax, this combinator is defined in Figure 5.8. The clause `throws VisitFailure` also needs to be added to the headers of the previously defined visit and accept methods.

```
class Fail implements Visitor {
  public void visit_Leaf(Leaf leaf) throws VF {
    throw new VisitFailure();
  }
  public void visit_Fork(Fork fork) throws VF {
    throw new VisitFailure();
  }
}
```

Figure 5.8: The `Failure` combinator. The `throws VisitFailure` clause is abbreviated to `throws VF` for reasons of space. We will do so in all figures throughout the chapter.

```
class Choice implements Visitor {
  Visitor first;
  Visitor then;
  public Choice(Visitor first, Visitor then) {
    this.first = first;
    this.then  = then;
  }
  public void visit_Leaf(Leaf leaf) throws VF {
    try { leaf.accept(first); }
    catch (VisitFailure f) { leaf.accept(then); }
  }
  public void visit_Fork(Fork fork) throws VF {
    try { fork.accept(first); }
    catch (VisitFailure f) { fork.accept(then); }
  }
}
```

Figure 5.9: The `Choice` combinator.

Finally, we can proceed to the `Choice` combinator we set out to define. Figure 5.9 shows its definition for our example tree syntax. Like `Sequence`, the `Choice` combinator has two visitor arguments, which are modeled by fields `first` and `then`. It implements the visit method for each kind of node by a try statement that attempts to apply the `first` visitor. If this visitor raises a `VisitFailure` exception, the subsequent catch statement defaults to the `then` visitor.

`Fail` is a zero element for `Sequence`, and an identity for `Choice`. Because of its left-bias, `Choice` has `Identity` as left-zero, but not as right-zero.

$$\begin{array}{lcll}
\textit{Sequence(Fail,v)} & = & \textit{Fail} & \\
\textit{Sequence(v,Fail)} & = & \textit{Fail} & \text{(if } v \text{ side-effect free)} \\
\textit{Choice(Fail,v)} & = & v & \\
\textit{Choice(v,Fail)} & = & v & \\
\textit{Choice(Identity,v)} & = & \textit{Identity} &
\end{array}$$

```
class IsZero extends Fail {
  public void visit_Leaf(Leaf leaf) throws VF {
    if (leaf.value != 0) {
      throw new VisitFailure();
    }
  }
}
```

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

```
class IfZeroAddOne extends Try {
  public IfZeroAddOne() {
    super(new Sequence(new IsZero(),new AddOne()));
  }
}
```

Figure 5.10: Conditional application of visitors, using `Choice` and `Fail`.

The `Fail` and `Choice` combinators can be used to create visitors that *conditionally* fire at certain nodes. For instance, Figure 5.10 demonstrates how to construct a visitor `IfZeroAddOne` that applies the `AddOne` visitor only to leaf nodes that contain the value *0*. A visitor `IsZero` that tests the value of a leaf node is defined by extending `Fail`. The auxiliary unary visitor combinator `Try` is defined as the alternative composition of its visitor argument and the `Identity` visitor. The sequential composition of `IsZero` and `AddOne` is supplied as visitor argument to `Try`. In mathematical notation, these definitions would be written down as follows:

$$\begin{array}{lll} \textit{Try(v)} & =_{def} & \textit{Choice(v,Identity)} \\ \textit{IfZeroAddOne} & =_{def} & \textit{Try(Sequence(IsZero,AddOne))} \end{array}$$

5.4 Traversal combinators

The visitor combinators introduced above can be used to construct new visitors from given ones. A visitor thus created can be applied to a tree by passing it to an `accept` method. Depending on whether this method performs iteration, the visitor is applied to the top node of the tree, or according to a fixed traversal strategy to all nodes in the tree. To obtain more control over traversal behavior, we additionally introduce two *traversal combinators*.

Our first traversal combinator, called `All`, takes one visitor as argument, and applies it to every immediate subtree of the current top node. For our example tree syntax, Figure 5.11 provides the definition of `All`. Since leaf nodes have no subtrees, their visit method does nothing. Fork nodes have two subtrees, to which

```
class All implements Visitor {
  Visitor v;
  public All(Visitor v) {
    this.v = v;
  }
  public void visit_Leaf(Leaf leaf) throws VF { }
  public void visit_Fork(Fork fork) throws VF {
    fork.left.accept(v);
    fork.right.accept(v);
  }
}
```

Figure 5.11: The traversal combinator `All` applies its argument visitor to each immediate subtree.

```
class TopDown extends Sequence {
  public TopDown(Visitor v) {
    super(v,null);
    then = new All(this);
  }
}
```

```
class BottomUp extends Sequence {
  public BottomUp(Visitor v) {
    super(null,v);
    first = new All(this);
  }
}
```

Figure 5.12: Reconstruction of the top-down and bottom-up traversal strategies in terms of `Sequence` and `All`.

the argument visitor is applied one after the other.

The `All` combinator suffices to reconstruct the top-down (pre-order) and bottom-up (post-order) traversal strategies. In mathematical terms, their definitions are as follows:

$$\begin{array}{lll} \mathit{TopDown(v)} & =_{def} & \mathit{Sequence(v,All(TopDown(v)))} \\ \mathit{BottomUp(v)} & =_{def} & \mathit{Sequence(All(BottomUp(v)),v)} \end{array}$$

Note that these definitions are recursive: the combinator being defined occurs in its own definition. For our example tree syntax, the definitions of `TopDown` and `BottomUp` are given in Figure 5.12. The `TopDown` and `BottomUp` visitor combinators are both defined as specializations of the `Sequence` combinator. To model the recursive call of the combinator in Java is somewhat tricky since it is not allowed to reference `this` before the superclass constructor has been called. This is solved by first setting the corresponding visitor argument to `null`. Subsequently this argument is set to its proper value `All(this)`. The combinators `TopDown` and `BottomUp` demonstrate that our set of basic visitor combinators obviates the need for implementing the pre- or post-order traversal strategies in

```
class One implements Visitor {
  Visitor v;
  public void One(Visitor v) {
    this.v = v;
  }
  public void visit_Leaf(Leaf leaf) throws VF {
    throw new VisitFailure(); // Leaf has no kids.
  }
  public void visit_Fork(Fork fork) throws VF {
    try { fork.left.accept(v); }
    catch (VisitFailure f) {
      fork.right.accept(v);
    }
  }
}
```

Figure 5.13: The traversal combinator `One` applies its argument visitor to one of its immediate subtrees.

accept methods or in a default visitor implementation. In fact, visitor combinators make any traversal strategy programmable.

A visitor combinator similar to `All` is the `One` traversal combinator. Whereas `All` applies its argument visitor to all its subtrees, `One` applies it to exactly one. More precisely, it tries to apply it to each subtree in turn, until application succeeds. Figure 5.13 gives the definition of `One` for our example tree grammar. As leaf nodes have no subtrees, the corresponding visit method of `One` immediately fails. In the case of fork nodes, a visit to the left subtree is attempted first. If it fails, a visit to the next subtree is attempted.

As an indication of the level of traversal control that can be obtained with the combinators `All` and `One`, Figure 5.14 lists a number of different traversal strategy combinators. These combinators are object-oriented reconstructions of a few out of many strategy combinators that can be found in the standard library of the strategic term rewriting language Stratego [Vis01a].

5.5 Syntax-independence

All combinators presented above were defined relative to our example tree syntax of Figure 5.1. So the question arises to what extent they are specific to this particular syntax, and to what extent they are generic[1]and reusable for any syntax. In this section we will explain that our combinators are generic in nature, and we will

[1]We use the term ‘generic’ in the general sense that a generic program is not restricted to one particular type, but can work on entities of many different types. In a more restricted sense, ‘generic’ has been used for programs that *uniformly* work for entities of *every* type (parametric polymorphism), for instance in the context of Ada, Eiffel, and GenericJava. In the context of PolyP [JJ97b], ‘generic’ has been used for programs that perform induction on a type parameter (polytypism).

```
class OnceBottomUp extends Choice {
  public OnceBottomUp(Visitor v) {
    super(null,v);
    first = new One(this);
  }
}
```

```
class SpineBottomUp extends Sequence {
  public SpineBottomUp(Visitor v) {
    super(null,v);
    first = new Choice(new One(this),
                       new All(new Fail()));
  }
}
```

```
class DownUp extends Sequence {
  public DownUp(Visitor down, stop, up) {
    super(null, up);
    first = new Sequence(
                  down,
                  new Choice(stop,new All(this)));
  }
}
```

Figure 5.14: With the combinators `All` and `One`, arbitrary traversal strategies can be defined. `OnceBottomUp` applies `v` exactly once at the first location found during bottom-up traversal. `SpineBottomUp` applies `v` bottom-up along a path which reaches from the root to one of the leaves. `DownUp` applies `down` going down the tree, and applies `up` when coming back up. It cuts off the traversal below nodes where `stop` succeeds.

show how we can modify their encoding to isolate them from the specifics of any particular syntax.

5.5.1 Lack of genericity

The combinators presented above lack genericity in two respects.

Firstly, when defining a visitor, whether from scratch or by specialization of a basic visitor combinator, all its visit methods need to be (re)defined separately, even if the *same* behavior is required for each of them. At the risk of sounding paradoxical, one might say that a more generic way of specialization is needed. For instance, specialization should be possible of all visit methods at once, or, in case of a syntax with multiple sorts, of all visit methods for a particular sort at once.

Genericity is lacking in a second respect. In natural language we would have no trouble defining the behavior of all our basic combinators and most defined ones without reference to the example syntax. In fact, we gave just such a generic explanation in the running text of this chapter. Still, they can not be reused 'as is' for other syntaxes, because they refer (directly or indirectly) to the specific terminals of our example syntax.

5.5.2 Visitor combinators in frameworks

To remove these limitations on the genericity of our visitor combinators, we need to refactor the design pattern of Figure 5.2 such that syntax-specific functionality is separated from generic functionality. Our solution is a variation on the *staggered* Visitor pattern [Vli99], which introduces generic counterparts *`AnyVisitor`* and *`AnyVistable`* for the syntax-specific interfaces. This is illustrated in Figure 5.15. Note that *`AnyVisitable`* and *`Visitable`* are now abstract classes instead of interfaces because *`AnyVisitable`* implements the `accept_Any` method. Likewise, *`Visitable`* is now an abstract class, because it implements the `visit_Any` method. For our example tree syntax, this implementation is shown in Figure 5.16. It uses runtime type identification (RTTI) to cast its generic *`AnyVisitable`* argument to a syntax-specific *`Visitable`*. If the cast succeeds, it applies itself, using the syntax-specific `accept` method. Thus, syntax-specific visitors have forwarding (delegation) built-in from generic to syntax-specific visit methods.

In the original staggered visitor pattern, the *`Visitor`* class also has the converse forwarding built-in, from syntax-specific visit methods to generic visit methods. In Figure 5.15, this converse forwarding has been factored out into a new combinator `Fwd`. Its definition for our example syntax is shown in Figure 5.17. `Fwd` is a unary combinator that takes a generic visitor as argument, and implements all syntax-specific visit methods by forwarding to that generic visitor. The

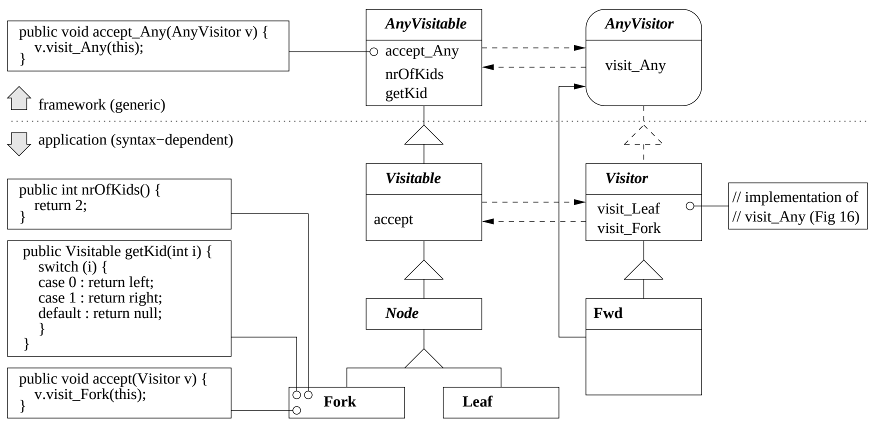


Figure 5.15: Separating out generics from specifics.

```
public void visit_Any(AnyVisitable x) throws VF {
  if (x instanceof Visitable) {
      ((Visitable) x).accept(this);
  } else {
      throw new VisitFailure();
  }
}
```

Figure 5.16: The default implementation of `visit_Any` of the syntax-specific abstract *`Visitor`* class.

```
public class Fwd implements Visitor {
  AnyVisitor v;
  public Fwd(AnyVisitor v) {
    this.v = v;
  }
  public void visit_Leaf(Leaf leaf) throws VF {
    v.visit_Any(leaf);
  }
  public void visit_Fork(Fork fork) throws VF {
    v.visit_Any(fork);
  }
}
```

Figure 5.17: The visitor combinator `Fwd` creates a syntax-specific visitor from a generic visitor.

```
Identity: ; // Skip
Sequence: x.accept_Any(this.first);
          x.accept_Any(this.then);
    Fail: throw new VisitFailure();
  Choice: try { x.accept_Any(first); }
          catch (VisitFailure f) {
            x.accept_Any(then);
          }
     All: for (int i = 0; i < x.nrOfKids(); i++) {
            x.getKid(i).accept_Any(this.v);
          }
     One: for (int i = 0; i < x.nrOfKids(); i++) {
            try {
              x.getKid(i).accept_Any(this.v);
              return;
            }
            catch(VisitFailure f) { ; }
          }
          throw new VisitFailure();
```

Figure 5.18: Generic reformulations of the basic visitor combinators. Only the body of `visit_Any(AnyVisitable x)` for the various combinators is shown.

benefit of this new combinator over built-in converse forwarding is that the forwarding behavior is reusable for several generic visitors. This is essential in the setting of visitor *combinators*, because these are typically used to continuously construct new visitors.

Apart from the generic visit method `visit_Any`, the abstract *`AnyVisitor`* class declares two additional methods. `nrOfKids` returns the number of children of a visitable node, and `getKid(i)` returns its i^{th} child. Below, we will use these methods for the generic definition of the traversal combinators `All`, and `One`. For *fork* nodes, the definitions of these methods are shown in Figure 5.15.

5.5.3 Generic combinators

With this refactoring in place, 'generic specialization' of visitors is possible. In particular, the basic visitor combinators can now be given syntax-independent definitions. The required implementations of the corresponding `visit_Any` methods are shown in Figure 5.18. When these generic combinators are passed to `Fwd`, the original, syntax-specific combinators are obtained. The penalty for the additional genericity is the method call from `Fwd` to the generic visitor, and, in the case of combinators with arguments, the explicit cast from *`AnyVisitable`* to *`Visitable`* in the `visit_Any` method.

Note that the generic formulation of the traversal combinators `All` and `One` make use of the methods `nrOfKids` and `getKid` of the generic *`AnyVisitor`* interface. Thus, the syntax-specific knowledge about children is hidden behind these two methods.

To assess the performance penalty of the additional genericity, we compared three implementations of a topdown traversal.

Iterating visitor Node actions and traversal code are entangled in a single syntax-specific visitor, which is passed to a syntax-specific accept method.

Syntax-specific combinator The syntax-specific `TopDown` combinator (see Figure 5.12) takes a syntax specific visitor with node actions as argument, and is passed to a syntax-specific accept method.

Generic combinator The generic `TopDown` combinator takes a syntax-specific visitor with node actions as argument, and is passed to a generic accept method.

Benchmarks on balanced trees of various sizes indicated that the syntax-specific combinator is a constant factor 3 slower than the iterating visitor. The generic combinator is yet another constant factor 2 slower.

5.5.4 Towards libraries of generic algorithms

Given these generic definitions of our visitor combinators where generics are cleanly separated from syntax-specifics, all (traversal) combinators that are generic 'in nature' can indeed be implemented as classes that are reusable across syntaxes. This opens the door to the construction of libraries of reusable generic visitor combinators, such as those in Figures 5.12 and 5.14. Programming with visitor combinators then becomes a matter of specializing (in generic or syntax-specific manner) and composing predefined combinators, and feeding them to the accept methods of the particular tree structures that need to be visited. The required syntax-specific code is limited to the specific *`Visitor`* and *`Visitable`* interfaces, the accept method implementations, and the `Fwd` combinator. Such syntax-specific code can be generated from a grammar (see Section 5.6.1 and Chapter 6).

To demonstrate the development of generic combinators and their instantiation for specific syntaxes we discuss a small example. First, we will define a generic combinator for simple def-use analysis. This combinator abstracts from which syntactic constructs count as definitions, and which as uses. Secondly, we will instantiate the generic combinator for the syntax of a specific language: GraphXML [HM00]. GraphXML is an XML-based graph description and exchange language. It allows graphs to be described in terms of nodes and edges, where each edge is defined by a source and a target attribute. We will use the generic def-use analysis to determine the roots and sinks of a graph.

```
public interface Collector extends AnyVisitor {
  public Set getSet();
}
```

```
public class DefUse extends TopDown {
  Collector def;
  Collector use;
  public DefUse(Collector def, Collector use) {
    super(new Sequence(def,use));
    this.def = def;
    this.use = use;
  }
  public Set getUndefined() {
    HashSet result = new HashSet(use.getSet());
    return result.removeAll(def.getSet());
  }
  public Set getUnused() {
    HashSet result = new HashSet(def.getSet());
    return result.removeAll(use.getSet());
  }
}
```

Figure 5.19: A language-independent combinator for def-use analysis.

Generic def-use analysis The simple algorithm we wish to implement collects *use* and *definition* occurrences from an input tree while performing a single top-down traversal. After the traversal, two sets should be obtained: the set of entities that are used but not defined, and the set of entities that are defined but not used. Our implementation is shown in Figure 5.19. It consists of the interface *`Collector`*, which extends the generic visitor interface *`AnyVistor`* with a `getSet` method, and the class `DefUse` which implements the actual algorithm. Informally, `DefUse` is the following combinator:

$$\mathit{DefUse(def,use)} \quad =_{\mathit{def}} \quad \mathit{TopDown(Sequence(def,use))}$$

Here, *def* and *use* are visitor parameters that have collecting defined and used entities as side effect. Because *TopDown* is the outermost symbol in the definition of *DefUse*, the latter is implemented by extending the former. The fields `def` and `use` store the references to the visitor parameters, for reference by the methods `getUndefined` and `getUnused`. These use simple set operations to compute the result sets we wanted to obtain.

Instantiation To instantiate the generic algorithm for GraphXML, we need to provide visitors that collect defined and used GraphXML entities. In the domain of graphs, nodes can be considered ‘definitions’ when they occur as the source of a directed edge. Likewise, they can be considered ‘uses’ when they occur as target. A node that occurs as source but not as target is a *root* of the graph. Conversely. a

```
EdgeAttribute := 'source' '=' AttValue
               | 'target' '=' AttValue
```

```
public class CollectTarget extends Fwd
                           implements Collector {
  Set targets = new Set();
  public CollectTarget() {
    super(new Identity());
  }
  public void visit_Target(Target attValue) {
    targets.add(attValue);
  }
  public Set getSet() {
    return targets;
  }
}
```

```
public void testDefUseVisitor(GraphXML g) throws VF {
  DefUse defuse = new DefUse(new CollectSource(),
                             new CollectTarget());
  g.accept_Any(defuse);
  System.out.println("Sinks: "+defuse.getUndefined());
  System.out.println("Roots: "+defuse.getUnused());
}
```

Figure 5.20: Retrieving the roots and sinks of a GraphXML document by GraphXML-specific instantiation of the generic def-use combinator. The class `CollectSource` is not shown. It is similar to `CollectTarget`.

node that occurs as target, but not as source is a *sink*. Determining roots and sinks corresponds to determining unused definitions and undefined uses.

Figure 5.20 shows the relevant fragment of the GraphXML syntax, and the code that instantiates the generic def-use analysis for GraphXML. The classes `CollectTarget` and `CollectSource` are the required implementations of the interface *`Collector`*. Their definitions are similar. The combinator `CollectTarget`, for example, is a specialization of the GraphXML-specific identity combinator, which is defined as *Fwd(Identity)*. It redefines a single visit method: the one corresponding to the syntax rule for targets. The redefined visit method simply adds the target it encountered to the local `Set` field. Given these collector classes, the generic def-use analysis can be instantiated for GraphXML as follows:

DefUse(CollectSource,CollectTarget)

This is conveyed by the test method in Figure 5.20.

The def-use analysis is only one example of a generic algorithm that can be programmed with visitor combinators. In Stratego, generic algorithms have been defined e.g. for graph transformation and analysis, and for substitution, renaming and unification [Vis01a, Vis00b]. In [Läm02a], a generic refactoring algorithm is developed, which is a generalization of the Java extract method refactoring we presented in Section 4.

5.6 Support

In the previous sections, we have seen small, tutorial examples of programming with (generic) visitor combinators. In Chapters 6 and 7, we will provide a detailed account of our experiences with applying them on a larger scale. Our primary objective in developing visitor combinators has been to support the employment of object-oriented programming technology in the domain of language processing. Chapter 6 discusses static analysis of Toolbus scripts to generate communication graphs. Chapter 7 discusses the application to Cobol control flow analysis, in the context of legacy system redocumentation [DK99a]. This application involves traversal over both tree-shaped and graph-shaped object structures.

In this section, we provide a brief preview over the tool support that we developed to make such applications possible. Chapter 6 presents this support in detail.

5.6.1 JJTraveler

The generic visitor combinator framework presented in this chapter, as well as the various generic basic and defined combinators, are reusable for any Java class hierarchy. We have collected the combinators in a visitor combinator library. We have bundled both framework and library into a single distribution, called JJTraveler. We have used JJTraveler in several applications, and in the course of these applications, new generic combinators have been developed and added to JJTraveler. An excerpt of JJTraveler's library is shown in Table 5.2. Each of the combinators in the table is briefly explained by a single sentence or a concise mathematical definition. A full overview of the library can be found in the online documentation of JJTraveler.

5.6.2 JJForester

As mentioned above, the only syntax-specific code that is needed for programming with visitor combinators, are the interfaces *`Visitor`* and *`Visitable`*, the accept methods, and the combinator `Fwd`. Rather than writing this code manually, it can be generated from tree or syntax definitions. As will be discussed in Chapter 6, we have extended our parser and visitor generator JJForester with this functionality. The input to JJForester consists of a grammar specified in the syntax definition formalism SDF [HHKR89]. SDF is supported by a parse table generator, and a *generalized* LR parser generator. These tools are available as stand-alone components, which are reused by JJForester to generate a parse table at compilation

Combinator	Description of behavior
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)	$Choice(All(AllBottomUp(v)), v)$
TopDownWhile(v)	$Try(Sequence(v, All(TopDownWhile(v))))$
TopDownUntil(v)	$Choice(v, All(TopDownUntil(v)))$
BreadthFirst(v)	Breadth-first traversal strategy
BreadthFirstWhile(v)	Breadth-first traversal strategy with cut-off
SpineTopDown(v)	Top-down traversal along a spine
SpineBottomUp(v)	Bottom-up traversal along a spine
GuaranteeSuccess(v)	Catch `VisitFailure` and re-throw it as runtime exception
LogVisitor(v)	Create log of each invocation of v
Visited	Succeed if not accepted by current visitable before
IsDag	Test if graph rooted by current visitable is directed and acyclic
IsTree	Test if graph rooted by current visitable is tree-shaped

Table 5.2: JJTraveler's library (excerpt).

time, and to parse input terms at run time. The SDF grammar is also passed to the code-generation component of JJForester, which emits the Java classes that instantiate the visitor combinator framework JJTraveler. The user programs against the generated code and the framework. The framework, the generated code and the user code can be compiled to byte code, and run on a virtual machine. During run time, calls to parse methods will lead to invocations of the parser, which returns abstract syntax trees (ASTs). These ASTs are passed to factory methods to build the actual object structure.

In Section 5.5, we added the possibility of generic specialization to the possibility of syntax-specific, per production specialization. For a many-sorted syntax, an intermediate level of genericity is conceivable: per sort specialization. This can be realized by inserting another set of interfaces, and another forwarding combinator, like `Fwd`. JJForester supports a simplified variation on this scheme, where both layers of forwarding are done in a single `Fwd` combinator. This combinator offers one visit method per sort as well as one per production. The per-production methods forward to the per-sort methods, and these in turn forward to the generic inherited visit method.

5.6.3 Visitor combinators for ATerms

The ATerm library [BJKO00] supports (space) efficient representation and exchange of generic trees through maximal subtree sharing. The Java implementation of the ATerm library has been extended to enable the use of JJTraveler's visitor combinators to process ATerms. ATerms are an instantiation of the Flyweight pattern [GHJV94], where the children of a term belong to its *internal* state. Hence, ATerms are immutable objects, and visiting them is done most appropriately with visitors that have return values. JJTraveler caters for this need.

5.7 Concluding remarks

We reconstructed the basic strategy combinators of Stratego [VBT99] in the object-oriented setting as a suite of basic visitor combinators (reusable classes). These visitor combinators remove the limitations of the classic visitor pattern with respect to composability of visitors and with respect to traversal control. Additionally, we refactored the visitor combinator design pattern into a syntax-independent framework, and a design pattern for instantiating the framework for a specific syntax. This visitor combinator framework opens the door to a new style of generic object-oriented programming, where new frameworks are built by composition of basic visitor combinators of the basic framework.

5.7.1 Evaluation

How does programming visitor combinators compare with programming with ordinary visitors, or without visitors at all?

Explicit stack maintenance Visitors *iterate* over an object structure. Consequently, they can not use the call stack to pass data. Instead, the state of the visitor is used for this. When, at back-tracking, the state needs to be restored, this is not done automatically as it would when the call stack would be used. Instead, state restoration at back-tracking needs to be done explicitly by the visitor. When using visitor *combinators*, the stack maintenance can be done by separate, reusable combinators.

Another technique to pass data is to stop iteration, and restart with a new *instance* of the visitor, which has its own state to hold data. When the new instance finishes, the iteration can be resumed with the old state. With this technique, no explicit stack maintenance is needed. At each recursive calls to the visitor constructor method, the call stack is (implicitly) used to implement the desired back-tracking behavior. This technique will be demonstrated in Chapter 7.

Performance Using visitor combinators introduces the overhead of forwarding of method calls between combinators. Of course, the precise amount of forwarding overhead is strongly dependent on the particular constellation of combinators, and on the ratio of combinator code. On the other hand, the additional traversal control can be used to construct more efficient traversal strategies. In our benchmarks experiments and in the Cobol control-flow application (see Chapter 7), we did not experience performance problems. For a complete picture of performance consequences, more experience and experiments are needed.

Paradigm shift The style of programming with visitors is one step removed from the 'natural' method passing style of object-oriented programming, where data and operations on the data are encapsulated in the same object. The style of programming with visitor combinators is yet one more step removed from this 'natural' style. In fact, programming with visitor combinators can be considered a *paradigm shift*, as it not only separates data from operations, but also introduces the technique of developing programs, not by adding classes and methods, but by composing compound classes from basic ones.

Robustness A well-know problem of the visitor design pattern is that visitors are *brittle* with respect to changes in the class hierarchy on which they operate. When, for instance, a class is added to the hierarchy, all previously defined visitors need to implement an additional visit method. To some extent, default `Visitor` implementations provide isolation against such changes [GHJV94].

Our visitor combinators actually are such default implementations. By inheriting from them, user-defined combinators only need to depend on those fragments of the class hierarchy (or syntax) that are relevant to the functionality they implement. By making the combinators syntax-independent, we have even made these default implementations robust against changes in the class hierarchy. Finally, the generator concentrates all syntax-dependence in the syntax itself. For instance, one simply adds a new production to the syntax, and the generator takes care of updating the class hierarchy, the *`Visitor`* class, and the `Fwd` combinator.

5.7.2 Generic traversal across paradigms

The generic traversal combinators presented in this chapter are the result of transposing concepts that have (recently) been developed in other paradigms. The most important source of inspiration is the strategic term rewriting language Stratego [VBT99]. Our visitor combinators are reconstructions of the primitive strategy combinators of Stratego. Similar primitives can also be found in the rewriting calculus and its extensions [CK99]. However, both these languages are untyped (though a proposal for an appropriate type system has recently been drafted [Läm02b]). Previously, we reconstructed Stratego's primitives in the strongly typed functional language Haskell (see [LV00] and Chapter 4), and these typed strategies guided the design of our visitor combinators. Thus, incarnations of the concept of strategy combinators, including combinators for generic traversal, are now available in three different programming paradigms.

Strategy combinators are closely related to folds. Many-sorted folds can be constructed by specialization and combination of fold combinators just like strategies [LVK00]. The main difference between folds and strategies is that the former employ a fixed bottom-up traversal strategy. The relation between strategies and folds is discussed in more detail in [LV00]. The *Translator* pattern [Küh98] shows how many-sorted folds can be implemented in an object-oriented setting.

5.7.3 Related work

Traversal control The *hierarchical* visitor pattern [c2] employs a visitor interface with two methods per visitable class: one to be performed upon entering the class, and one to be performed before leaving it. This pattern allows hierarchical navigation (keeping track of depth) and conditional navigation (cutting off traversal below a certain point). As Figure 5.14 demonstrates, visitor combinators can be used to achieve such traversal control, and much more.

In adaptive programming, and its implementation by the Demeter system [LPS97], a notion is present of traversal *strategies* for object structures. These strategies should not be confused with the strategies and strategy combinators of the Stratego language which inspired our visitor combinators. Demeter's strategies are high-level descriptions of paths through object graphs in terms of source node, target node, intermediate nodes, and predicates on nodes and edges. These high level descriptions are translated (at compile time) into ‘dynamic roadmaps’: methods that upon invocation traverse the object structure along a path that satisfies the description. During traversal, a visitor can be applied. The aim of these strategies is to make classes less dependent on the particular class structure in which they are embedded, i.e. to make them more robust, or adaptive. Unlike our visitor combinators, Demeter’s strategies are *declarative* in nature and can not be executed themselves. Instead, traversal code must be generated from them by a constraint-solving compiler. On the other hand, while reducing commitment to the class structure, Demeter’s strategies do not eliminate all references to the class structure. Visitor combinators allow definition of *fully* generic traversals.

To complement Demeter’s declarative strategies, a domain-specific language (DSL) has been proposed to express recursive traversals at a lower, more explicit level [OW99]. This traversal DSL sacrifices some compactness and adaptiveness in order to gain more control over propagation and computation of results, and to prevent unexpected traversal paths due to underspecification of traversals. With respect to our visitor combinators, this traversal DSL provides cleaner support for *recursive* traversals. On the other hand, visitor combinators are more generic, extensible and reusable, and they offer more traversal control. Also, they do not essentially rely on tool support.

Generics The separation of specifics and generics in the visitor pattern is addressed by Vlissides’ *staggered* visitor pattern [Vli99], and the *extended visitor* pattern supported by the SableCC tool [GH98]. Here the aim of this separation is to allow extension of the syntax without altering existing (visitor) code. In the extended visitor pattern of SableCC, the generic visitor interface does not contain any methods. In the staggered pattern, the generic visitor contains a generic visit method, similar to our `visit_Any`. The main difference with our approach is that in these patterns forwarding from specific to generic visit methods is done in the *`Visitor`* class, while we do it in a separate reusable combinator `Fwd`. In the presence of `Choice`, the `Fwd` combinator allows not only extension of a syntax, but also merging of several syntaxes.

The `Walkabout` class [PJ98] makes essential use of reflection (including, but not limited to RTTI) to model generic visiting behavior. The class performs a traversal through an object structure. At each node it reflects on itself to ascertain whether it contains a visit method for the current node. If not, it uses reflection to determine the fields of the current node and calls itself on these. The authors report high performance penalties for the extensive reliance on reflection. The benefit is that no (syntax-specific) accept methods, visitor interface, or visitor combinators need to be supplied. The `Walkabout` class implements a fixed top-down traversal strategy, which is cut off below nodes for which the visitor fires (i.e. *DownUp(Identity,v,Identity)*, see Figure 5.14).

5.7.4 Future work

Implementation The support currently offered by JJForester has a restraining effect. Classes are generated in their entirety from a grammar. Programmers can not add their own methods or fields. This may be desirable, for instance, to allow decoration of the tree that is being traversed. To offer more flexibility, JJForester should additionally generate such decoration fields, or *weave* the generated methods into given classes (a form of aspect-oriented programming [KL$^+$97]).

For several visitor combinators we gave definitions in mathematical notation, as well as in Java. Obviously, a language extension could be defined which allows combinator definitions in this concise, mathematical style. An implementation of such an extension would map these high-level combinator definitions to their more verbose counterparts in the base language. Such a language extension would be amenable to optimization by means of source-to-source transformation, on the basis of the algebraic equations that hold between visitor combinators.

Extensions and alternatives It would be convenient to include some special-purpose visitors into the combinator set, e.g. `ToString`, `Equals`, and `Clone` visitors. These would help to address common traversal scenarios such as pretty-printing, and non-destructive transformation. Currently, only a `ToString` combinator is generated by JJForester.

Document processing, like language processing, essentially involves tree traversal. The GraphXML example of Section 5.5.4 illustrates how (DTD-aware) document processing can be done with visitor combinators. We want to compare our techniques with existing proposals for XML-document traversal (*cf*. DOM [DOM98], XSLT [XSL99]), and investigate whether these could benefit from the combinator approach.

Until now, we assumed tree shaped object graphs. This restriction is not essential. When it is removed, the need arises for some mechanism to mark nodes as visited, and to ensure termination (in case of cycles). Chapter 7 discusses how, with the use of a single additional basic combinator, graph traversals can be implemented with visitor combinators.

Chapter 6

Object-oriented Tree Traversal with JJForester

In this chapter, we complement the generic traversal support for object-oriented programming introduced in the previous chapter with the advanced language processing technology available in the ASF+SDF Meta-Environment. In particular, we combine the syntax definition formalism SDF and the associated components that support generalized LR parsing with the general purpose programming language Java.

To this end, we implemented JJForester: a parser and visitor generator for Java that takes SDF grammars definition as input. It generates class structures that implement a number of *design patterns* to facilitate construction and traversal of parse trees represented by object structures. JJForester supports both simple traversals following the plain visitor pattern and advanced traversals using our visitor combinator framework JJTraveler. In small examples and a detailed case study, we demonstrate how program analyses and transformations can be constructed with JJForester.

This chapter is based on [KV01].

6.1 Introduction

JJForester is a parser and visitor generator for Java that takes language definitions in the syntax definition formalism SDF [HHKR89, Vis97] as input. It generates Java code that facilitates the construction, representation, and manipulation of syntax trees in an object-oriented style. To support *generalized LR parsing* [Tom85, Rek92], JJForester reuses the parsing components of the ASF+SDF Meta-Environment [Kli93]. To enable visitor code reuse and to address advanced tree traversal scenarios, JJForester instantiates the *visitor combinator framework* JJTraveler (see Chapter 5).

The ASF+SDF Meta-Environment is an interactive environment for the development of language definitions and tools. It combines SDF (Syntax Definition Formalism) with the term rewriting language ASF (Algebraic Specification Formalism [BHK89]). SDF is supported with generalized LR parsing technology. For language-centered software engineering applications, generalized parsing offers many benefits over conventional parsing technology [BSV98]. ASF is a rather pure executable specification language that allows rewrite rules to be written in concrete syntax.

In spite of its many qualities, a number of drawbacks of the ASF+SDF Meta-Environment have been identified over the years. One of these is its unconditional bias towards ASF as programming language. Though ASF was well suited for the *prototyping* of language processing systems, it lacked some features to build mature *implementations*. For instance, ASF does not come with a strong library mechanism, I/O capabilities, or support for generic term traversal[1]. Also, the closed nature of the meta-environment obstructed interoperation with external tools. As a result, for a mature implementation one was forced to abandon the prototype and fall back to conventional parsing technology. An example is the ToolBus [BK98], a software interconnection architecture and accompanying language, that has been simulated extensively using the ASF+SDF Meta-Environment, but has been implemented using traditional Lex and Yacc parser technology and a manually coded C program. For Stratego [VBT99], a system for term rewriting with strategies, a simulator has been defined using the ASF+SDF Meta-Environment, but the parser has been hand coded using ML-Yacc and Bison. A compiler for RISLA, an industrially successful domain-specific language for financial products, has been prototyped in the ASF+SDF Meta-Environment and re-implemented in C [B$^+$96].

To relieve these drawbacks, the Meta-Environment has recently been re-implemented in a component-based fashion [BDH$^+$01]. Its components, including the parsing tools, can now be used separately. This paves the way to adding support for alternative programming languages to the Meta-Environment.

As a major step into this direction, we have designed and implemented JJForester. This tool combines SDF with the mainstream general purpose programming language Java. Apart from the obvious advantages of object-oriented programming (e.g. data hiding, intuitive modularization, coupling of data and accompanying computation), it also provides language tool builders with the massive library of classes and design patterns that are available for Java. Furthermore, it facilitates a myriad of interconnections with other tools, ranging from database servers to remote procedure calls. Apart from Java code for constructing and representing syntax trees, JJForester generates visitor classes that facilitate generic

[1]Recently, some support for generic traversal has been added to the ASF interpreter (see also Section 6.5.2).

traversal of these trees. For advanced traversal scenarios, JJForester enables the use of visitor *combinators*. This combination of features makes JJForester suitable for component-based development of program analyses and transformations for languages of non-trivial size.

The chapter is structured as follows. Section 6.2 explains JJForester. We discuss what code it generates, and how this code can be used to construct various kinds of tree traversals. Section 6.3 explains JJForester's connection to JJTraveler. We briefly review the notion of *visitor combinators* and demonstrate their use in constructing complex tree traversals. Section 6.4 provides a case study that demonstrates in depth how a program analyzer (for the ToolBus language) can be constructed using JJForester.

6.2 JJForester

JJForester is a parser and visitor generator for Java. Its distinction with respect to existing parser and visitor generators, e.g. Java Tree Builder, is twofold. First, it deploys generalized LR parsing, and allows *unrestricted*, *modular*, and *declarative* syntax definition in SDF (see Section 6.2.2). These properties are essential in the context of component-based language tool development where grammars are used as *contracts* (see Chapter 2). Second, to cater for a number of recurring tree traversal scenarios, it generates variants on the Visitor pattern that allow different traversal strategies.

In this section we will give an overview of JJForester. We will briefly review SDF which is used as its input language (a similar review was given in Chapter 2). By means of a running example, we will explain what code is generated by JJForester and how to program against the generated code. In the next section, we will provide a more in-depth discussion of tree traversal using visitor combinators.

6.2.1 Overview

The global architecture of JJForester is shown in Figure 6.1. Tools are shown as ellipses. Shaded boxes are generated code. Arrows in the bottom row depict run time events, the other arrows depict compile time events. JJForester takes a grammar defined in SDF as input, and generates Java code. In parallel, the parse table generator PGEN is called to generate a parse table from the grammar. The generated code is compiled together with code supplied by the user. When the resulting byte code is run on a Java Virtual Machine, invocations of *parse* methods will result in calls to the scannerless, generalized LR parser SGLR. From a given input term, SGLR produces a parse tree as output. These parse trees are passed through the parse tree implosion tool *implode* to obtain abstract syntax trees. Note that the PGEN and SGLR components are reused from the ASF+SDF Meta-Environment.

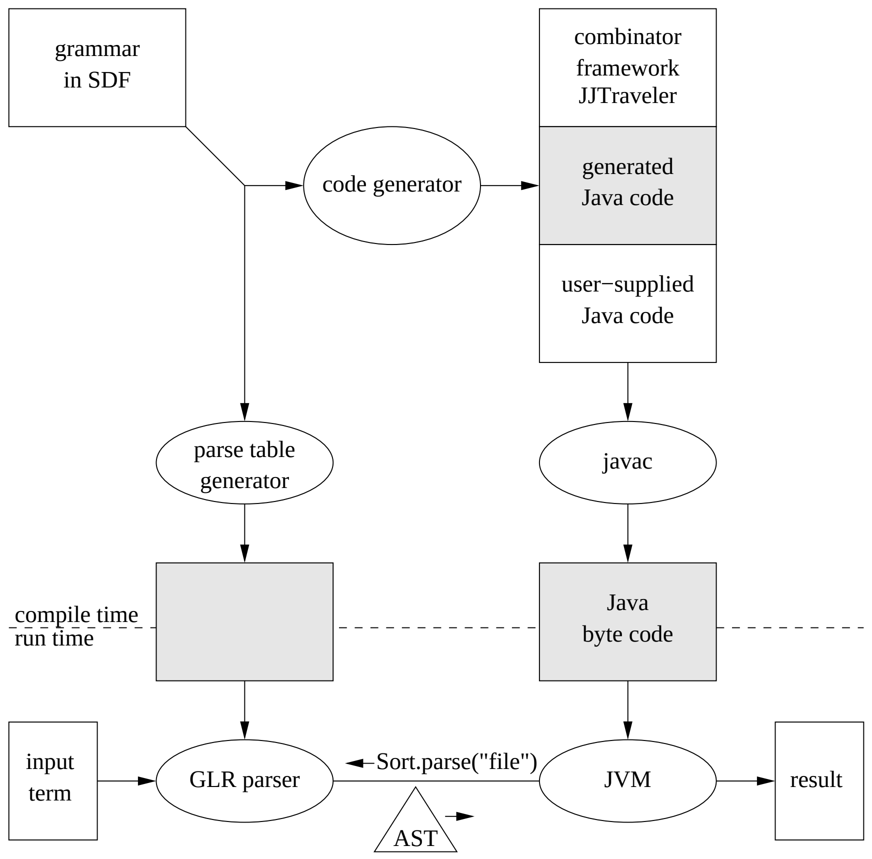


Figure 6.1: Global architecture of JJForester. Ellipses are tools. Shaded boxes are generated code.

6.2.2 SDF

The language definition that JJForester takes as input is written in SDF. In order to explain JJForester, we will give a short introduction to SDF. A complete account of SDF can be found in [HHKR89, Vis97].

SDF stands for Syntax Definition Formalism, and it is just that: a formalism to define syntax. SDF allows the definition of lexical and context-free syntax in the same formalism. SDF is a modular formalism; it allows productions to be distributed at will over modules. For instance, mutually dependent productions can appear in different modules, as can different productions for the same non-terminal. This implies, for instance, that a kernel language and its extensions can be defined in different modules. Like extended BNF, SDF offers constructs to define optional symbols and iteration of symbols, but also for separated iteration of symbols, alternatives, and more.

Figure 6.2 shows an example of an SDF grammar. This example grammar gives a modular definition of a tiny lambda calculus-like language with typed lambda functions. Note that the orientation of SDF productions is reversed with respect to BNF notation. The grammar contains two context-free non-terminals, Expr and Type, and two lexical non-terminals, Identifier and LAYOUT. The latter non-terminal is used *implicitly* between all symbols in context-free productions. As the example details, expressions can be variables, applications, or typed lambda abstractions, while types can be type variables or function types.

SDF's expressiveness allows for defining syntax concisely and naturally. SDF's modularity facilitates reuse. SDF's declarativeness makes it easy and retargetable. But the most important strength of SDF is that it is supported by *Generalized LR Parsing*. Generalized parsing removes the restriction to a non-ambiguous subclass of the context-free grammars, such as the LR(k) class. This allows a maximally natural expression of the intended syntax; no more need for 'bending over backwards' to encode the intended grammar in a restricted subclass. Furthermore, generalized parsing leads to better modularity and allows 'as-is' syntax reuse.

As SDF removes any restriction on the class of context-free grammars, the grammars defined with it potentially contain ambiguities. For most applications, these ambiguities need to be resolved. To this end, SDF offers a number of disambiguation constructs. The example of Figure 6.2 shows four such constructs. The *left* and *right* attributes indicate associativity. The *bracket* attribute indicates that parentheses can be used to disambiguate Exprs and Types. For the lexical non-terminals the longest match rule is explicitly specified by means of *follow restrictions* (indicated by the `-/-` notation). Not shown in the example is SDF's notation for relative priorities.

In the example grammar, each context-free production is attributed with a *constructor name*, using the *cons(..)* attribute. Such a grammar with constructor names amounts to a simultaneous definition of concrete and abstract syntax of the language at hand. The *implode* back-end turns concrete parse trees emanated

```
definition

module Expr
exports
  context-free syntax
    Identifier                          → Expr {cons("Var")}
    Expr Expr                           → Expr {cons("Apply"), left}
    "\\" Identifier ":" Type "." Expr   → Expr {cons("Lambda")}
    "(" Expr ")"                        → Expr {bracket}

module Type
exports
  context-free syntax
    Identifier        → Type {cons("TVar")}
    Type "->" Type    → Type {cons("Arrow"), right}
    "(" Type ")"      → Type {bracket}

module Identifier
exports
  lexical syntax
    [A-Za-z0-9]+ → Identifier

  lexical restrictions
    Identifier-/- [A-Za-z0-9]

module Layout
exports
  lexical syntax
    [\␣\t\n] → LAYOUT

  context-free restrictions
    LAYOUT?-/- [\␣\t\n]
```

Figure 6.2: Example SDF grammar.

by the parser into more concise abstract syntax trees (ASTs) for further processing. The constructor names defined in the grammar are used to build nodes in the AST[2]. As will become apparent below, JJForester operates on these abstract syntax trees, and thus requires grammars with constructor names. A utility, called *sdf-cons* is available to automatically synthesize these attributes when absent.

SDF is supported by two tools: the parse table generator PGEN, and the scannerless generalized parser SGLR. These tools were originally developed as components of the ASF+SDF Meta-Environment and are now separately available as stand-alone, reusable tools.

6.2.3 Code generation

From an SDF grammar, JJForester generates the following Java code:

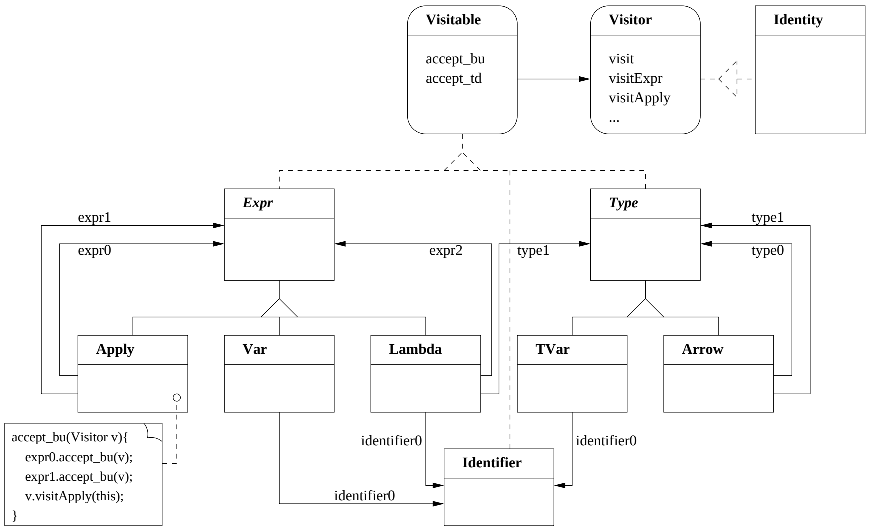
Class structure For each non-terminal symbol in the grammar, an *abstract* class is generated. For each production in the grammar with a *cons(..)* attribute, a *concrete* class is generated that extends the abstract class corresponding to the result non-terminal of the production. For example, Figure 6.3 shows a UML diagram of the code that JJForester generates for the grammar in Figure 6.2. The relationships between the abstract classes *Expr* and *Type*, and their concrete subclasses are known as the Composite pattern [GHJV94].

Lexical non-terminals and productions are treated slightly different: for each lexical non-terminal a class can be supplied by the user. Otherwise, this lexical non-terminal is replaced by the pre-defined non-terminal `Identifier`, for which a single concrete class is provided by JJForester. This is the case in our example. The `Identifier` contains a `String` representation of the actual lexical that is being modeled.

When the input grammar, unlike our example, contains complex symbols such as optionals or iterated symbols, additional classes are generated for them as well. The case study in Section 6.4 will illustrate this.

Parsers Also, for every non-terminal in the grammar, a parse method is generated for parsing a term (plain text) and constructing a tree (object structure). The actual parsing is done externally by SGLR. The parse method implements the Abstract Factory design pattern [GHJV94]; each non-terminal class has a parse method that returns an object of the type of one of the constructors for that non-terminal. Which object gets returned depends on the string that is parsed.

[2]The particular parse tree format emanated by SGLR contains for each node the production with which it was parsed. Consequently, our *implode* tool does not need the original grammar as input.



```
accept_bu(Visitor v){
    expr0.accept_bu(v);
    expr1.accept_bu(v);
    v.visitApply(this);
}
```

Figure 6.3: The UML diagram of the code generated from the grammar in Figure 6.2.

Constructor methods In the generated concrete classes, constructor methods are generated that build *language-specific* tree nodes from the generic tree that results from the call to the external parser.

Set and get methods In the generated concrete classes, set and get methods are generated to inspect and modify the fields that represent the subtrees. For example, the Apply class will have `getExpr0` and `setExpr0` methods for its first child.

Accept methods In the generated concrete classes, several accept methods are generated that take a Visitor object as argument, and apply it to a tree node. The accept method for each class dispatches its invocation to a visit method in the visitor that is specific to that class. Currently, two *iterating* accept methods are generated: `accept_td` and `accept_bu`, for top-down and bottom-up traversal, respectively. For the Apply class, the bottom-up accept method is shown in the Figure 6.3. We will additionally introduce an *non-iterating* accept method in Section 6.3.

Visitor interface and classes A Visitor interface is generated which declares a visit method for each production and each non-terminal in the grammar. Furthermore, it contains one method named `visit` which is useful for *generic* refinements (see below). Some default implementations of the Visitor interface are generated as well. First, a class named Identity is generated. Its visit methods are *non-iterating*: they make no calls to accept methods of children to obtain recursion. The default behavior offered by these generated visit methods is simply to do nothing. Second, a ToStringVisitor is generated which provides an updatable default pretty-printer for the input language. Finally, a class Fwd that implements the Visitor interface is generated. Its use will become clear in Section 6.3.

Together, the Visitor interface and the iterating accept methods in the various concrete classes implement a variant of the Visitor pattern [GHJV94], where the responsibility for iteration lies with the accept methods, not with the visit methods. We have chosen this variant for several reasons. First of all, it relieves the programmer who specializes a visitor from reconstructing the iteration behavior in the visit methods he redefines. This makes specializing visitors less involved and less error-prone. In the second place, it allows the traversal behavior (top-down or bottom-up) to be varied simply by selecting a different accept method.. In Section 6.3, we will explain a second, more powerful way to control iteration behavior, involving a non-iterating accept method in combination with visitor *combinators* that control traversal.

Apart from generating Java code, JJForester calls PGEN to generate a parse table from its input grammar. This table is used by SGLR which is called by the generated parse methods.

6.2.4 Programming against the generated code

The generated code can be used by a tool builder to construct tree traversals through the following steps:

1. Refine a visitor class by redefining one or more of its visit methods. As will be explained below, such refinement can be done at various levels of genericity, and in a step-wise fashion.
2. Start a traversal with the refined visitor by feeding it to the accept method of a tree node. Different accept methods are available to realize top-down or bottom-up traversals.

This method of programming traversals by refining (generated) visitors provides interesting possibilities for reuse. Firstly, many traversals only need to do something 'interesting' at a limited number of nodes. For these nodes, the programmer needs to supply code, while for all others the behavior of the generated visitor is inherited. Secondly, different traversals often share behavior for a number of nodes. Such common behavior can be captured in an initial refinement, which is then further refined in diverging directions. Unfortunately, Java's lack of multiple inheritance prohibits the converse: construction of a visitor by inheritance from

```
public class VarCountVisitor extends Identity {
    public int counter = 0;
    public void visitVar(Var x) {
        counter++;
    }
    public void visitTVar(TVar x) {
        counter++;
    }
}
```

Figure 6.4: Specific refinement: a visitor for counting variables.

two others. In Section 6.3 we will explain how visitor combinators can remedy this limitation. Thirdly, some traversal actions may be specific to nodes with a certain constructor, while other actions are the same for all nodes of the same type (non-terminal), or even for all nodes of any type. As the visitors generated by JJForester allow refinement at each of these levels of specificity, there is no need to repeat the same code for several constructors or types. We will explain these issues through a number of small examples.

Constructor-specific refinement Figure 6.4 shows a refinement of the Identity visitor class which implements a traversal that counts the number of variables occurring in a syntax tree. Both expression variables and type variables are counted. This refinement extends Identity with a counter field, and redefines the visit methods for Var and TVar such that the counter is incremented when such nodes are visited. The behavior for all other nodes is inherited from the generated Identity visitor: do nothing. Note that redefined methods need not restart the recursion behavior by calling an accept method on the children of the current node. The recursion is completely handled by the generated accept methods.

Generic refinement The refinement in the previous example is specific for particular node constructors. The visitors generated by JJForester additionally allow more generic refinements. Figure 6.5 shows refinements of the Identity visitor class that implement a more generic expression counter and a fully generic node counter. Thus, the first visitor counts all expressions, irrespective of their constructor, and the second visitor counts all nodes, irrespective of their type. No code duplication is necessary. Such per-sort refinements and fully generic refinements are possible, because in the generated Identity visitor, the specific methods such as `visitExpr` invoke the visit methods for sorts, which in turn call the generic method `visit`. In Section 6.3, we will show that such forwarding behavior can be captured in a separate visitor combinator.

Note that the visitors in Figures 6.4 and 6.5 can be refactored as refinements of a common initial refinement, say CountVisitor, which contains only the field

```
public class ExprCountVisitor extends Identity {
    public int counter = 0;
    public void visitExpr(Expr x) {
        counter++;
    }
}
```

```
public class NodeCountVisitor extends Identity {
    public int counter = 0;
    public void visit(Object x) {
        counter++;
    }
}
```

Figure 6.5: Generic refinement: visitors for counting expressions and nodes.

counter.

Step-wise refinement Visitors can be refined in several steps. For our example grammar, two subsequent refinements of the Identity visitor class are shown in Figure 6.6. The class GetVarsVisitor is a visitor for collecting all variables used in expressions. It is defined by extending the Identity class with a field `vars` initialized as the empty set of variables, and by redefining the visit method for the Var class to insert each variable it encounters into this set. The GetVarsVisitor is further refined into a visitor that collects *all* variables, by additionally redefining the visit methods for the Lambda class and the TVar class. These redefined methods insert type variables and bound variables in the set of variables `vars`. Finally, this second visitor can be unleashed on a tree using the `accept_bu` method. This is illustrated by an example of usage in Figure 6.6.

Of course, our running example does not mean to suggest that Java would be the ideal vehicle for implementing the lambda calculus. Our choice of example was motivated by simplicity and self-containedness. To compare, an implementation of the lambda calculus in the ASF+SDF Meta-Environment can be found in [DHK96]. In Section 6.4 we will move into the territory for which JJForester is intended: component-based development of program analyses and transformations for languages of non-trivial size.

6.2.5 Assessment of expressiveness

To evaluate the expressiveness of JJForester within the domain of language processing, we will assess which program transformation scenarios can be addressed with it. We distinguish three main scenarios:

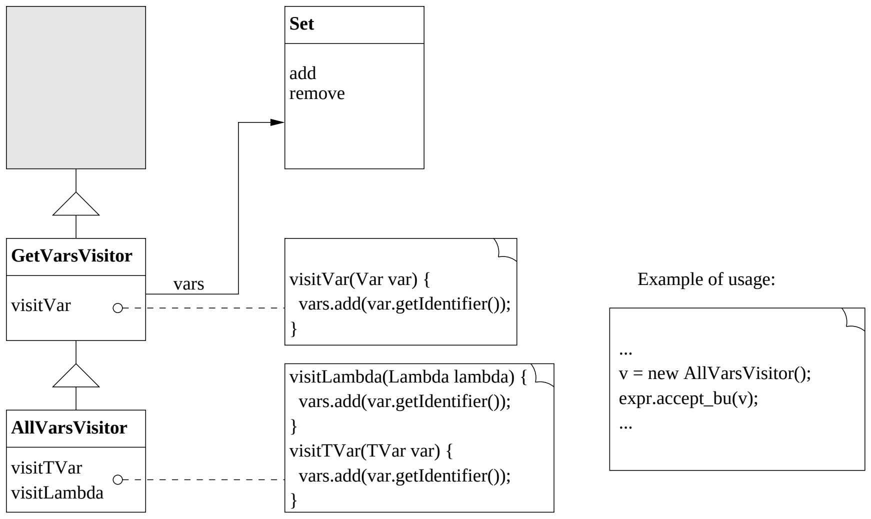


Figure 6.6: Step-wise refinement: visitors for collecting variables.

Analysis A value or property is distilled from a syntax tree. Type-checking is a prime example.

Translation A program is transformed into a program in a different language. Examples include generating code from a specification, and compilation.

Rephrasing A program is transformed into another program, where the source and target language coincide. Examples include normalization and renovation.

For a more elaborate taxonomy of program transformation scenarios, we refer to [JVV01, V^+]. The distinction between analysis and translation is not clear-cut. When the value of an analysis is highly structured, especially when it is an expression in another language, the label ‘translation’ is also appropriate.

The traversal examples discussed above are all tree analyses with simple accumulation in a state. Here, ‘simple’ accumulation means that the state is a value or collection to which values are added one at a time. This was the case both for the counting and the collecting examples. However, some analyses require more complex ways of combining the results of subtree traversals than simple accumulation. An example is pretty-printing, where literals need to be inserted *between* pretty-printed subtrees. In the case study, a visitor for pretty-printing will demonstrate that JJForester is sufficiently expressive to address such more complex analyses. Other examples are analyses that involve a notion of *scoping*. In section 6.3 a visitor for *free* variable analysis will demonstrate how such scoping issues can be handled with visitor combinators.

Translating transformations are also completely covered by JJForester's expressiveness. As in the case of analysis, the degree of reuse of generated visit methods can be very low. Here, however, the cause lies in the nature of translation, because it typically takes every syntactic construct into account. This is not always the case, for instance, when the translation has the character of an analysis with highly structured results. An example is program visualization where only dependencies of a particular kind are shown, e.g. module structures or call graphs.

In the object-oriented setting, a distinction needs to be made between destructive and non-destructive rephrasings. Destructive rephrasings are covered by JJForester. However, as objects can not modify their *self* reference, destructive modifications can only change subtrees and fields of the current node, but they can not replace the current node by another. Non-destructive rephrasings can be implemented by refining a traversal that clones the input tree. A visitor for tree cloning can be generated, as will be discussed in Section 6.5.3.

A special case of rephrasing is decoration. Here, the tree itself is traversed, but not modified except for designated attribute fields. Decoration is useful when several traversals are sequenced that need to share information about specific nodes. JJForester does not support decoration yet.

6.2.6 Limitations

The traversal support of JJForester, covered so far, caters for many basic traversal scenarios, but it is limited in a few respects.

Traversal control Traversal control is limited to selection between top-down or bottom-up accept methods. To obtain more complex traversal scenarios, the user must fall back to entangling traversal and node behavior in the visitor.

Visitor combination A new visitor can be constructed by refinement of a given one. But no support is present to combine the behavior of several given visitors. For instance, the `AllVarsVisitor` of Figure 6.6 can not be built by combining three visitors that each counts a different kind of variable.

Genericity Generic behavior implemented by refining the generic visit method is still class-hierarchy specific, because the visit interface is. For instance, the `NodeCountVisitor` of Figure 6.5 is specific to our little lambda language, and can not be applied to count nodes of syntax trees of other languages.

These limitations can be lifted with the *visitor combinators* of Chapter 5, as will be explained in the next section.

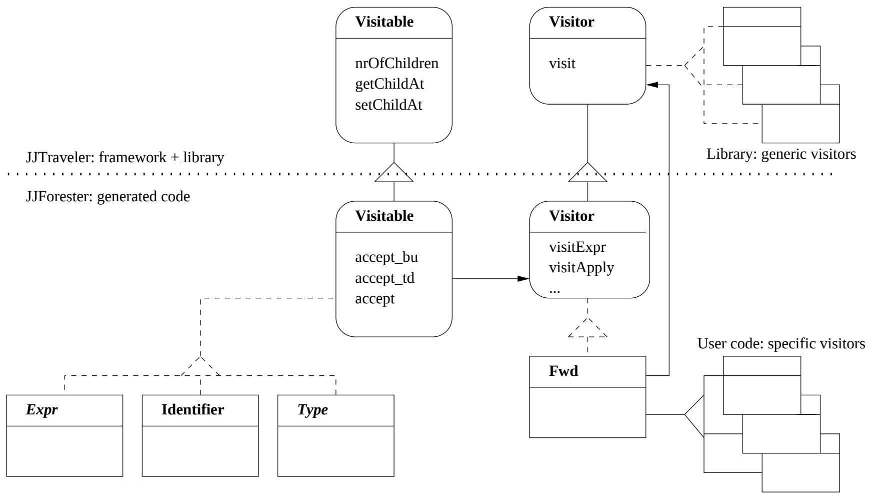


Figure 6.7: The architecture of JJTraveler in relation to JJForester. Class-hierarchy specific entities are shown below the dashed line.

6.3 JJTraveler

In Chapter 5 we introduced the notion of a *generic visitor combinator*, and we introduced JJTraveler: a combination of a framework and library that provide generic visitor combinators for Java.

Recall that visitor combinators are small, reusable classes that implement a *generic* visitor interface. Here 'generic' means: independent of any specific class hierarchy. Each combinator captures a basic piece of functionality. They can be composed in different constellations to build more complex visitors.

In this section, we explain how JJForester makes use of JJTraveler to offer more advanced traversal support, and to overcome the limitations of the basic traversal support that was explained in the previous section. To keep the discussion self-contained, we will recapitulate the essentials of JJTraveler and visitor combinators.

6.3.1 The architecture of JJTraveler

Figure 6.7 shows the architecture of JJTraveler and its relationship with JJForester. JJTraveler consists of a *framework* and a *library*.

Framework The framework consists of two interfaces, *Visitor* and *Visitable*. Unlike the interfaces of the same name generated by JJForester, these interfaces are not hierarchy-specific. The Visitor interface declares a single visit method, which takes *any* visitable object as argument. The Visitable interface declares three methods, called `getChildCount`, `getChildAt`, and `setChildAt`, that provide generic access to the children of *any* visitable object.

Library The library consists of a number of predefined visitor combinators. Each combinators implements the generic *Visitor* interface. An overview of the combinators is shown in Table 6.1. They will be explained in more detail below.

To use JJTraveler, one needs to instantiate the framework for the class hierarchy of a particular application. This can be done manually, but JJForester automates it. The Visitor and Visitable interfaces must be implemented. The Visitable interface is implemented by the various classes that model the grammar, as generated by JJForester. The Visitor interface is implemented by a number of generic Visitors from the library, and a JJForester generated Fwd combinator which knows about the structure of the grammar.

After instantiation, the user can do the following:

- Apply a generic visitor to an application-specific object with the generic `visit` method. Note that generic visitors do not need to be passed to an accept method to be applied, because they have only a single visit method, and no class-specific dispatch is needed.

- Turn a generic visitor into an application-specific one by supplying it as an argument to the *Fwd* combinator. The resulting specific visitor can be then be refined in constructor-specific or sort-specific manner.

- Supply an application-specific visitor as an argument to a generic visitor combinator.

Below, these types of usage will be explained and demonstrated for some concrete cases.

6.3.2 Generic visitor combinators

Table 6.1 shows high-level descriptions for an excerpt of JJTraveler's library of generic visitor combinators. A larger excerpt can be found in Table 5.2, and 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. The defined combinators can be described in terms of the basic ones as indicated in the overview. The implementation of both basic and defined combinators in Java is straightforward (for details see Chapter 5).

Combinator	Description of behavior
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
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)	$Choice(All(AllBottomUp(v)), v)$

Table 6.1: JJTraveler's library of generic visitor combinators (excerpt).

6.3.3 Building visitors from combinators

In order to demonstrate how visitor combinators can be used to build complex visitors with sophisticated traversal behavior, we will return to our example language, and develop a solution to the problem of finding *free* variables in a lambda term. The notion of *scope* plays an essential role in this problem.

To properly deal with scope, we can no longer rely on simple top-down or bottom-up traversal. Instead, we must stop the traversal and restart it in a new scope. For this purpose, we will develop a new generic visitor combinator:

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

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 6.8. Note that the second constructor method provides a shorthand for calling the first constructor with $Identity$ as second argument.

Given the $TopDownWhile$ combinator, we can compose a visitor for free variable analysis by specialization of the $GetVarsVisitor$ of Figure 6.6. The specialized visitor is shown in Figure 6.9. Recall that the $GetVarsVisitor$ accumulates variables in a `vars` field of type `Set`. Additionally, the $FreeVarsVisitor$ redefines the visit method for lambda expressions. In this method, four things happen:

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

Figure 6.8: Encoding of the $TopDownWhile$ combinator in Java.

```
public class FreeVarsVisitor extends GetVarsVisitor {
    public void visit_Lambda(Lambda lambda) {
        Expr body = lambda.getExpr();
        Set freeInBody = freeVars(body);
        Identifier bindingVar = lambda.getIdentifier();
        freeInBody.remove(bindingVar);
        vars.addAll(freeInBody);
        throw new VisitFailure();
    }
    public static Set freeVars(Expr e)
      throws VisitFailure {
        FreeVarsVisitor v = new FreeVarsVisitor();
        (new TopDownWhile(v)).visit(e);
        return v.getVars();
    }
}
```

Figure 6.9: A visitor for free variable analysis.

(i) the free variable analysis is recursively carried out for the body of the lambda via the method `freeVars`, (ii) the binding variable of the lambda expression is subtracted from the resulting set of free variables, (iii) the remaining free variables are added to the current local set `vars`, and (vi) the traversal is stopped by raising an exception. In the function `freeVars`, the $TopDownWhile$ combinator is applied to a new $FreeVarsVisitor$ to (re)start the traversal.

In the case study to be presented in Section 6.4, further examples of using visitor combinators will be given.

6.3.4 Evaluation

In Section 6.2.6 we listed some limitations of the basic traversal support provided by JJForester, with respect to *traversal control*, *visitor composition*, and *genericity*. The additional traversal support realized by JJForester's link to JJTraveler removes these limitations.

Traversal control JJTraveler's library provides combinators for a variety of generic traversal scenarios in its library. Further (generic) scenarios can be programmed as needed by combining (basic) combinators.

Visitor combination Application-specific visitors can be supplied as arguments to generic visitor combinators to build more complex visitors.

Genericity Visit behavior (traversing or non-traversing) that is generic in nature can be implemented with reference only to the generic framework and library of JJTraveler.

There is also a down-side to the additional power of visitor combinators offered by JJTraveler. When visitors are not monolithic, but built out of combinators, their performance suffers, due to the forwarding of control between the various combinators. Also, visitor combinators are conceptually more challenging to the object-oriented programmer than plain visitors. With these trade-offs in mind, JJForester supports both styles of visitor programming.

6.4 Case study

Now that we have explained the workings of JJForester, we will show how it is used to build a program analyzer for an actual language. In particular, this case study concerns a static analyzer for the ToolBus [BK98] script language. In Section 6.4.1 we describe the situation from which a need for a static analyzer emerged. In Section 6.4.2 the language to be analyzed is briefly explained. Finally, Section 6.4.3 describes in detail what code needs to be supplied to implement the analyzer.

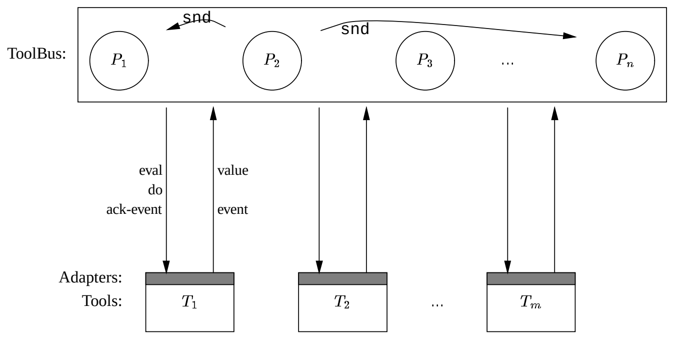


Figure 6.10: The Toolbus architecture. Tools are connected to the bus through adapters. Inside the bus, several processes run in parallel. These processes communicate with each other and the adapters according to the protocol defined in a T-script.

6.4.1 The Problem

The ToolBus is a coordination language which implements the idea of a software bus. It allows components (or *tools*) to be “plugged into” a bus, and to communicate with each other over that bus. Figure 6.10 gives a schematic overview of the ToolBus. The protocol used for communication between the applications is not fixed, but is programmed through a ToolBus script, or T-script.

A T-script defines one or more processes that run inside the ToolBus in parallel. These processes can communicate with each other, either via synchronous point-to-point communication, or via asynchronous broadcast communication. The processes can direct and activate external components via *adapters*, small pieces of software that translate the ToolBus’s remote procedure calls into calls that are native to the particular software component that needs to be activated. Adapters can be compiled into components, but off-the-shelf components can be used, too, as long as they possess some kind of external interface.

Communication between processes inside the ToolBus does not occur over named channels, but through pattern matching on terms. Communication between processes occurs when a term sent by one matches the term that is expected by another. This will be explained in more detail in the next section. This style of communication is powerful, flexible and convenient, but tends to make it hard to pinpoint errors in T-scripts. To support the T-script developer, the ToolBus runtime system provides an interactive visualizer, which shows the communications taking place in a running ToolBus. Though effective, this debugging process is tedious and slow, especially when debugging systems with a large number of processes.

To complement the runtime visualizer, a *static* analysis of T-scripts is needed to support the T-script developer. Static analysis can show that some processes can never communicate with each other, that messages that are sent can never be received (or vice versa), or that two processes that should not communicate with each other may do so anyway. Using JJForester, such a static analyzer is constructed in Section 6.4.3.

6.4.2 T-scripts explained

T-scripts are based on ACP (Algebra of Communicating Processes) [BV95]. They define communication protocols in terms of *actions*, and operations on these actions. We will be mainly concerned with the communication actions, which we will describe below. Apart from these, there are assignment actions, conditional actions and basic arithmetic actions. The action operators include sequential composition ($a.b$), non-deterministic choice ($a + b$), parallel composition ($a \mid\mid b$), and repetition ($a * b$, a is repeated zero or more times, and finally b is executed). The deadlock action ($delta$) always fails. The full specification of the ToolBus script language can be found in [BK94].

The T-script language offers actions for communication between processes and tools, and for synchronous and asynchronous communication between processes. For the purposes of this chapter we will limit ourselves to the most commonly used *synchronous* actions; for brevity, asynchronous actions are not covered. The synchronous actions are `snd-msg(T)` and `rec-msg(T)` for sending and receiving messages, respectively. These actions are parameterized with arbitrary data `T`, represented as ATerms [BJKO00]. A successful synchronous communication occurs when a term that is sent matches a term that is received. For instance, the closed term `snd-msg(f(a))` can match the closed term `rec-msg(f(a))` or the open term `rec-msg(f(T?))`. At successful communication, variables in the data of the receiving process are instantiated according to the match.

To illustrate, a small example T-script is shown in Figure 6.11. This example contains only processes. In a more realistic situation these processes would communicate with external tools, for instance to get the input of the initial value, and to actually activate the gas pump. The script's last statement is a mandatory `toolbus(..)` statement, which declares that upon startup the processes GasStation, Pump, Customer and Operator are all started in parallel. The variables `C` and `D` in the process definitions stand for the customer's process-id and an amount of money (dollars), respectively. The first action of all processes, apart from Customer, is a `rec-msg` action. This means that those processes will block until an appropriate communication is received. The Customer process starts by doing two assignment statements: `process-id` (a built-in variable that contains the identifier of the current process) is assigned to `C`, and 10 to `D`. The first communication action performed by Customer is a `snd-msg` of the term `prepay(D,C)`. This term is received by the GasStation process, which in turn sends the term

```
process GasStation is
let
D: int, C: int
in
( rec-msg(prepay(D?,C?)).
  snd-msg(request(D,C))
||rec-msg(schedule(D?,C?)).
  snd-msg(activate(D)).
  snd-msg(okay(C))
||rec-msg(turn-on).
  snd-msg(on)
||rec-msg(report(D?)).
  snd-msg(stop).
  snd-msg(result(D))
||rec-msg(remit(D?)).
  snd-msg(change(D))
)*
delta
endlet

process Operator is
let C: int, D: int,
    Payment: int, Amount: int
in
( rec-msg(request(D?,C?)).
  Payment := D.
  snd-msg(schedule(Payment,C)).
  rec-msg(result(D?)).
  Amount := sub(Payment,D).
  snd-msg(remit(Amount))
) *
delta
endlet
```

```
process Pump is
let D: int
in
( rec-msg(activate(D?)).
  rec-msg(on).
  snd-msg(report(D))
) *
delta
endlet

process Customer is
let
C: int, D: int
in
C := process-id.
D := 10.
snd-msg(prepay(D,C)).
rec-msg(okay(C)).
snd-msg(turn-on).
printf(
 "Customer %d using pump\n",
  C).
rec-msg(stop).
rec-msg(change(D?)).
printf(
 "Customer %d got $%d change\n",
  C, D)
endlet

toolbus(GasStation,Pump,
  Customer,Operator)
```

Figure 6.11: The T-script for the gas station with control process.

`request(D,C)` message. This is received by Operator, and so on.

The script writer can use the mechanism of communication through term matching to specify that any one of a number of processes should receive a message, depending on the state they are in, and the sending process does not need to know which. It just sends out a term into the ToolBus, and any one of the accepting processes can "pick it up". Unfortunately, when incorrect or too general terms are specified in a `rec-msg` action, communication will not occur as expected, and the exact cause will be difficult to trace. The static analyzer developed in the next section is intended to solve this problem.

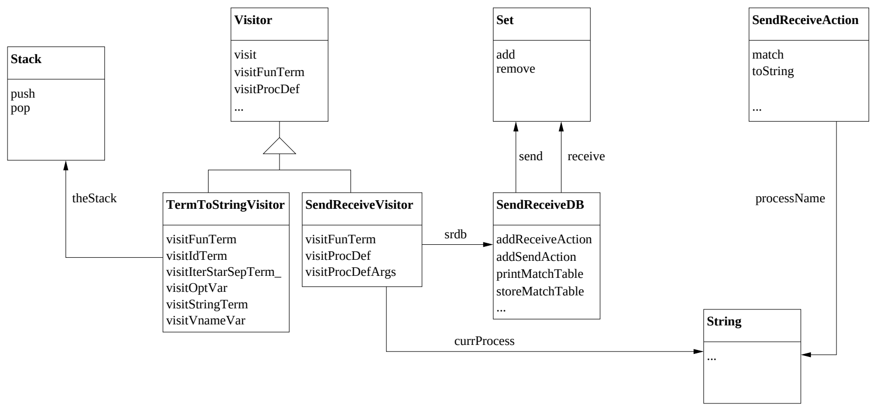


Figure 6.12: UML diagram of the ToolBus analyzer.

6.4.3 Analysis using JJForester

We will first sketch the outlines of the static analysis algorithm that we implemented. It consists of two phases: collection and matching. In the collection phase, *all* send and receive actions in the T-script are collected into a (internal, non-persistent) database. In the matching phase, the send and receive actions in the database are matched to obtain a table of potential matching events, which can either be stored in a file, or in an external, persistent relational database. To visualize this table, we use the back-end tools of a documentation generator we developed earlier (DocGen [DK99a]).

We used JJForester to implement the parsing of T-scripts and the representation and traversal of T-script parse trees. To this end, we ran JJForester on the grammar of the ToolBus[3] which contains 35 non-terminals and 80 productions (both lexical and context-free). From this grammar, JJForester generated 23 non-terminal classes, 64 constructor classes, and 1 visitor class, amounting to a total of 4221 lines of Java code.

We will now explain in detail how we programmed the two phases of the analysis. Figure 6.12 shows a UML diagram of the implementation.

The collection phase

We implemented the collection phase as a top-down traversal of the syntax tree with a visitor called SendReceiveVisitor. This refinement of the Visitor class has two kinds of state: a database for storing send and receive actions, and a field that indicates the name of the process currently being analyzed. Whenever a term with

[3]This SDF grammar can be downloaded from the online Grammar Base [GB].

context-free syntax
"process" ProcessName "is" ProcessExpr
$\rightarrow$ ProcessDef {**cons**("procDef")}
"process" ProcessName "(" {VarDecl ","}* ")" "is" ProcessExpr
$\rightarrow$ ProcessDef {**cons**("procDefArgs")}

Figure 6.13: The syntax of process definitions.

```
public void visitProcDef(procDef definition) {
  currProcess = definition.getIdentifier0().toString();
}
public void visitProcDefArgs(procDefArgs definition) {
  currProcess = definition.getIdentifier0().toString();
}
```

Figure 6.14: Specialized visit methods to extract process definition names.

outermost function symbol snd-msg or rec-msg is encountered, the visitor will add a corresponding action to the database, tagged with the current process name. The current process name is set whenever a process definition is encountered during traversal. Since sends and receives occur only *below* process definitions in the parse tree, the top-down traversal strategy guarantees that the current process name field is always correctly set when it is needed to tag an action.

To discover which visit methods need to be redefined in the SendReceiveVisitor, the ToolBus grammar needs to be inspected. To extract process definition names, we need to know which syntactic constructs are used to declare these names. The two relevant productions are shown in Figure 6.13. So, in order to extract process names, we need to redefine `visitProcDef` and `visit-ProcDefArgs` in our specialized SendReceiveVisitor. These redefinitions are shown in Figure 6.14. Whenever the built-in iterator comes across a node in the tree of type `procDef`, it will call our specialized `visitProcDef` with that `procDef` as argument. From the SDF definition in Figure 6.13 we learn that a `procDef` has two children: a ProcessName and a ProcessExpr. Since ProcessName is a *lexical* non-terminal, and we chose to have JJForester identify all lexical non-terminals with a single type `Identifier`, the Java class `procDef` has a field of type `Identifier` and one of type `ProcessExpr`. Through the `getIdentifier0()` method we get the actual process name which gets converted to a String so it can be assigned to `currProcess`.

Now that we have taken care of extracting process names, we need to address the collection of communication actions. The ToolBus grammar allows for arbitrary terms ('Atoms' in the grammar) as actions. Their syntax is shown in Figure 6.15.

```
context-free syntax
  Vname                    → Var      {cons("vnameVar")}
  Var                      → GenVar   {cons("var")}
  Var "?"                  → GenVar   {cons("optVar")}
  GenVar                   → Term     {cons("genvarTerm")}
  Id                       → Term     {cons("idTerm")}
  Id "(" TermList ")"      → Term     {cons("funTerm")}
  {Term ","}*              → TermList {cons("termStar")}
  Term                     → Atom     {cons("termAtom")}
```

Figure 6.15: Syntax of relevant ToolBus terms.

```java
public void visitFunTerm(funTerm term) {
  SendReceiveAction action
    = new SendReceiveAction(currProcess, term.getTermlist1());
  if (term.getIdentifier0().equals("snd-msg")) {
    srdb.addSendAction(action);
  } else if (term.getIdentifier0().equals("rec-msg")) {
    srdb.addReceiveAction(action);
  }
}
```

Figure 6.16: The visit method for send and receive messages.

Thus, send and receive actions are not distinct syntactic constructs, but they are functional terms (`funTerms`) where the `Id` child has value `snd-msg` or `rec-msg`. Consequently, we need to redefine the `visitFunTerm` method such that it inspects the value of its first child to decide if and how to collect a communication action. Figure 6.16 shows the redefined method.

The visit method starts by constructing a new `SendReceiveAction`. This is an object that contains the term that is being communicated and the process that sends or receives it. The process name is available in the `SendReceive-Visitor` in the field `currProcess`, because it is put there by the `visit-ProcDef` methods we just described. The term that is being communicated can be selected from the `funTerm` we are currently visiting. From the SDF grammar in Figure 6.15 it follows that the term is the second child of a `funTerm`, and that it is of type `TermList`. Therefore, the method `getTermlist1` will return it.

The newly constructed action is added to the database as a send action, a receive action, or not at all, depending on the first child of the `funTerm`. This child is of lexical type `Id`, and thus converted to an `Identifier` type in the generated Java classes. The `Identifier` class contains an `equals(String)` method, so we use string comparison to determine whether the current `funTerm`

```
public static void main(String[] args) throws ParseException {
  String inFile = args[0];
  Tscript theScript = Tscript.parse(inFile);
  SendReceiveVisitor srvisitor = new SendReceiveVisitor();
  theScript.accept_td(srvisitor);       // collection phase
  srvisitor.srdb.constructMatchTable(); // matching phase
}
```

Figure 6.17: The main() method of the ToolBus analyzer.

has “snd-msg” or “rec-msg” as its function symbol.

Now that we have built the specialized visitor to perform the collection, we still need to activate it. Before we can activate it, we need to have parsed a T-script, and built a class structure out of the parse tree for the visitor to operate on. This is all done in the `main()` method of the analyzer, as shown in Figure 6.17. The main method shows how we use the generated parse method for `Tscript` to build a tree of objects. Tscript.parse() takes a filename as an argument and tries to parse that file as a Tscript. If it fails it throws a ParseException that contains the location of the parse error. If it succeeds it returns a `Tscript`. We then construct a new `SendReceiveVisitor` as described in the previous section. The `Tscript` is subsequently told to accept this visitor, and, as described in Section 6.2.4 iterates over all the nodes in the tree and calls the specific visit methods for each node. When the iterator has visited all nodes, the `SendReceiveVisitor` contains a filled `SendReceiveDb`. The results in this database object can then be processed further, in the matching phase. In our case we call the method `constructMatchTable()` which is explained below.

The collection phase – using JJTraveler

The implementation of the collection phase given in the previous section is somewhat naive. It uses a single top-down traversal strategy to visit *all* nodes. Since send and receive actions are always top-level functional terms, there is no need to traverse into other functional terms. Therefore, a more sophisticated traversal scenario is desirable that stops descending where possible.

Figure 6.18 shows an implementation of the collection phase using JJTraveler. The main method differs from the previous version in three respects. First of all, the action to be performed at each node is implemented by a different visitor class, called *SendReceiveTraveler*. Second, we do not rely on the accept method for iteration, but we use the TopDownWhile visitor combinator introduced in Section 6.3.3. Finally, we call the visit method of the visitor, and pass the script as its argument. Recall that generic visitors, such as TopDownWhile, need not be passed via an accept method; their only visit method can be called directly.

Figure 6.19 shows part of the implementation of *SendReceiveTraveler*. Previ-

```
public static void main(String[] args) throws ParseException {
  String inFile = args[0];
  Tscript theScript = Tscript.parse(inFile);
  SendReceiveTraveler srvisitor = new SendReceiveTraveler();
  jjtraveler.Visitor v = new TopDownWhile(srvisitor);
  v.visit(theScript);
  srvisitor.srdb.constructMatchTable(); // matching phase
}
```

Figure 6.18: The main() method of the ToolBus analyzer using JJTraveler.

```
public class SendReceiveTraveler extends Fwd {
  public SendReceiveTraveler() { super(new Identity()); }
  public void visitFunTerm(funTerm term)
    throws jjtraveler.VisitFailure {
    SendReceiveAction action
      = new SendReceiveAction(currProcess, term.getTermlist1());
    if (term.getIdentifier0().equals("snd-msg")) {
      srdb.addSendAction(action);
    } else if (term.getIdentifier0().equals("rec-msg")) {
      srdb.addReceiveAction(action);
    }
    throw new jjtraveler.VisitFailure();
  }
}
```

Figure 6.19: The visitor using JJTraveler.

ously we explained that JJForester generates a *Fwd* combinator to use a generic visitor as an application-specific one. Here we see that *SendReceiveTraveler* extends the *Fwd* combinator to which the *Identity* combinator is passed as the generic visitor argument (first method). The relevant visit method shown here is `visitFunTerm()` as it is the only method that is different with respect to the *SendReceiveVisitor*. The difference between the two methods is that the method in the traveler *fails* after it has encountered a functional term. This failure indicates that the traversal should be stopped. Thus, when the visitor encounters a functional term, it checks whether this term is a send or receive term, if so, it stores the corresponding *SendReceiveAction*. Either way it throws a *VisitFailure* exception.

As is shown in Figure 6.18 we pass the *SendReceiveTraveler* to the *TopDownWhile* combinator, which is responsible for traversing the tree. As was demonstrated in Section 6.3.3 the *TopDownWhile* combinator will perform a top-down traversal as long as it does not encounter a failure. When it encounters a failure, it will stop the traversal at the node that failed, apply its second argument, and then continue with the next sibling of the current node. In the current case, the traversal does not need to be restarted. Therefore, we used the unary constructor of *TopDownWhile*, which silently supplies *Identity* as a second argument.

The composed visitor indeed behaves as we wanted. Since the default traversal lets all visit methods succeed, we are guaranteed to descend to the level of funTerms. Once it reaches the funTerms the visitor fails (by throwing the *VisitFailure* exception). As a consequence, the traversal will not go deeper.

It turns out that, using this more sophisticated traversal on typical ToolBus scripts, the number of visited nodes is reduced by up to 70%.

The matching phase

In the matching phase, the send and receive actions collected in the `SendReceiveDb` are matched to construct a table of potential communication events, which is then printed to a file or stored in a relational database. We will not discuss the matching itself in great detail, because it is not implemented with a visitor. A visitor implementation would be possible, but clumsy, since two trees need to be traversed simultaneously. Instead it is implemented with nested iteration over the sets of send and receive actions in the database, and simple case discrimination on terms. The result of matching is a table where each row contains the process names and data of a pair of matching send and receive actions.

We focus on an aspect of the matching phase where a visitor *does* play a role. When writing the match table to file, the terms (data) it contains need to be pretty-printed, i.e. to be converted to `String`. We implemented this pretty-printer with a bottom-up traversal with the `TermToStringVisitor`. We chose not to use generated `toString` methods of the constructor classes, because using a visitor leaves open the possibility of refining the pretty-print functionality.

Note that pretty-printing a node may involve inserting literals before, in between, and after its pretty-printed children. In particular, when we have a list of terms, we would like to print a “,” between children. To implement this behavior, a visitor with a single `String` field in combination with a top-down or bottom-up accept method does not suffice. If JJForester would generate *iterating* visitors and *non-iterating* accept methods, this complication would not arise. Then, literals could be added to the `String` field in between recursive calls.

We overcome this complication by using a visitor with a *stack* of strings as field, in combination with the bottom-up accept method. The visit method for each leaf node pushes the string representation of that leaf on the stack. The visit method for each internal node pops one string off the stack for each of its children, constructs a new string from these, possibly adding literals in between, and pushes the resulting string back on the stack. When the traversal is done, the user can pop the last element off the stack. This element is the string representation of the visited term. Figure 6.20 shows the visit method in the `TermToStringVisitor` for lists of terms separated by commas[4]. In this method, the Vector containing the term

[4]The name of the method reflects the fact that this is a visit method for the symbol `{Term ","}*`, i.e. the list of zero or more elements of type Term, separated by commas. Because the comma is an

```
public void visitIterStarSepTerm_(iterStarSepTerm_ terms) {
  Vector v = terms.getTerm0();
  String str = "";
  for (int i = 0; i < v.size(); i++){
    if (i != 0) {
      str += ",";
    }
    str += (String) theStack.pop();
  }
  theStack.push(str);
}
```

Figure 6.20: Converting a list of terms to a string.

list is retrieved, to get the number of terms in this list. This number of elements is then popped from the stack, and commas are placed between them. Finally the new string is placed back on the stack. In the conclusion we will return to this issue, and discuss alternative and complementary generation schemes that make implementing this kind of functionality more convenient.

After constructing the matching table, the `constructMatchTable` method writes the table to a file or stores it in an SQL database, using JDBC (Java Database Connectivity). We used a visualization back-end of the documentation generator DocGen to query the database and generate a *communication* graph. The result of the full analysis of the T-script in Figure 6.11 is shown in Figure 6.21.

Evaluation of the case study

We conducted the ToolBus case study to learn about feasibility, productivity, performance, and connectivity issues surrounding JJForester. Below we briefly discuss our preliminary conclusions. In the upcoming Chapter, we describe a more involved case study involving procedure reconstruction for Cobol programs. This case study also corroborates our findings.

Feasibility At first glance, the object-oriented programming paradigm may seem to be ill-suited for language processing applications. Terms, pattern-matching, many-sorted signatures are typically useful for language processing, but are not native to an object-oriented language like Java. More generally, the reference semantics of objects seems to clash with the value semantics of terms in a language. Thus, in spite of Java's many advantages with respect to e.g. portability, maintainability, and reuse, its usefulness in language processing is not evident.

The case study, as well as the techniques for coping with traversal scenarios outlined in Section 6.2, demonstrate that object-oriented programming *can* be ap-

illegal character in a Java identifier, it is converted to an underscore in the method name. When several sorts are mapped to the same name, conflicts are prevented by adding additional underscores.

Sender		Receiver	
Pump	report(D)	GasStation	report(D?)
GasStation	change(D)	Customer	change(D?)
Customer	prepay(D,C)	GasStation	prepay(D?,C?)
GasStation	okay(C)	Customer	okay(C)
Operator	remit(Amount)	GasStation	remit(D?)
GasStation	result(D)	Operator	result(D?)
GasStation	activate(D)	Pump	activate(D?)
GasStation	stop	Customer	stop
Customer	turn-on	GasStation	turn-on
Operator	schedule(Payment,C)	GasStation	schedule(D?,C?)
GasStation	request(D,C)	Operator	request(D?,C?)
GasStation	on	Pump	on

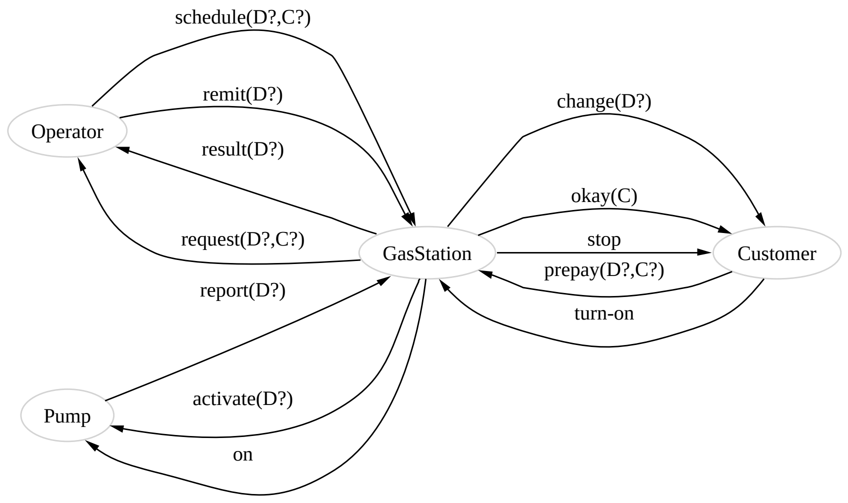


Figure 6.21: The analysis results for the input file from Figure 6.11.

plied usefully to language processing problems. In fact, the support offered by JJForester makes object-oriented language processing not only feasible, but even easy.

Productivity Recall that the Java code generated by JJForester from the ToolBus grammar amounts to 4221 lines of code. By contrast, the user code we developed to program the T-script analyzer consists of 323 lines. Thus, 93% of the application was generated, while 7% is hand-written.

These figures indicate that the potential for increased development productivity is considerable when using JJForester. Of course, actual productivity gains are highly dependable on which program transformation scenarios need to be addressed (see Section 6.2.5). The productivity gain is largely attributable to the support for generic traversals.

Components and connectivity Apart from reuse of generated code, the case study demonstrates reuse of standard Java libraries and of external (non-Java) tools. Examples of such tools are PGEN, SGLR and *implode*, an SQL database, and the visualization back-end of DocGen. Externally, the syntax trees that JJForester operates upon are represented in the common exchange format ATerms. This exchange format was developed in the context of the ASF+SDF Meta-Environment, but has been used in numerous other contexts as well. In Chapter 2 we advocated the use of grammars as tree type definitions that fix the interface between language tools. JJForester implements these ideas, and can interact smoothly with tools that do the same. The transformation tool bundle XT [JVV01] contains a variety of such tools.

Performance To get a first indication of the time and space performance of applications developed with JJForester, we have applied our T-script analyzer to a script of 2479 lines. This script contains about 40 process definitions, and 700 send and receive actions. We used a machine with Mobile Pentium processor, 64Mb of memory, running at 266Mhz. The memory consumption of this experiment did not exceed 6Mb. The runtime was 69 seconds, of which 9 seconds parsing, 55 seconds implosion, and 5 seconds to analyze the syntax tree. A safe conclusion seems to be that the Java code performs acceptably, while the implosion tool needs optimization. Needless to say, larger applications and larger code bases are needed for a good assessment.

6.5 Concluding remarks

6.5.1 Contributions

In this chapter we set out to combine SDF support of the ASF+SDF Meta-Environment with the general-purpose object-oriented programming language Java. To this end we designed and implemented JJForester, a parser and visitor generator for Java that takes SDF grammars as input. To support generic traversals, JJForester generates accept methods and visitor classes. We discussed techniques for programming against the generated code, and we demonstrated these in detail in a case study. We have assessed the expressivity of our approach in terms of the program-transformation scenarios that can be addressed with it. Based on the case study, we evaluated the approach with respect to productivity and performance issues.

6.5.2 Related Work

A number of parser generators, "tree builders", and visitor generators exist for Java. JavaCC is an LL parser generator by Metamata/Sun Microsystems. Its input format is not modular, it allows Java code in semantic actions, and it separates parsing from lexical scanning. JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source. The Java Tree Builder (JTB) is another front-end for JavaCC for tree building and visitor generation. JTB generates two iterating (bottom-up) visitors, one with and one without an extra argument in the visit methods to pass objects down the tree. A version of JTB for GJ (Generic Java) exists which takes advantages of type parameters to prevent type casts. Demeter/Java is an implementation of adaptive programming [PXL95] for Java. It extends the Java language with a little (or domain-specific) language to specify traversal strategies, visitor methods, and class diagrams. Again, the underlying parser generator is JavaCC. The SmartTools system supports language tool development using XML and Java [A$^+$02]. From an *abstract* syntax definition, it generates a development environment that includes a structure editor and some basic visitors. If the user specifies additional syntactic sugar, a parser and pretty-printer are generated as well. In a little language designed for this purpose, the user can specify visitor *profiles* to obtain more sophisticated visitors. JJForester's main improvement with respect to these approaches is the support of *generalized* LR parsing. Concerning traversals, JJForester is different from JJTree and JTB, because it generates both iterating and non-iterating accept methods and supports the use of visitor combinators to obtain full traversal control. Demeter and SmartTools provide more traversal control than the plain visitor pattern via little traversal languages. JJForester is less ambitious and more lightweight than Demeter or SmartTools, which are rather elaborate programming systems rather than code-generators.

ASDL (Abstract Syntax Definition Language [WAKS97]) comes with a visitor generator for Java (and other languages). It generates non-iterating visitors and non-iterating accept methods. Thus, traversals are not supported. ASDL does not incorporate parsing or parser generation; it only addresses issues of *abstract* syntax.

In other programming paradigms, work has been done on incorporating support for SDF and traversals. Previously, we combined the SDF support of the ASF+SDF Meta-Environment with the functional programming language Haskell [KLV00]. In this approach, traversal of syntax trees is supported either with updatable, many-sorted folds and fold combinators (see Chapter 3), or with generic function combinators (see Chapter 4). Recently, support for generic traversals has been added to the ASF interpreter [BKV02]. These traversals allow concise specification of many-sorted analyses and rephrasing transformations. Stepwise refinement or generic refinement of such traversals is not supported. Stratego [VBT99] is a language for term rewriting with strategies. It offers a suite of primitives that allow programming of (as yet untyped) generic traversals. Stratego natively supports ATerms. It is used extensively in combination with the SDF components of the ASF+SDF Meta-Environment.

6.5.3 Future Work

Concrete syntax and subtree sharing Currently, JJForester only supports processing of *abstract* syntax trees. Though the parser SGLR emits full *concrete* parse trees, these are imploded before being consumed by JJForester. For many program transformation problems it is desirable, if not essential, to process concrete syntax trees. A prime example is software renovation [DKV99], which requires preservation of layout and comments in the source code. The ASF+SDF Meta-Environment supports processing of concrete syntax trees which contain layout and comments. In order to broaden JJForester's applicability, and to ensure its smooth interoperation with components developed in ASF, we are considering to add concrete syntax support.

When concrete syntax is supported, the trees to be processed are significantly larger. To cope with such trees, the ASF+SDF Meta-Environment uses the ATerm library which implements maximal subtree sharing, which leads to significant space efficiency. As a Java implementation of the ATerm library is available, subtree sharing support could be added to JJForester. We would like to investigate the repercussions of such a change to tree representation for the expressiveness and performance of JJForester.

Decoration and aspect-orientation Adding a Decoration field to all generated classes would make it possible to store intermediate results inside the object structure in between visits. This way, a first visitor could calculate some data and store it in the object structure, and then a second visitor could “harvest” these data and perform some additional calculation on them.

More generally, we would like to experiment with aspect-oriented techniques [KL$^+$97] to customize or adapt generated code. Adding decoration fields to generated classes would be an instance of such customization.

Chapter 7

Building Program Understanding Tools Using Visitor Combinators

In this chapter, we apply the object-oriented support for generic traversal presented in Chapters 5 and 6 to the construction of program understanding tools.

Program understanding tools manipulate program representations, such as abstract syntax trees, control-flow graphs, or data-flow graphs. This chapter deals with the use of *visitor combinators* to conduct such manipulations. Visitor combinators are an extension of the well-known visitor design pattern. They are small, reusable classes that carry out specific visiting steps. They can be composed in different constellations to build more complex visitors. We evaluate the expressiveness, reusability, ease of development, and applicability of visitor combinators to the construction of program understanding tools. To that end, we conduct a case study in the use of visitor combinators for control flow analysis and visualization as used in a commercial Cobol program understanding tool.

This chapter was based on [DV02b].

7.1 Introduction

Program analysis and source models Program analysis is a crucial part of many program understanding tools. Program analysis involves the construction of source models from the program source text and the subsequent analysis of these models. Depending on the analysis problem, these source models might be represented by tables, trees, or graphs.

More often than not, the models are obtained through a sequence of steps. Each step can construct new models or refine existing ones. Usually, the first model is an (abstract) syntax tree constructed during parsing, which is then used to derive graphs representing, for example, control or data flow.

Visiting source models The intent of the visitor design pattern is to "represent an operation to be performed on the elements of an object structure. A visitor lets you define a new operation without changing the classes of the elements on which it operates" [GHJV94]. Often, visitors are constructed to *traverse* an object structure according to a particular built-in strategy, such as *top-down*, *bottom-up*, or *breadth-first*.

A typical example of the use of the visitor pattern in program understanding tools involves the traversal of abstract syntax trees. The pattern offers an abstract class *Visitor*, which defines a series of methods that are invoked when nodes of a particular type (expressions, statements, *etc.*) are visited. A concrete *Visitor* subclass refines these methods in order to perform specific actions when accepted by a given syntax tree.

Visitors are useful for analysis and transformation of source models for several reasons. Using visitors makes it easy to traverse structures that consist of many different kinds of nodes, while conducting actions on only a selected number of them. Moreover, visitors help to separate traversal from representation, making it possible to use a single source model for various sorts of analysis.

Visitor Combinators In Chapter 5, visitor *combinators* have been proposed 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.

Goals of the chapter The concept of visitor combinators is based on solid theoretical ground, and it promises to be a powerful implementation technique for processing source models in the context of program analysis and understanding. Now this concept needs to be put to the test of practice.

We have implemented ControlCruiser, a tool for analyzing and visualizing intra-program control-flow for Cobol. In this chapter, we explain by reference

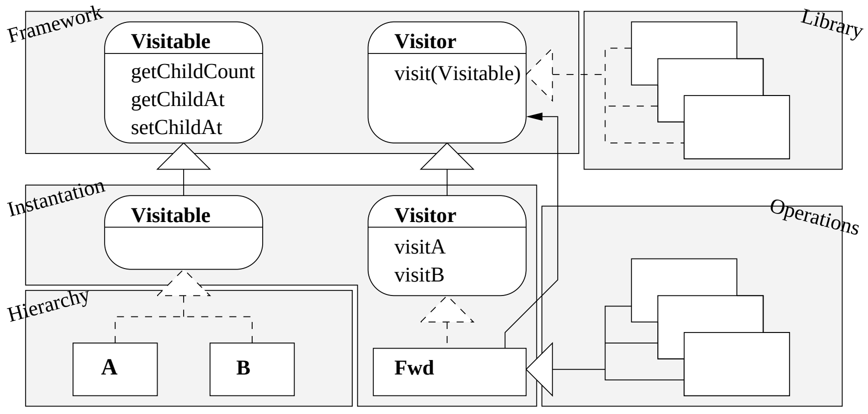


Figure 7.1: The architecture of JJTraveler.

to ControlCruiser how visitor combinators can be used to develop program understanding tools. We discuss design tactics, programming techniques, unit testing, implementation trade-offs, and other engineering practices related to visitor combinator development. Finally, we asses the risks and benefits of adopting visitor combinators for building program understanding tools.

7.2 Visitor Combinators

Visitor combinator programming was introduced in Chapter 5 and is supported by JJTraveler: a combination of a framework and library that provides *generic visitor combinators* for Java. This section briefly recapitulates the key elements of JJTraveler (readers familiar with Chapters 5 and 6 may wish to skip to Section 7.3).

7.2.1 The architecture of JJTraveler

Figure 7.1 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

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	$Choice(All(AllBottomUp(v)), v)$

Table 7.1: JJTraveler's library (excerpt).

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 Table 7.1. They will be explained in more detail below. A larger excerpt can be found in Table 5.2, and a full overview of the library can be found in the online documentation of JJTraveler.

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 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 (see Chapter 6). 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 ad-

```
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 7.2: The $Sequence$ combinator.

dition 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).

7.2.2 A library of generic visitor combinators

Table 7.1 shows high-level descriptions for an excerpt of JJTraveler's library of generic visitor combinators. 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 7.2 and 7.3 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,

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

Figure 7.3: The Try combinator.

```
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 7.4: The $TopDownWhile$ combinator.

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.

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 7.4. 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.

7.3 Cobol Control Flow

The example we use to study the application of visitor combinators to the construction of program understanding tools deals with Cobol control flow. Cobol has some special control-flow features, making analysis and visualization an interesting and non-trivial task. The analysis we describe is taken from DocGen (see [DK99a]), an industrial documentation generator for a range of languages including Cobol, which has been applied to millions of lines of code.

Control-flow in Cobol takes place at two different levels. A Cobol system consists of a series of *programs*. These programs can invoke each other using *call* statements. A Cobol system typically consists of several hundreds of programs.

In this chapter, we focus on control-flow *within* a program, for which the *perform* statement is used. This perform statement is like a procedure call, except that no parameters can be passed (global variables have to be used for that). Typical programs are 1500 lines large, but is not uncommon to have individual programs of more than 25,000 lines of code, resulting in significant program comprehension challenges.

7.3.1 Cobol Procedures

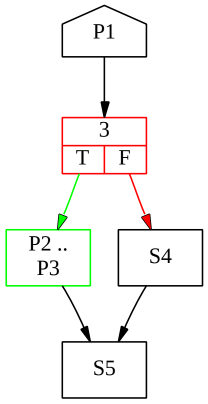
Cobol does not have explicit language constructs for procedure calls and declarations. Instead, it has labeled *sections* and *paragraphs*, which are the targets of *perform* and *goto* statements. Perform statements may invoke individual sections and paragraphs, or *ranges* of them. A section can group a number of paragraphs, but this is not necessary.

Figure 7.5(a) shows an example program in which sections, paragraphs, and ranges are performed. Paragraph `P1` acts as the main block, which reads an input value X. If it is “1”, the program invokes the range of paragraphs `P2` through `P3`. This range first prints HELLO, and then performs section `S5`, which prints WORLD. If the value read is not “1”, the main program invokes just the section `S4`. This section consists of two paragraphs, of which `P4` displays HI, and `P5` invokes `S5` to display WORLD.

This example illustrates an important program understanding challenge for Cobol systems. Viewed at an abstract level the program involves four *procedures*: `P1`, the range `P2..P3`, `S4`, and `S5`. Paragraphs `P3`, `P4` and `P5` are not intended as procedures. This abstract view needs to be reconstructed by analysis, because the entry and exit points of performed blocks of code is determined not by their declaration, but by the way they are invoked in other parts of the program. In gen-

```
PROCEDURE DIVISION.
 P1. ACCEPT X
     IF X = "1"
        PERFORM P2 THRU P3
     ELSE
        PERFORM S4.
     STOP RUN.
 P2. DISPLAY "HELLO".
 P3. PERFORM S5.
 S4 SECTION.
 P4. DISPLAY "HI".
 P5. PERFORM S5.
 S5 SECTION.
     DISPLAY "WORLD".
```

(a) Cobol source



(b) Corresponding call graph

Figure 7.5: Example Cobol source and graph

eral, this makes it hard to grasp the control-flow of a Cobol program, especially if it is of non-trivial size.

Typical, Cobol programmers try to deal with this issue by following a particular *coding standard*. Such a standard prescribes that, for example, only sections can be performed, or only ranges, or that “perform . . . thru . . .” can only be used for paragraphs with names that explicitly indicate that they are the start or end-label of a range. Such standards, however, are not enforced. Moreover, especially older systems may have been subjected to multiple standards, leaving a mixed style for performing procedures. Again, it takes analysis in order to find out which styles are actually being used at each point.

The formal semantics of “perform P_1 thru P_n” is that paragraphs are executed starting with P_1 until control reaches P_n. In principle, this makes determining which paragraphs are actually spanned by a range a run time problem, which cannot necessarily be solved statically. In the vast majority (99%) of Cobol programs, however, ranges coincide with syntactic sequences. In this chapter, we will assume that ranges are syntactically sequenced, and we refer to [FR99] for ways of dealing with dynamic ranges (where visitor combinators may well be applicable as well).

7.3.2 Analysis and visualization

To help maintenance programmers understand the control flow of individual Cobol programs, a tool is needed for analysis and visualization of a program’s perform dependencies. From such a call graph, one could instantly glean which perform style is predominant, which sections, paragraphs or ranges make up procedures, and how control is passed between these procedures.

When discussing these procedure-based call graphs with maintenance programmers, they indicated that they would also like to know *under what conditions* a procedure gets performed. This gave raise to the so-called *conditional call graph* (CCG), an example of which is shown in Figure 7.5(b). These graphs contain nodes for procedures and conditionals, which are connected by edges that represent call relations and syntactic nesting relations. CCGs are part of the DocGen redocumentation system, in which these graphs are hyperlinked to both the sources and to documentation at higher levels of abstraction (see [DK99a]).

Conditional call graphs are also a good starting point for computing detailed (per-procedure) metrics, as part of a systematic quality assurance (QA) effort. Example QA metrics include McCabe's cyclomatic complexity, fan-in, fan-out, deepest nesting level, coding style violations (goto's across section boundaries, paragraphs performing sections, or v.v.), dead-code analysis, and more.

7.4 ControlCruiser Architecture

We have implemented the analysis and visualization requirements just described using visitor combinators. The result is ControlCruiser, a Cobol analysis tool that provides insight into the intra-program call structure of Cobol programs. The tool employs several visitable source models, and performs various visitor-based traversals over them. This section discusses the ControlCruiser architecture; the next covers in detail how visitor combinators have been used in its implementation.

7.4.1 Initial Representation

The starting point for ControlCruiser is a simple language containing just the statements representing Cobol sections, paragraphs, perform statements, and conditional or looping constructs. An example of this *Conditional Perform Format* (CPF) is shown in Figure 7.6(a).

We obtain CPF from Cobol sources using a Perl script written according to the principles discussed in [DK98]. This script takes care of handling the tricky details of the Cobol syntax, such as scope termination of if-constructs.

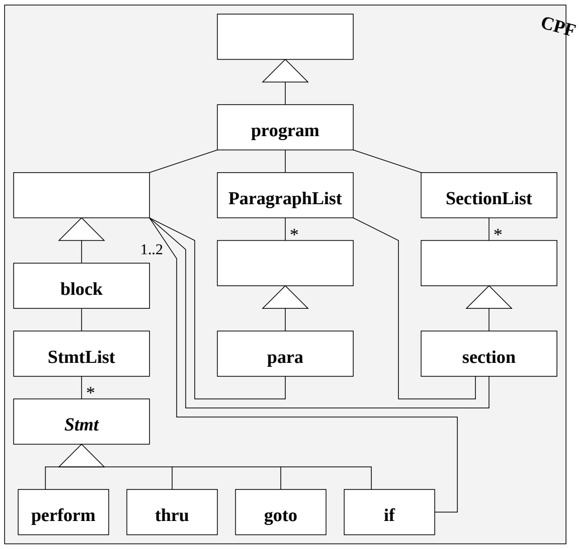
The result is an easy to parse CPF file. We have written a grammar for the CPF format, and used JJForester to derive a class hierarchy for representing the corresponding trees. All nodes in such trees are of one of the types shown in Figure 7.6(b). Since these all realize the *Visitable* interface, we can implement all subsequent steps with visitor combinators.

7.4.2 Graph Representation

To analyze Cobol's control flow in an easy way, we have to create a graph out of the tree representation corresponding to Cobol statements. For this, we use an

```
PARA 2 P1
  IF 3
    THRU 4 P2 P3
  ELSE 5
    PERFORM 6 S4
  END-IF 7
END-PARA 9 P1
PARA 9 P2
END-PARA 10 P2
PARA 10 P3
  PERFORM 10 S5
END-PARA 11 P3
SECTION 11 S4
  PARA 12 P4
  END-PARA 13 P4
  PARA 13 P5
    PERFORM 13 S5
  END-PARA 14 P5
END-SECTION 14 S4
SECTION 14 S5
END-SECTION 15 S5
```

(a) CPF for Fig 7.5



(b) The generated CPF class hierarchy

Figure 7.6: Conditional Perform Format (CPF)

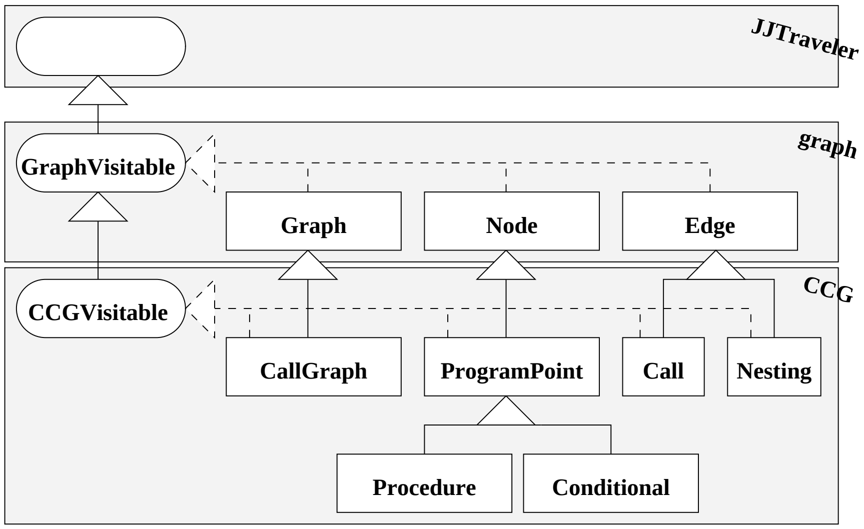


Figure 7.7: Class hierarchy for graph representations.

additional visitable source model which consists of two layers (see Figure 7.7).

The first layer is a generic graph model, with explicit classes for nodes, edges, and the overall graph providing entry points into the graph. Each of these classes implements a *GraphVisitable* interface, which is an extension of generic visitables. The classes are implemented such that the children of a node are defined as its outgoing edges, the children of an edge as its target node, and the children of a graph as the collection of all nodes, thus making it possible to traverse a graph using visitor combinators. A forwarding visitor combinator taking a generic visitor as argument is provided as required (not shown).

The second layer is a specialization of the generic graph model to the level of control flow, called *Conditional Control Graphs* (CCGs). This representation contains classes for procedures, conditional statements, and different types of edges. Program points correspond to places in the original CPF tree, and have a pointer back to their originating construct. Each class implements the *CCGVisitable* interface. The forwarding combinator of CCG (not shown) contains three levels of forwarding. First, visit methods of classes low in the hierarchy (such as Procedure and Conditional) invoke a visit method higher up in the hierarchy (to ProgramPoint). Second, visit methods for top-level CCG classes forward to visit methods in a visitor at the generic graph level. Third, graph-specific visitors forward to generic visitors by default. Observe that thanks to this two-layer design, visitors designed for graphs can be reused to build visitors for CCGs. This will be demonstrated in Section 7.5.2.

7.4.3 Graph Construction

Constructing the CCG graph from the initial CPF tree representation is done using various visitors operating on CPF trees. In order to identify those paragraphs, sections and ranges that act as procedures, a visitor *PerformedLabels* is used to collect all performed labels and ranges. A second visitor *ConstructProcedures* then uses these to find the corresponding paragraphs or sections and to add procedure nodes to the graph. For ranges, the corresponding *list* of paragraphs or sections is collected.

After the procedure nodes are created, the *RefineProcedure* visitor is applied, in order to extend the graph with the conditionals and outgoing call edges of these procedures.

7.4.4 Graph Analysis

Once the CCG graph is constructed, it can be analyzed. For this, we use a number of visitors that operate on CCG graphs.

To visualize a CCG graph, we traverse it with a visitor that emits input for the graph-drawing back-end `dot`. This visitor is layered, as is the CCG class hierarchy on which it operates.

SuccessCounter	v	Add one if v succeeds
CpfIfRecognizer		Succeed on CPF conditions
CcgIfRecognizer		Succeed on CCG conditions
...		Other recognizers
McCabeIndex	i	SuccessCounter(i), i an IfRecognizer
FanOut	p	SuccessCounter(p), p PerformRecogn.
GotoCounter	g	SuccessCounter(g), g GotoRecognizer
MaxNesting	v	Maximum nesting level of v-Recognizer
MaxNestedIf	i	MaxNesting(i), i an IfRecognizer

Figure 7.8: Selected Metrics Visitors

To compute metrics per procedure we have devised a number of collaborating visitors, shown in Figure 7.8. Most of these metrics are based on a *Success-Counter(v)*, which, when visited, applies its argument *v* and increments a counter if this application was successful. An example application is the *McCabeIndex* combinator, which takes a visitor recognizing if-statements, and then counts the number of successes. Observe that these metrics combinators are parameterized by recognizers: hence they can be applied to both the CPF and the CCG source models.

In a similar way we construct visitors for recognizing coding standards. For example, a visitor *MixedStyle* operates on the CCG format, and recognizes all call edges from *section* to *paragraph* or vice versa. Such edges indicate a mixed style, and usually are forbidden by coding standards.

7.5 ControlCruiser Implementation

In this section we discuss some of ControlCruiser's visitors in full detail. Due to space limitations, we limit ourselves to the visitors dealing with graph construction and visualization.

Collect performed labels Recall that perform statements come in two flavors: with and without *thru* clause. Consequently, we need to collect both individual labels, and pairs of labels. For this purpose we use a visitor combinator `PerformedLabels` with two collections in its state (see Figure 7.9). Note that there are no dependencies between the code in this visitor pertaining to pairs of labels and the code pertaining to individual labels. If desired, we could refactor this visitor into two even smaller separate ones, and re-join them with `Sequence` (visitor extraction).

To actually collect the labels from the input program `p`, we need to create the visitor, pass it to the generic `TopDown` combinator, and visit the tree with it:

```
public class PerformedLabels extends cpf.Fwd {
 Set performedLabels = ...;
 Set performedRanges = ...;
 public PerformedLabels() {
  super( new Identity());
 }
 public void visit_perform(perform p) {
  performedLabels.add(p.getcallee());
 }
 public void visit_thru(thru x) {
  performedRanges.add(
   new Pair(x.getstartlabel(), x.getendlabel()));
}}
```

Figure 7.9: Collect performed labels.

```
public class CreateProcedures extends cpf.Fwd {
 CallGraph callGraph;
 Set performedLabels;
 public CreateProcedures(CallGraph g, Set labs){
  super(new Identity());
  ...
 }
 public void visit_section(section s) {
  addProc(s.getlabel(), s);
 }
 public void visit_para(para p) {
  addProc(p.getlabel(), p)
 }
 void addProc(String name, Visitable v) {
  if (performedLabels.contains(name)) {
   Procedure p = new Procedure(name,v);
   callGraph.addProcedure(p);
}}}
```

Figure 7.10: Create procedures for individual labels.

```
PerformedLabels pl = new PerformedLabels();
( new TopDown(pl) ).visit(p);
```

After the traversal has completed, we can obtain the performed labels and ranges via the instance variables of `pl`.

Paragraphs and Sections Every performed label corresponds to either a section or a paragraph. In order to create a procedure node with the proper link back to the CPF tree representing the procedure body, we use a visitor that triggers at individual sections and paragraphs (see Figure 7.10). It only actually creates a procedure node if the given label is one of the performed labels, which it receives at construction time. The created procedure nodes are added to a call graph, which is also provided at construction time. To ensure we will be able to retrieve the

```
public class SpannedASTs extends cpf.Fwd {
 VisitableList spannedASTs = new VisitableList();
 String startLabel;
 String endLabel;
 boolean withinRange = false;
 public SpannedASTs(String start, String end) {
  super(new Identity());
  ...
 }
 public void visit_para(para p) {
  addIfWithinRange(p.getlabel(), p);
 }
 public void visit_section(section s) {
  addIfWithinRange(s.getlabel(), s);
 }
 void addIfWithinRange(String label,
                       Visitable x) {
  if (label.equals(startLabel)) {
    withinRange = true; }
  if (withinRange) {
    spannedASTs.add(x); }
  if (label.equals(endLabel)) {
    withinRange = false;
}}}
```

Figure 7.11: Collect section and paragraph nodes spanned by a given pair of labels.

added nodes at a later stage, we assume they become direct *children* of the graph.

Again, this visitor can be passed to the TopDown combinator, in order to traverse the tree and collect the procedures. Below, however, we will see how we can make better use of combinators in order to avoid visiting too many nodes.

Ranges To construct procedure nodes for a pair of (start and end) labels, we collect those section or paragraph nodes that lie between those labels. For this purpose we have developed an auxiliary visitor (see Figure 7.11) which takes the start and end labels, and is triggered at each section or paragraph. If the start or end label is encountered, a boolean flag is switched, and paragraphs or sections visited are added to the list.

Given this auxiliary visitor, a visitor can be developed that constructs procedure nodes for pairs of labels (see Figure 7.12). This visitor triggers at ParagraphList and SectionList nodes. This is appropriate, because the sections and paragraphs spanned by a pair of labels must always occur in the same list. When such a list is encountered, the method `addSpannedASTs` is invoked to perform an iteration over the collection of label pairs. At each iteration, the `All` combinator is used to fire the auxiliary visitor `SpannedASTs` sequentially at all members of the current paragraph or section list. If this yields a non-empty result, a new procedure node is created and added to the graph.

```
public class CreateRanges extends cpf.Fwd {
  CallGraph callGraph;
  Set todoRanges;
  public CreateRanges(CallGraph g, Set todo) {
    super(new Identity());
    ...
  }
  public void visit_ParaList(ParaList pl) {
   addSpannedASTs(pl);
  }
  public void visit_SectionList(SectionList sl) {
    addSpannedASTs(sl);
  }
  void addSpannedASTs(Visitable list) {
    Iterator pairs = todoRanges.iterator();
    while (pairs.hasNext()) {
      Pair pair = (Pair) pairs.next();
      VisitableList asts = getASTs(pair, list);
      if (! asts.isEmpty()) {
        addProc(pair.start, pair.end, asts);
  } } }
  VisitableList getASTs(Pair p, Visitable list) {
    SpannedASTs sa=new SpannedASTs(p.start, p.end);
    (new GuaranteeSuccess(new All(sa))).visit(list);
    return sa.spannedASTs;
  }
  void addProc(Pair p, VisitableList ast) {
  ...
} }
```

Figure 7.12: Create procedure for ranges

Top Down *While* Finally, we can apply the developed visitors to the input program. This could be done with a simple top-down traversal. However, any nodes at the block level and lower would be visited superfluously, because our visitors have effect only on sections, paragraphs, and lists of these. To gain efficiency, we will use the `TopDownWhile` combinator instead. To detect blocks, we first define the following visitor (using an anonymous class):

```
Visitor isBlock
  = new Fwd(new Fail())
         { public void visit_block(block x) {} };
```

This visitor fails for all nodes, except blocks. We compose it with our procedure creation visitors to do a partial traversal:

```
graph = new CallGraph();
cp    = new CreateProcedures(graph,labels);
cr    = new CreateRanges(graph,ranges);
(new TopDownWhile(
       new IfThenElse(isBlock,
                      new Fail(),
                          new Sequence(cp,cr))
        ) ).visit(p);
```

Thus, at each node the `IfThenElse` combinator is used to determine whether a block is reached and the traversal should stop, or the visitors for procedure creation should be applied. Note that these two separate visitors are combined into one with the `Sequence` combinator. After this traversal, the graph `g` contains a node for every procedure reconstructed from the CPF tree. Each such procedure node contains a reference to the CPF subtrees that gave rise to it.

Construct program entry point We will not show the visitors for constructing the program entry point. They are similar to the creation of performed procedure nodes. An auxiliary visitor collects ASTs, starting from the top of the program, and stopping at the first `STOP RUN` statement or the first performed label. This implements the heuristic that performed sections and paragraphs are never part of the main program.

7.5.1 CCG Refinement

Now we have created the CCG's procedure nodes, we need to refine them by creating nodes that represent the conditions that occur in their bodies, and by adding nesting and call relations between the nodes. For these tasks, we have developed the `RefineProcedure` visitor (see Figure 7.13). For a given procedure node in the CCG, this visitor is used to create nodes and edges for the conditionals and performs contained in its AST.

For a perform or a perform-thru statement, it adds a call edge from the `caller` to the procedure node that corresponds to its label (pair).

For if statements, it first creates a new conditional node and adds a nesting edge from the `callee` to this new conditional node. It then *restarts* itself with two new starting points: one for the then branch, and another for the else branch. The restart invokes the `TopDownUntil` combinator to traverse these branches. Such restarts are a general mechanism that can be used when stack-like behavior is needed, for example when dealing with nested constructs such as if statements.

We need to traverse the initial CCG to actually apply the `RefineProcedure` visitor at each procedure node. To prevent visiting nodes more than once and running in circles, we use the visitor `Visited` from JJTraveler's library (See Figure 7.14). This generic combinator keeps track of nodes already visited in its state. Now, to traverse the graph, we do a top-down traversal where each node that has not been visited yet is refined:

```
Visitor refine = new ccg.Fwd(new Identity()){
  public void visitProcedure(Procedure p) {
    RefineProcedure.start(graph, p);
  } };
```

```
public class RefineProcedure extends cpf.Fwd {
 CallGraph graph;
 ProgramPoint caller;
 public RefineProcedure(CallGraph g,
                        ProgramPoint c) {
  super(new Fail());
  ...
 }
 public void visit_perform(perform perform) {
  String label = perform.getcallee();
  Procedure callee = graph.getProcedure(label);
  caller.addCallEdgeTo(callee);
 }
 public void visit_thru(thru x) {
  String s = x.getstartlabel();
  String e = x.getendlabel();
  Procedure callee = graph.getProcedure(s,e);
  caller.addCallEdgeTo(callee );
 }
 public void visit_if$(if$ x) {
  Conditional cond = graph.addConditional(x);
  caller.addNestingEdgeTo(cond);
  start(graph, cond.getThenPart());
  start(graph, cond.getElsePart());
 }
 public static void start(CallGraph graph,
                          ProgramPoint caller) {
  Visitable ast = caller.getAst();
  RefineProcedure rp
    = new RefineProcedure(graph, caller);
  (new GuaranteeSuccess(
           new TopDownUntil(rp))) . visit(ast);
}}}
```

Figure 7.13: Refine the CCG for a given procedure.

```
(new TopDownWhile(
  new IfThenElse(new Visited(),
                 new Fail(),
                 refine)
  ) ).visit( graph );
```

Note that we use an anonymous extension of the `Identity` visitor to invoke the `start()` method of the visitor that does the actual refinement.

7.5.2 CCG visualization

The layered class hierarchy for graph representation allows us to implement a layered visualization visitor as well.

```
public class Visited implements Visitor {
  Set visited = new HashSet();
  public void  visit(Visitable x)
   throws VisitFailure {
    if (!visited.contains(x)) {
      visited.add(x);
      throw new VisitFailure();
} } }
```

Figure 7.14: The *Visited* combinator.

```
public class GraphToDot extends graph.Fwd {
 Set dotStatements = new TreeSet();
 public GraphToDot() {
  super(new Identity());
 }
 public void visitNode(GraphNode n) {
  add(n+";")
 }
 public void visitEdge(DirectedEdge e) {
  add(e.inNode()+"->"+e.outNode()+";");
 }
 void add (String dotStatement) { ...  }
 public void printDotFile(String fname) {...}
}
```

Figure 7.15: Generic graph visualization.

Visualizing generic graphs The visitor `GraphToDot` implements the construction of a representation in the `dot` input format for a given generic graph (see Figure 7.15). This visitor simply collects a set of `dot` statements, where an appropriate statement is added for each node and edge. After application of this visitor to each node and edge in a graph, the `printDotFile` method can be used to print the collected statements to a file.

Visualizing CCGs For our CCGs, the generic graph visualization does not suffice, because we want to generate different visual clues, for instance for call edges. For this purpose, we implemented `CCGToDot` (see Figure 7.16). Note that this visitor forwards to a generic `GraphToDot` visitor for all CCG elements but call edges. For these, the redefined visit method generates an adapted dot statement.

The visualization visitors are applied to the CCG in the exact same fashion as the `refine` visitor above. This calls for a refactoring of this traversal strategy into a reusable `GraphTopDown` combinator (extract strategy). We have added this combinator to JJTraveler's library.

```
public class CCGToDot extends ccg.Fwd {
 GraphToDot printer;
 public CCGToDot() {
  super(new GraphToDot());
  printer = (GraphToDot) fwd;
 }
 public void visitCall(Call c) {
  add(e.inNode()+"->"+e.outNode()
      +"[style=bold,color=blue];")
 }
 void add(String dotStatement) {
  printer.add(dotStatement);
 }
 public void printDotFile(String fname) {
  printer.printDotFile(fname);
}}
```

Figure 7.16: CCG visualization.

7.6 Evaluation

During the development of ControlCruiser we have learned many practical lessons about the use of visitor combinators for constructing program understanding tools. In this section we summarize some development techniques we have adopted and evaluate the benefits and risks of visitor combinator programming.

7.6.1 Development techniques

Separation of concerns Visitor combinators allow one to implement conceptually separable concerns in different modules, whilst otherwise they would be entangled in a single code fragment. As a result, these concerns can be understood, developed, tested, and maintained separately. Examples of (categories of) concerns we encountered include traversal, control, state, and testing (see below). Throughout all these concerns, we found it natural and beneficial to separate application-specifics from generics.

Testing and benchmarking We developed ControlCruiser following the extreme programming maxim of *test-first* design, which involves writing *unit tests* for every piece of code that can potentially fail. As a result, we wanted to test not only the compound visitors that are invoked by the application, but also each individual visitor combinator from which such compound visitors are composed.

To this end, we developed a testing combinator `LogVisitor`, which logs every invocation of its argument visitor into a special `Logger`. In combination with the standard unit testing utility JUnit, this testing combinator can be used to write detailed tests for hierarchy-specific visitors. To test the generic visitors of

JJTraveler itself, we used a mock instantiation of JJTraveler's framework (with a single visitable class).

For detailed benchmarking, we needed to collect timing results, again not just on compound visitors, but also on individual visitor combinators. To this end, we created a specialization `TimeLogVisitor` of our testing combinator that measures and aggregates the activity bursts of its argument visitor. This enables us to separately measure the time consumed by different concerns, such as traversal and node action.

Failure containment When using visitor combinators that potentially fail, one needs to declare the `VisitFailure` exception in a `throws` clause. In many cases, the programmer knows from the context that such failure can actually never occur. Examples are the expressions $Try(Fail)$ and $TopDownWhile(Fail)$. To relieve the programmer from the burden of writing catch-throws contexts to contain such 'impossible' failures, we developed the combinator `Guarantee-Success`. Judicious placement of this combinator reduces code cluttering and makes code more self-documenting.

Class organization We have used several kinds of inner classes to improve code organization. For tiny visitors (no more than a few lines) we have used *anonymous* classes. For small visitors (no more than a few methods) that operate within the context of another visitor (i.e. using its state), we used *member* classes. This removes the need for additional instance variables and constructor method arguments.

7.6.2 Benefits and risks

Benefits Visitor combinators enable separation of concerns. This helps understanding, development, testing, and reuse. Combinators enable reuse in several dimensions. Within an application, a single concern, such as a particular traversal strategy or applicability condition, needs to be implemented only once in a reusable combinator. Across applications, visitors can be reused that capture generic behavior. Examples are the fully generic combinators of the JJTraveler library, but also the `DotPrinter` combinator that can be refined by any application that uses or even specializes the `graph` package on which this combinator operates.

A related benefit is robustness against class-hierarchy changes. Using visitor combinators, each concern can be implemented with explicit reference only to classes that are relevant to it. As a result, changes in other classes will not unduly affect the implementation of the concern.

In relation to other approaches to separation of concerns and object traversal, visitor combinators are extremely lightweight. Optionally, the JJForester tool can be used to instantiate JJTraveler's framework. However, visitor combinators do not essentially rely on tools. The required implementation of the (very thin) `Visitable` interface and the `Fwd` combinator is straightforward, and can easily be done by hand.

Risks Visitor combinators pose two risks with respect to performance. Firstly, the development of many little visitors may lead to many (relatively expensive) object creations. One should take care to keep these within reasonable limit. For instance, stateless combinators need only be created once. Stateful visitors can often be re-initialized to run again, instead of continually creating new ones.

Another performance penalty may come from heavy reliance on exceptions for steering visitor control. One should take care to choose the interpretation of `VisitFailure` such that failure is less common than success. E.g. one can use `TopDownWhile` with `Identity` as default, instead of `TopDownUntil` with `Fail` as default.

These performance risks can be combatted by profiling (maybe using `Time-LogVisitor`) and refactoring. Refactoring rules for combinators can often be described with simple equations. However, when we applied ControlCruiser to our code bases, including a 3,000,000 loc system, we did not experience performance problems. (in fact, the majority of the time was spent on parsing the CPF format, not on running the visitors on them).

7.7 Concluding Remarks

Related work We refer to Chapter 5 for a full account of related work in the areas of design patterns and object navigation approaches: of particular interest are the *extended* [GH98] and *staggered* [Vli99] visitor patterns, and adaptive programming [LPS97] for expressing "roadmaps" through object structures. The origins of visitor combinators can furthermore be traced back to *strategic term rewriting*, in particular [VBT99].

Traversals in the context of reverse engineering tools are discussed by [BSV00], who provide a top-down analysis or transformation traversal. Their traversals have been generalized in the context of ASF+SDF in [BKV02]. Similar traversals are present in the Refine toolset [MNB$^+$94], which contains a pre-order and post-order traversal. In both cases, only a few traversal strategies are provided, and little support is available for composing complex traversals from basic building blocks or controlling the visiting behavior.

In the field of program understanding and reengineering tools *exchange formats* have attracted considerable attention since 1998 [WOL$^+$98]. Visitor combinators provide an interesting perspective on such formats. Instead of focusing on the underlying structure, visitor combinators make assumptions on what they can observe in a structure. By minimizing these assumptions, for example by trying to use the generic *Visitable* interface, the reusability of these combinators is maximized.

One of the outcomes of the exchange format research is the Graph Exchange Language GXL [HWS00]. Visitor combinators are likely to be a suitable mechanism for processing GXL representations. This requires generating directed graph structures that implement the *Visitable* interface from GXL schema's, similar to the way JJForester generates visitable trees from context free grammars and to the way our graph package implements the visitable interface.

Contributions We have demonstrated that visitor combinators provide a powerful programming technique for processing source models. We have given concrete examples of instantiating the visitor combinator framework provided by JJTraveler, and of developing complex program understanding visitors by specialization and combination of JJTraveler's combinator library. We have applied the developed visitors to a large code base to establish feasibility and scalability of the approach. Finally, we have summarized the development techniques surrounding visitor combinator programming and we have made an assessment of the risks and benefits involved.

Chapter 8

Conclusions

In the introduction to this thesis, these research questions have been posed:

1. Can traversal over source code representations be both generic and strongly typed?

2. Can typed generic traversal be supported within the context of general-purpose, mainstream programming languages?

3. Can typed generic traversal support be integrated with support for other common language tool development tasks?

We will now assess how the material of the various chapters have addressed these questions.

8.1 Typed generic traversal

In both the functional programming paradigm and the object-oriented programming paradigm, we have developed a combinatorial approach to typeful traversal construction, where generic traversal combinators are first-class citizens that allow the amalgamation of generic and type-specific behavior. Our languages of choice were the non-strict, strongly typed functional language Haskell, and the class-based object-oriented language Java.

For Haskell, we presented a 'conservative' approach that stays close to the underlying paradigm but is limited in expressivity, and a more 'radical' approach that makes a paradigm shift to achieve more power. The 'conventional' functional approach, presented in Chapter 3 stays close to current functional programming practice, since it builds on the established notion of generalized folds (bananas). For these, we introduced the notion of *fold algebra update*, we invented a small set of basic fold algebra combinators, and we demonstrated how these ingredients enable a combinatorial style of traversal construction. Updatable folds improve over non-updatable folds, because they realize conciseness and robustness via the reuse of generic default algebras. The updatable fold algebras also realize a limited amount of composability and traversal control. A clear disadvantage of the updatable fold approach is that extension of the set of combinators requires modification of the underlying (generative) tool support; users can not program new basic combinators themselves. Another disadvantage is that the reliance on folds implies that the traversal scheme can not be controlled by the programmer, but is always the primitive recursion scheme associated to the source code representation at hand. This makes this approach suitable for a limited category of traversal scenarios only.

The ‘radical’ functional approach, presented in Chapter 4, is further removed from current functional programming practices and can be considered to constitute a *paradigm shift*. It takes its inspiration from the untyped language Stratego for term rewriting with strategies. The most important feature adopted from Stratego is the decoupling of recursion and one-step traversal. To meet the challenge of providing definitions of the one-step traversal combinators as well as for the combinators that blend generic and type-specific behavior, we needed to go beyond ordinary parametric and ad-hoc polymorphism, as available in Haskell. This was accomplished, again via generative tool support. In contrast to the updatable fold approach, users can construct new basic combinators without adapting the generator, and full traversal control can be exerted. Conciseness and robustness are realized by both the fold approach and the strategy approach.

In the object-oriented paradigm, a ‘conservative’ approach to generic traversal was already available in the form of the *Visitor* pattern. Unfortunately, this approach shares the disadvantages of the functional approach with updatable folds. Visitors resists composition, and they allow almost no traversal control. Further disadvantages are the lack of robustness and conciseness of default visitors. Nonetheless, for a limited category of traversal scenarios, the plain *Visitor* pattern is sufficient, and for this reason it is supported by our visitor combinator JJForester.

In Chapter 5, we develop a ‘radical’ approach to traversal construction in Java, again inspired by Stratego. We introduced the notion of a *visitor combinator* as the object-oriented counterpart to a strategy combinator. All one-step traversal combinators can be expressed concisely as visitor combinators. The blending of type-specific and generic behavior was realized in a special *forwarding* visitor combinator *Fwd* by a combination of run-time type inspection (RTTI) and double dispatch as simulated by the plain *Visitor* pattern. Our visitor generator JJForester provides appropriate tool support to relieve the programmer of supplying this tedious and usually lengthy combinator. In contrast to plain visitors, our visitor combinators realize conciseness, robustness, composability, and traversal control.

8.2 Mainstream programming

Rather than offering a dedicated niche-language with generic traversal support, we have chosen to add this support to existing (relatively) mainstream languages, in particular to Java and Haskell. The typing systems of these languages, and the accompanying expressiveness are insufficient to cope with generic typeful traversal. Haskell supports ad-hoc polymorphism and parametric polymorphism. These forms of polymorphism are too limited to support our envisioned combinator style of traversal construction that blends genericity and specificity. Java features subtype polymorphism and dynamic dispatch. These are likewise insufficient for our purposes.

For both languages we have found a way to add to the expressiveness via a simple generative approach that involves the representation type only. The operations on the representations need not be touched by our generators. In the case of Haskell, this means that class instances are generated from datatype definitions. Programmers can write functions on these datatypes in plain Haskell that does not need to be pre-compiled or pre-processed. In the case of Java, both the visitable class-hierarchy and a single forwarding visitor combinator are generated from an SDF definition. The visitors operating on the class-hierarchy are written in plain Java without a need for pre-compilation.

This set-up has many advantages. Firstly, the amount of generated code is related to the size of the representation type, not to the size of the operations that are programmed on them. Secondly, programmers need not program in an extension of the original language, but can stick with what they know. The disadvantages usually associated with preprocessing – poor error messages, poor static checks – are circumvented. Also, all tooling, libraries, manuals, and programming expertise for the programming language at hand remains valid when using the generic traversal support. For instance, we have been able to generate technical documentation with the *javadoc* tool both for the generic visitor combinators of JJTraveler, and for the specific visitor combinators of ControlCruiser. In case of Haskell, we have for instance been able to use libraries for parsing, pretty-printing, collections, and XML processing directly in all our traversal code.

8.3 Integrated language tool development

Generic traversal is an important and challenging aspect of language tool development support, but to be truly useful, it must be integrated with support for other aspects. In Chapter 2 we have identified those aspects and established requirements on the components that are to support them. We have also developed a comprehensive architecture in which all aspects are integrated. The binding factor in this architecture are the grammars of the languages that must be processed, following the *Grammars as Contracts* maxim. We have also briefly discussed a number of instantiations of the architecture for programming languages such Java, Haskell, C, and Stratego. These instantiations of the architecture feature SDF as grammar formalism, and the ATerms as common exchange format. Using this format, components written in these languages can be connected to each other and to the generalized LR parser `sglr` and the generic pretty printing toolset.

For Haskell and Java, we have taken a closer look at the traversal aspect in Chapters 3, 4, and 5. For Java, we have developed parsing support in Chapter 6. In particular, we have provided a connection between SDF on one hand, and Java applications on the other hand. In line with the *Grammars as Contracts* idea, this connection is generated from an SDF grammar. Connectivity to SDF for Haskell was developed and described by us in [KLV00], and is available in the *Sdf2Haskell* package (see below).

Finally, Chapter 7 shows the Java instantiation at work of the comprehensive architecture for language tool development. The particular application developed in that chapter involves parsing, as well as traversal of tree-shaped and graph-shaped source code representations.

8.4 Available software

Throughout the chapters of this thesis, we have reported on the development of tools and libraries that support the presented techniques. Here we will give a short summary of software developed during the course of our investigations:

Grammar Base The grammar base is a collection of SDF grammars that can be used as contracts according to the meta-tool architecture described in Chapter 2. All included grammars are subjected to the versioning regime explained in Section 2.5.

Many different people have contributed to the grammar base. Its main maintainers are Merijn de Jonge, Eelco Visser, and Joost Visser. The Grammar Base is available online at:

`http:/www.program-transformation.org/gb/`

At this site, grammars can be browsed and downloaded. The Grammar Base is also available as one of the packages of XT (see below).

XT The Transformation Tools bundle XT provides a wide variety of language tool components. The meta-tools described in Chapter 2 are among XT's components. XT reuses various components from the ASF+SDF Meta-Environment [BDH$^+$01]. An overall description of the XT system can be found in [JVV01].

The maintainers of XT are Merijn de Jonge, Eelco Visser, and Joost Visser. XT is available from:

```
http://www.program-transformation.org/xt/
```

XT has been applied in a wide variety of applications, among which reverse engineering of SDL specifications [JM01], and Cobol transformation [Wes02].

Tabaluga Tabaluga supports programming with updatable fold algebras in Haskell. It consists of a code generator that consumes SDF grammars and generates Haskell datatypes, corresponding fold functions and fold algebra types, basic fold algebras, and fold algebra combinators. These were described in Chapter 3. Also, code for connectivity with the `sglr` parser is generated. Together, the generated code forms a framework for analysis and transformation of the input language, in accordance to the meta-tool architecture presented in Chapter 2. Layout and comment preservation is supported.

The main implementor of Tabaluga is Jan Kort. Tabaluga is available from:

```
http://www.science.uva.nl/~kort/tabaluga/
```

Apart from the main system, an additional package is available with pre-generated transformation and analysis frameworks for several languages, including Pico, Sdf, and Cobol.

Strafunski Strafunski supports programming with functional strategies in Haskell. The core of Strafunski are the library of strategy combinators *StrategyLib* and a supporting generator *DrIFT-Strafunski*. These were described in Chapter 4. DrIFT-Strafunski is an extended version of DrIFT (formerly called Derive [Win97]) which can be used (pending native support for strategies) to generate the instances of class $Term$ according to the model in Section 4.4.1 for any given algebraic data type. Actually, several alternative models with different (performance) characteristics are supported by the Strafunski distribution as well.

In addition to the library and the instance generator, some meta-tools have been developed to embed our strategy combinator support in a grammar-centered architecture as described in Chapter 2. These tools are the *Haskell ATerm Library*, to make the ATerm common exchange format available in Haskell, and the generator *Sdf2Haskell*, which generates Haskell datatypes from SDF definitions.

The main implementors of Strafunski are Ralf Lämmel and Joost Visser. Strafunski is available from its home page at:

```
http://www.cs.vu.nl/Strafunski/
```

The distribution contains a number of application examples that demonstrate the usability for processing languages of non-trivial size, such as Cobol, Java, Haskell, Sdf, and XML.

The Haskell ATerm Library is also separately distributed via:

`http://www.cwi.nl/projects/MetaEnv/haterm/`

JJTraveler JJTraveler is a combined visitor combinator framework and library for Java. It contains all the generic visitor combinators presented in Chapters 5, and more.

The main implementor of JJTraveler is Joost Visser. JJTraveler is available together with online documentation from the JJForester home page (see below).

JJForester JJForester is a combined parser and visitor combinator that automates instantiation of the JJTraveler framework by generation of Java code from an SDF grammar. JJForester was discussed in Chapter 6.

The main implementors of JJForester are Tobias Kuipers and Joost Visser. JJForester is available from its home page at:

`http://www.jjforester.org/`.

The distribution contains several example applications, among which a Java metrics extractor, and the Toolbus communication graph generator that was described in Chapter 6.

The Software Improvement Group has used JJForester in various legacy system reverse engineering projects. Tackled languages include Java, SQL, and Accell.

ControlCruiser Control Cruiser is a reimplementation and elaboration of the conditional control graph extraction component of DocGen [DK99a], a commercial documentation generator for Cobol which is developed and deployed by the Software Improvement Group. ControlCruiser was built using JJForester, JJTraveler, and some SDF tools developed for the ASF+SDF Meta-Environment. This was described in Chapter 7.

Control Cruiser was developed by Arie van Deursen and Joost Visser. ControlCruiser can be downloaded from the JJForester home page (see above).

Apart from the software developed in the course of our investigations, we also list a selection of tools that we made use of:

The ATerm Library The ATerm Library is an important component of many of the tools we used and developed. Apart from APIs for construction and inspection of ATerms, the ATerm distribution contains numerous command-line tools for ATerm processing.

The ATerm Library is available from:

`http://www.cwi.nl/projects/MetaEnv/haterm/`

The ATerm Library supports C and Java. As reported above, we additionally implemented a version of the ATerm Library that supports Haskell.

SDF parse table generator and generalized LR parser The parse table generator `pgen` and the scannerless generalized LR parser `sglr` that consumes these tables are the primary tools that support the syntax definition formalism SDF. They are available from:

```
http://www.cwi.nl/projects/MetaEnv/pgen/
http://www.cwi.nl/projects/MetaEnv/sglr/
```

The ASF+SDF Meta-Environment The ASF+SDF Meta-Environment is an interactive development environment that integrates the tools that support SDF and the term rewriting language ASF. It is available from:

`http://www.cwi.nl/projects/MetaEnv/`

Though currently the ASF+SDF Meta-Environment is limited to ASF as programming language, efforts are underway to generalize it to allow the use of different languages. The Haskell and Java related tools reported above might play a role in this development, since they accomplish the integration of SDF with these languages.

All these packages are also available through the online package base at:

`http://www.program-transformation.org/package-base/`

At this page, one or more of the above packages can be selected to generate a self-contained software distribution that bundles the selected packages and those they depend on.

8.5 Perspectives

The work presented in this thesis has laid the theoretical foundations for object-oriented and functional strategic programming, and it has demonstrated the feasibility of its practical use. Still, many open issues remain before typed strategic programming can become a main-stream technique and fully realize its potential for a general software development audience.

Fundamental issues How can the essential notions of strategic programming be formalized in a single, paradigm-independent theoretical framework? How can strategic programming be combined with other forms of generic programming, such as polytypic programming and generic attribute grammars?

What properties of generic traversals can be established that are useful for reasoning about strategic programs? How can one take advantage of recent developments in functional and object-oriented programming (rank-2 polymorphism, bounded polymorphism) to make generic function combinators and visitor combinators more expressive or more type-safe.

What is the class of traversal problems that is covered by the current strategic programming constructs? What additional *basic* combinators would be needed to enlarge this class? Can the technique of strategic programming be complemented or adapted to cater for specific application areas?

Tool and library development How can the tools that offer support for strategic programming (Strafunski, JJTraveler, JJForester) be made easy to use by the working strategic programmer. Can they be generalized to support a wider class of applications? Can benchmarks be developed to investigate the performance trade-offs when applying strategic programming support? Can optimizations be developed and implemented that exploit the theoretical properties of strategic programming constructs?

What additional useful defined combinators can be added to the combinator libraries? Can the libraries be made more accessible by providing (generated) documentation?

Can domain-specific frameworks, based on strategy combinators be developed, that allow rapid construction of applications in these domains? What ingredients are needed for strategy combinator frameworks for instance for XML transformation, Cobol legacy system renovation, or Java source code analysis?

Consolidation of design expertise How can the experience of strategic program construction be consolidated and communicated? An initial catalogue of design patterns for functional strategic programming has been drafted [LV02a]. Can this catalogue be expanded with new patterns? Can a design pattern catalogue likewise be developed for program construction with visitor combinators? Can more show-case application examples be developed that would guide strategic programmers in constructing their own applications?

These open issues indicate directions that future investigations concerning generic traversal of typed data structures could take.

Bibliography

[A$^+$02] I. Attali et al. Aspect and XML-oriented semantic framework generator: SmartTools. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[ABFP86] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39, May 1986.

[ACPP91] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[AG93] A. W. Appel and M. J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley, 1986.

[B$^+$96] M. G. J. van den Brand et al. Industrial applications of ASF+SDF. In *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[BB85] C. Bohm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39(2-3):135–153, August 1985.

[BDH$^+$01] M. van den Brand, A. van Deursen, J. Heering, H. de Jonge, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC 2001)*, volume 2027 of *LNCS*. Springer-Verlag, 2001.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In *Algebraic Specification*, chapter 1, pages 1–66. The ACM Press in coöperation with Addison-Wesley, 1989.

[BJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[BK94] J. A. Bergstra and P. Klint. The ToolBus: a component interconnection architecture. Technical Report P9408, University of Amsterdam, Programming Research Group, 1994. Available from `http://www.science.uva.nl/research/prog/reports/reports.html`.

[BK96] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996.

[BK98] J. A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[BKKR01] P. Borovansky, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.

[BKV02] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with type-safe traversal functions. In B. Gramlich and S. Lucas, editors, *Second International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[BM98] R. Bird and L. Meertens. Nested datatypes. In *4th International Conference on Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 1998.

[Bor98] P. Borovanský. *Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, 1998.

[Bou96] R. J. Boulton. SYN: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical report, Computer laboratory, University of Cambridge, 1996.

[BP99] R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[BPSM98] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.

[BSV97] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153. IEEE, 1997.

[BSV98] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117. IEEE, 1998.

[BSV00] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of Components for Software Renovation Factories from Context-free Grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.

[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.

[BV95] J. C. M. Baeten and C. Verhoef. Concrete process algebra. In *Handbook of Logic in Computer Science*, volume 4, pages 149–268. Clarendon Press, Oxford, 1995.

[BV96] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, 1996.

[BW90] J.C.M Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[c2] Portland pattern repository. `http://www.c2.com/cgi/wiki`.

[CC90] E.J. Chikofsky and J.H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CE99] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, 1999.

[CK99] H. Cirstea and C. Kirchner. Introduction to the rewriting calculus. Rapport de recherche 3818, INRIA, December 1999.

[CKL01] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In Furio Honsell, editor, *Foundations of Software Science and Computation Structures, ETAPS'2001*, Lecture Notes in Computer Science, pages 166–180, Genova, Italy, April 2001. Springer-Verlag.

[CL02] D. Clarke and A. Löh. Generic haskell, specifically. In *Proceedings of The Working Conference on Generic Programming*, Dagstuhl, Germany, 2002.

[Cle88] J. C. Cleaveland. Building application generators. *IEEE Software*, pages 25–33, July 1988.

[CS92] R. Cockett and D. Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS, 1992.

[CWM99] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. *ACM SIGPLAN Notices*, 34(1):301–312, January 1999.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[DK98] A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In *International Workshop on Program Comprehension*, pages 90–97. IEEE, 1998.

[DK99a] A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

[DK99b] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.

[DK02] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 2002.

[DKV99] A. van Deursen, P. Klint, and C. Verhoef. Research Issues in the Renovation of Legacy Systems. In J.P. Finance, editor, *Proc. of FASE'99*, volume 1577 of *LNCS*, pages 1–21. Springer-Verlag, 1999.

[DKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages – an annotated bibliography. *ACM SIGPLAN Notices*, 35(6), June 2000.

[DOM98] Document Object Model (DOM) Level 1 Specification Version 1.0, October 1998. W3C Recommendation.

[DRW95] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Conference record of POPL'95*, pages 118–129. ACM Press, 1995.

[DTS99] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Bern, August 1999.

[DV02a] A. van Deursen and E. Visser. The reengineering wiki. In *Proceedings 6th European Conference on Software Maintenance and Reengineering (CSMR).*, pages 217–220. IEEE Computer Society, 2002.

[DV02b] A. van Deursen and J. Visser. Building program understanding tools using visitor combinators. In *Proceedings of the Tenth International Workshop on Program Comprehension (IWPC 2002)*, pages 137–146. IEEE Computer Society, 2002.

[EHM$^+$99] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. B. Sørensen, and M. Tofte. AnnoDomini: From type theory to year 2000 conversion tool. In *Conference Record of POPL'99*, pages 1–14. ACM press, 1999. Invited paper.

[Fil99] A. Filinski. Representing layered monads. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 175–188, New York, N.Y., January 1999. ACM.

[Fok92] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[Fok94] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[FR99] J. Field and G. Ramalingam. Identifying procedural structure in cobol programs. In *Workshop on Program analysis for software tools and engineering; PASTE*, pages 1–10. ACM Press, 1999.

[FSS92] L. Fegaras, T. Sheard, and D. Stemple. Uniform Traversal Combinators: Definition, Use and Properties. In D. Kapur, editor, *Proc. 11th Intl. Conf. on Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Saratoga Springs, NY, USA, June 1992. Springer-Verlag.

[GB] The online grammar base. `http:/www.program-transformation.org/gb/`.

[GH98] E. M. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS USA 98 (Technology of Object-Oriented Languages and Systems)*. IEEE, 1998.

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

[Has99] *Haskell 98: A Non-strict, Purely Functional Language*, February 1999. `http://www.haskell.org/onlinereport/`.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Hin99] R. Hinze. A generic programming extension for Haskell. In E. Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. Technical report, Universiteit Utrecht, UU-CS-1999-28.

[Hin00] R. Hinze. A New Approach to Generic Functional Programming. In Thomas W. Reps, editor, *Conference record of POPL'00*, pages 119–132, January 2000.

[HM00] I. Herman and M.S. Marshall. GraphXML – An XML-based graph description format. In *Symposium on Graph Drawing (GD 2000)*, volume 1984 of *LNCS*, pages 52–62. Springer, 2000. A full grammar for GraphXML can be found at `http:/www.program-transformation.org/gb/`.

[HWS00] R. Holt, A. Winter, and A. Schürr. GXL: Toward a standard exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 162–171. IEEE Computer Society, 2000.

[Jay99] C.B. Jay. Programming in FISh. *International Journal on Software Tools for Technology Transfer*, 2:307–315, 1999.

[JB99] J. Jennings and E. Beuscher. Verischemelog: Verilog embedded in Scheme. In *Proceedings of the second USENIX Conference on Domain-Specific Languages*, pages 123–134. USENIX Association, October 3–5 1999.

[JJ97a] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Conference record of POPL'97*, pages 470–482. ACM Press, 1997.

[JJ97b] P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *POPL '97: 24th Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 15–17 January 1997.

[JM95] J. Jeuring and E. Meijer, editors. *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995.

[JM01] M. de Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. In *Proceedings of the International Conference on Software Maintenance (ICSM 2001)*, pages 240 – 249. IEEE Computer Society Press, november 2001.

[JO02] H.A. de Jong and P.A Olivier. Generation of abstract programming interfaces from syntax definitions. Technical Report SEN-R0212, St. Centrum voor Wiskunde en Informatica (CWI), August 2002. Submitted to Journal of Logic and Algebraic Programming.

[Joh75] S. C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[Jon95] M.P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In Jeuring and Meijer [JM95], pages 97–136.

[Jon97] M.P. Jones. First-class polymorphism with type inference. In *Conference record of POPL'97*, pages 483–496, Paris, France, 15–17 January 1997.

[Jon99] M.P. Jones. Type Classes and Functional Dependencies, 1999. http://www.cse.ogi.edu/˜mpj/.

[Jon00] M. de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

[Jon02a] M. de Jonge. Pretty-printing for software reengineering. submitted for publication, march 2002.

[Jon02b] M. de Jonge. Source tree composition. In *Proceedings: Seventh International Conference on Software Reuse*, LNCS. Springer-Verlag, 2002.

[JV00] M. de Jonge and J. Visser. Grammars as contracts. In *Proceedings of the Second International Conference on Generative and Component-based Software Engineering (GCSE 2000)*, volume 2177 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2000.

[JVV01] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In Mark van den Brand and Didier Parigot, editors, *Proceedings of Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[KKV95] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. MIT Press, 1995.

[KL$^+$97] G. Kiczales, J. Lamping, et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, number 1241 in LNCS. Springer Verlag, 1997.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

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

[Küh98] T. Kühne. The translator pattern — external functionality with homomorphic mappings. In Raimund Ege, Madhu Singh, and Bertrand Meyer, editors, *The 23^{rd} TOOLS conference USA '97*, pages 48–62. IEEE Computer Society, July 1998.

[KV01] T. Kuipers and J. Visser. Object-oriented tree traversal with JJ-Forester. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of the first Workshop on Language Descriptions, Tools and Applications (LDTA 2001), to appear also in Science of Computer Programming.

[Läm00] R. Lämmel. Reuse by Program Transformation. In Greg Michaelson and Phil Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000. Selected papers from the 1st Scottish Functional Programming Workshop.

[Läm02a] R. Lämmel. Towards Generic Refactoring. Technical Report cs.PL/0203001, arXiv, March1 2002.

[Läm02b] R. Lämmel. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming*, 2002. To appear.

[LPS97] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, July 1997.

[LR94] D. A. Ladd and J. C. Ramming. Two application languages in software production. In *USENIX Very High Level Languages Symposium Proceedings*, pages 169–178, October 1994.

[LV97] B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

[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*, pages 137–154. Springer, 2002.

[LVK00] R. Lämmel, J. Visser, and J. Kort. Dealing with large bananas. In Johan Jeuring, editor, *Proceedings of the second Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

[MBJ99] E Moggi, Bellè, and C.B. Jay. Monads, shapely functors and traversals. In M. Hoffman, Pavlović, and P. Rosolini, editors, *Proceedings of the Eighth Conference on Category Theory and Computer Science*

(CTCS'99), volume 24 of *Electronic Lecture Notes in Computer Science*, pages 265–286. Elsevier, 1999.

[Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[Mee96] L. Meertens. Calculate Polytypically! In H. Kuchen and S.D. Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *LNCS*, pages 1–16. Springer-Verlag, 1996.

[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.

[MH95] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Conf. Record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 324–333. ACM Press, New York, 1995.

[MJ95] E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In Jeuring and Meijer [JM95], pages 228–266.

[MLM01] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[MNB$^+$94] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Comm. of the ACM*, 37(5):58–70, 1994.

[Moo02] L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society Press, June 2002.

[OW99] J. Ovlinger and M. Wand. A language for specifying recursive traversals of object structures. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 70–81, 1999.

[Pau83] L.C. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, August 1983.

[PJ98] J. Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.

[PXL95] J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.

[Rei02] C. Reinke, editor. Haskell communities and activities report, second edition. `http://www.haskell.org/communities/`, May 2002.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[SAS99] S. Doaitse Swierstra, P. R. A. Alcocer, and J. Saraiva. Designing and implementing combinator languages. In S.D. Swierstra, P.R. Henriques, and J.N. Oliveira, editors, *Advanced Functional Programming, Third International School, AFP '98*, volume 1608 of *LNCS*, pages 150–206, Braga, Portugal, September 1999. Springer-Verlag.

[SF93] T. Sheard and L. Fegaras. A Fold for All Seasons. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, Denmark, June 1993. ACM Press. ISBN 0-89791-595-X.

[She91] T. Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, October 1991.

[Sne00] G. Snelting. Software reengineering based on concept lattices. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 3–12. IEEE Computer Society, 2000. Invited contribution.

[Tom85] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[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).

[Vis97] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[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.

[Vis00a] E. Visser. Language Independent Traversals for Program Transformation. In J. Jeuring, editor, *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, pages 86–104, July 2000.

[Vis00b] E. Visser. Language independent traversals for program transformation. In Johan Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

[Vis01a] E. Visser. *The Stratego Library (version 0.4.22)*. Institute of Information and Computing Sciences, Utrecht, The Netherlands, 2001. Most recent version at `http://www.stratego-language.org/`.

[Vis01b] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, 2001. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2001).

[Vli99] J. Vlissides. Visitor in frameworks. *C++ Report*, 11(10), November 1999.

[Wad89] P. Wadler. Theorems for Free! In *Proc. of FPCA'89, London*, pages 347–359. ACM Press, New York, September 1989.

[Wad92] P. Wadler. The essence of functional programming. In *Conference record of POPL'92*, pages 1–14. ACM Press, 1992.

[WAKS97] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–28, Berkeley, CA, October 15–17 1997. USENIX Association.

[Wei00] S. Weirich. Type-safe cast: (functional pearl). *ACM SIGPLAN Notices*, 35(9):58–67, September 2000.

[Wes02] H. Westra. Configurable transformations for high-quality automatic program improvement – CobolX: a case study. Master's thesis, Department of Computer Science, Utrecht University, January 2002. Appeared as technical report INF/SCR-02-01.

[Wil97] D. S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 472–480, Berlin - Heidelberg - New York, May 1997. Springer.

[Win97] N. Winstanley. Derive User Guide, version 1.0. Available at `http://www.dcs.gla.ac.uk/~nww/Derive/`, June 1997.

[WOL$^+$98] S. Woods, L. O'Brien, T. Lin, K. Gallagher, and A. Quilici. An architecture for interoperable program understanding tools. In *6th International Workshop on Program Comprehension (IWPC)*, pages 54–63. IEEE, 1998.

[WR99] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? *ACM SIGPLAN Notices*, 34(9):148–159, September 1999. Proceedings of the International Conference on Functional Programming (ICFP'99), Paris, France.

[XSL99] XSL Transformations (XSLT) Version 1.0, November 1999. W3C Recommendation.

Summary

Many areas of software engineering essentially involve analysis and transformation of source code representations. Generally, such representations are highly heterogenous data structures. Examples are parse trees, abstract syntax trees, dependency graphs, and call graphs. Preferably, the well-formedness of such data structures is guarded by strong static type systems.

Unfortunately, when using traditional approaches, typeful programming is at odds with conciseness, reusability, and robustness. Access to and traversal over subelements of typed representations involves dealing with many specific types in specific ways. As a consequence, type-safety comes at the cost of lengthy traversal code, which can not be reused in different parts of the representation or for differently typed representations, and which breaks with any change in the representation type.

In this thesis we present techniques to remedy the dilemma between type-safety on the one hand, and conciseness, reusability, and robustness on the other. For representative typed languages from the functional and object-oriented programming paradigms, *viz* Haskell and Java, we developed programming idioms that allow program construction from combinators which support *typeful generic traversal*. Using these combinators, program abstractions can be composed that capture e.g. reusable traversal strategies or analysis and transformation schemas. Though typeful, these abstractions need make little or no commitment to the specific type structure of the representations to which they are applied.

We have developed tool support to enable the application of our generic traversal techniques to source code representations that involve large numbers of different subelement types. These tools generate combinator support from SDF grammars. Parsers and pretty-printers can be generated from the same grammars, as well as the necessary code for representing and exchanging syntax trees between parsers, traversal components, and pretty-printers. In fact, SDF grammars are employed as contracts that govern all tree exchange, representation, and processing in a general multi-lingual architecture for source code analysis and transformation.

The practical applicability of all these techniques has been put to the test in several case studies, ranging from procedure reconstruction for Cobol programs, through static analysis of Toolbus scripts, to automated Java refactoring.

Samenvatting

In veel gebieden van de software engineering spelen analyse en transformatie van broncoderepresentaties een essentiële rol. In het algemeen zijn dit soort representaties zeer heterogene datastructuren. Voorbeelden hiervan zijn ontleedbomen, abstracte syntaxbomen, afhankelijkheidsgrafen, en aanroepgrafen. Bij voorkeur wordt de welgevormdheid van dit soort datastructuren bewaakt door een systeem van sterke, statische, types.

Wanneer men traditionele methoden gebruikt, is getypeerd programmeren helaas strijdig met beknoptheid, herbruikbaarheid en robuustheid van de code. Voor het benaderen en aflopen (*traversal*) van deelelementen van getypeerde representaties is men gedwongen grote aantallen specifieke types op specifieke manieren te behandelen. Hierdoor brengt type-veiligheid met zich mee dat code voor het aflopen van representaties langdradig is, niet herbruikt kan worden voor verschillende delen van de representatie of voor verschillende representaties, en bovendien aanpassing vereist bij elke verandering in het representatietype.

In dit proefschrift presenteren wij technieken om het dilemma te verhelpen tussen type-veiligheid enerzijds, en beknoptheid, herbruikbaarheid en robuustheid anderzijds. Voor representatieve getypeerde talen uit de paradigma's voor functioneel en object-geörienteerd programmeren, hebben wij programmeeridiomen ontwikkeld voor het samenstellen van programma's uit combinatoren die ondersteuning bieden voor het aflopen van representaties op een wijze die zowel *getypeerd* als *generiek* is. Met gebruikmaking van deze combinatoren kunnen programma-abstracties samengesteld worden die bijvoorbeeld herbruikbare navigatiestrategieën of analyse- en tranformatieschemas implementeren. Ondanks hun getypeerdheid zijn deze abstracties niet of nauwelijks gebonden aan de specifieke typestructuur van de representaties waarop zij worden toegepast.

Om het mogelijk te maken onze technieken voor het op generiek wijze aflopen van getypeerde representaties toe te passen op broncoderepresentaties die bestaan uit grote aantallen verschillende deelelementtypen, hebben wij ondersteunende gereedschappen ontwikkeld. Uit SDF grammatica's wordt door deze gereedschappen code gegenereerd die het programmeren met combinatoren ondersteunt. Ontleders en pretty-printers kunnen uit dezelfde grammatica's gegenereerd worden, alsmede alle code die benodigd is voor het representeren en uitwisselen van syntaxbomen tussen ontleders, pretty-printers, en de uit combinatoren samengestelde componenten. In feite worden zo SDF grammatica's gebruikt als contracten voor alle uitwisseling, representatie en verwerking van broncode in een algemene, meertalige architectuur voor broncode-analyse en -transformatie.

De praktische toepasbaarheid van al deze technieken beproefd in diverse situaties, variërend van procedure-reconstructie uit Cobol programma's, via statische analyse van Toolbus scripts, tot automatische herfactorisatie van Java programma's.

Titles in the IPA Dissertation Series

of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03