Centrum voor Wiskunde en Informatica

**REPORT***RAPPORT*

*SEN*

Software Engineering

***Software ENgineering***

# A Strafunski Application Letter

Ralf Lämmel, Joost Visser

**REPORT SEN-E0325 DECEMBER 23, 2003**

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)


P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

# A Strafunski Application Letter

ABSTRACT
Strafunski is a Haskell-centred software bundle for implementing language processing components — most notably program analyses and transformations. Typical application areas include program optimisation, refactoring, software metrics, software re- and reverse engineering. Strafunski started out as generic programming library complemented by generative tool support to address the concern of generic traversal over typed representations of parse trees in a scalable manner. Meanwhile, Strafunski also encompasses means of integrating external components such as parsers, pretty printers, and graph visualisation tools. In a selection of case studies, we demonstrate that typed functional programming in Haskell, augmented with Strafunski's support for generic traversal and external components, is very appropriate for the development of practical language processors. In particular, we discuss using Haskell for Cobol reverse engineering, Java code metrics, and Haskell re-engineering.

*1998 ACM Computing Classification System:* D.3.4; D.1.1
*Keywords and Phrases:* Program transformation; Program analysis; Language Processing; Haskell

# A *Strafunski* Application Letter

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

[1] CWI, Kruislaan 413, NL-1098 SJ Amsterdam
[2] Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
[3] Software Improvement Group, Kruislaan 419, NL-1098 SJ Amsterdam
Email: (Ralf.Laemmel|Joost.Visser)@cwi.nl
WWW: http://www.cwi.nl/~(ralf|jvisser)/

**Abstract.** *Strafunski* is a Haskell-centred software bundle for implementing language processing components — most notably program analyses and transformations. Typical application areas include program optimisation, refactoring, software metrics, software re- and reverse engineering.

*Strafunski* started out as generic programming library complemented by generative tool support to address the concern of *generic traversal* over typed representations of parse trees in a scalable manner. Meanwhile, *Strafunski* also encompasses means of *integrating external components* such as parsers, pretty printers, and graph visualisation tools.

In a selection of case studies, we demonstrate that typed functional programming in Haskell, augmented with *Strafunski*'s support for generic traversal and external components, is very appropriate for the development of practical language processors. In particular, we discuss using Haskell for Cobol reverse engineering, Java code metrics, and Haskell re-engineering.

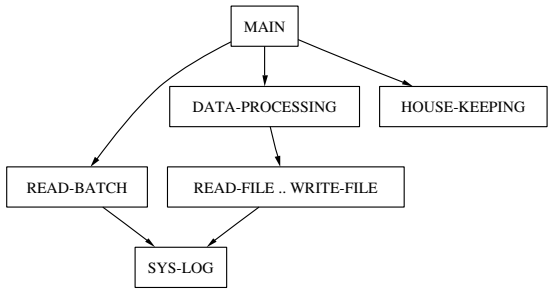
**Keywords:** *Strafunski*, Program transformation, Program analysis, Language processing, Generic traversal, External components, Interchange formats, Functional programming

## 1 Haskell meets Cobol

Consider the following software reverse engineering problem in the context of redocumentation of Cobol software. Given a Cobol program, we want to synthesise and view the so-called *perform graph*. It is called 'perform graph' because of Cobol's verb `PERFORM` for procedure invocation. Such a graph helps maintenance programmers to understand the control flow of Cobol programs of non-trivial size: typical Cobol programs are about 1500 lines, but individual programs of 25,000 lines are not uncommon. A perform graph contains nodes for each procedure, and edges for each procedure invocation. The perform graph of a simple Cobol program is shown in Fig. 1. Roughly, a perform graph is computed as follows:

1. Find all `PERFORM`s to reconstruct what labelled code blocks represent procedures.
2. Reconstruct the main procedure of the program by a kind of control-flow analysis.
3. Find all `PERFORM`s per procedure to determine outgoing procedure invocations.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ...
DATA DIVISION.
  WORKING-STORAGE SECTION.
  ...
PROCEDURE DIVISION.
   PERFORM READ-BATCH.
   PERFORM DATA-PROCESSING.
   PERFORM HOUSE-KEEPING.
   STOP RUN.
 READ-BATCH.
   ...
   PERFORM SYS-LOG.
 DATA-PROCESSING.
   PERFORM READ-FILE
      THRU WRITE-FILE.
 READ-FILE.
   ...
   PERFORM SYS-LOG.
 UPDATE-BUFFER.
   ...
 WRITE-FILE.
   ...
   PERFORM SYS-LOG.
 SYS-LOG.
   ...
 HOUSE-KEEPING.
   ...
```




There are no language constructs for procedure declarations in Cobol. Instead, labelled *sections* and *paragraphs* are the targets of both `PERFORM` and `GOTO` statements. In fact, `PERFORM` statements may also invoke *ranges* of sections and paragraphs. Hence, the entry and exit points of performed blocks of code are defined indirectly. In the sample program, the paragraphs `READ-FILE`, `UPDATE-RECORD`, and `WRITE-FILE` form a procedure because of the `PERFORM ... THRU ...` statement in the `DATA-PROCESSING` paragraph. The reconstruction of a Cobol program's main procedure requires a decent control-flow analysis because one had to observe `GOTO`s and other control-flow patterns.

**Fig. 1.** A Cobol program and the corresponding perform graph.

In addition to the specific problem of defining precisely how to compute a perform graph, there are more general complications that surround the implementation of a perform-graph extractor in a practical setting. These include the extraordinary size of the Cobol language, the proliferation of its dialects, the size of the typical code base to be processed, and the realities of limited budgets and time frames.

In this paper, we report on using *typed functional programming in Haskell* to implement problems like the one above. Haskell seems to be suited for language processing: meta-programs in Haskell operate on representations of object-programs based on algebraic datatypes. However, in the typical textbook approach, two bits are missing, namely support for generic traversal, and integration of external components:

**Generic traversal** We must be able to employ generic programming techniques. That is, we want to deal generically with all language constructs that are not immediately relevant to our problem. As for the discussed perform-graph extractor, we do not want to take all of the several hundred syntactic elements of Cobol into account, but only `PERFORM`s and code blocks. Also, every time a new dialect or language cocktail pops up (think of embedded SQL or CICS, in-house preprocessors, OO Cobol), we want to adapt the tool with minimal effort.

**External components** We must be able to integrate external components on the basis of suitable interchange formats. As for the perform-graph extractor, we want to reuse an existing Cobol parser. Note that the development of an industrial-strength Cobol parser from scratch takes at least a few months, and choosing the right parsing technology is crucial for scalability. Other typical external components are graph visualisers, browsers, pretty printers, and databases.

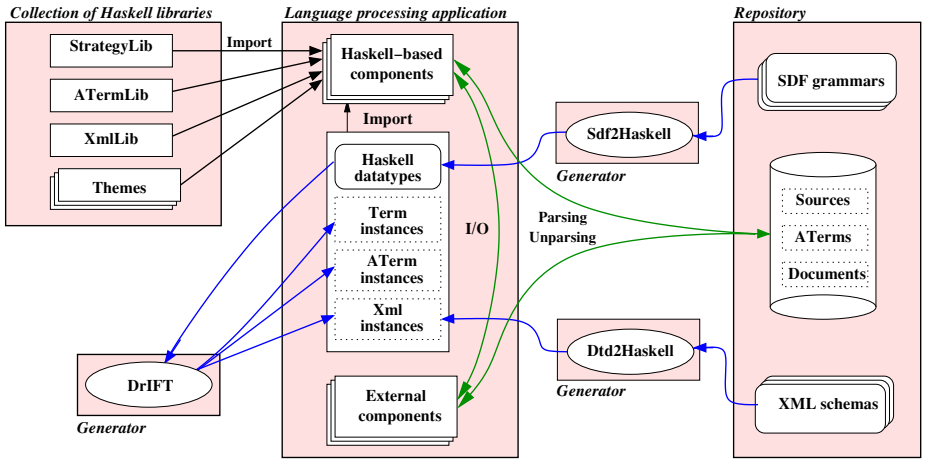


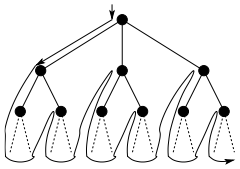

**Fig. 2.** Haskell-centred language processing with *Strafunski*.

The Haskell-centred software bundle *Strafunski* [1] addresses these two concerns as illustrated in Fig. 2. The block labelled 'language processing application' emphasises that Haskell-based and external components coexist in an application. The components communicate on the basis of the interchange formats XML and ATerms [2], or they access a repository with source programs and XML documents. Haskell-based components take advantage of generic programming with 'functional strategies' [12, 11, 8] based on *Strafunski*'s Haskell library *StrategyLib*. Functional strategies are generic functions that can traverse into terms of any type while mixing type-specific and uniform behaviour. Here we assume that algebraic datatypes serve for the typed representation of parse trees. Strategic programming in Haskell relies on supportive code per term type. The corresponding instances of a *Term* class can be generated using the *DrIFT* preprocessing technology [17]. The algebraic datatypes might be derived from XML schemas (or DTDs) and syntax definitions in SDF [5]; see the generators *Sdf2Haskell* and *Dtd2Haskell*. There are further Haskell libraries: *XmlLib* for XML document processing (contributed by HaXML [16]) and *ATermLib* for data interchange. *DrIFT* is also used to generate *XML* instances and *ATerm* instances needed as mediators between Haskell terms and the interchange formats. The collection of libraries also encompasses themes for language processing such as name analyses and refactorings [9].

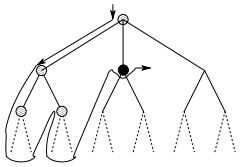
**Road-map** This application letter reports on the use of the *Strafunski* bundle for the implementation of language processing tools. We discuss three case studies: reverse engineering (Sec. 4), software metrics (Sec 5), and re-engineering (Sec. 6). The object languages involved are Cobol, Java, and Haskell, respectively. Before we embark on the case studies, we explain *Strafunski*'s two contributions to language processing in Haskell: generic traversal (Sec. 2) and external components (Sec. 3).

[1] *Strafunski* home page: `http://www.cs.vu.nl/Strafunski/` — `Stra` refers to strategies, `fun` refers to functional programming, and their harmonious composition is a homage to the music of Igor Stravinsky.

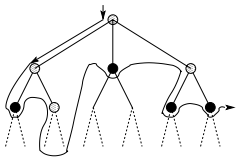
*full_td*:
Process all nodes.



*once_td*:
Process one subtree.



*stop_td*:
Descend into failing subtrees.



Typically, a traversal scheme takes one argument strategy for node processing. The above schemes all stick to a top-down, left-to-right order of node processing but they vary as for the coverage of nodes (indicated by bullets; black nodes denote success; dashed nodes denote failure). In the case of *full_td*, node processing is assumed to succeed for all nodes while success and failure behaviour controls descent for *once_td* and *stop_td*.

**Fig. 3.** Full traversal vs. single-hit traversal vs. cut-off traversal.

## 2 Generic traversal

**Strategic programming** The key idea underlying the *Strafunski*-style of generic programming is to view traversals as a kind of generic functions that can traverse into terms while mixing uniform and type-specific behaviour. In [12], we defined *functional strategies* accordingly. Strategies are composed via *function combinators*. *Strafunski* supports 'strategic programming' via the library *StrategyLib* of reusable strategy combinators, and the *DrIFT* generator for supportive code for user-supplied Haskell datatypes. Recall that the Haskell datatypes are typically generated from SDF grammars or XML DTDs. Hence, functional strategies allow us to process parse trees and XML documents in both *typed-based* and *generic* manner. By contrast, the HaXML [16] combinator library for generic XML processing is DTD-unaware.

**Strategy combinators** We qualify combinators by a postfix `TP` vs. `TU` for 'type preservation' or 'type unification' resp. to point out if they deal with transformation or analysis. The originality of strategic programming arises from the following concepts:

- Update strategies by type-specific cases (denoted by `adhocTP` and `adhocTU`).
- One-layer traversal that acts on immediate subterms (e.g., `allTU` for reduction).

Using strategy update, ingredients for actual traversals can be composed. Using one-layer traversal combinators, all kinds of traversal schemes can be assembled as recursive functions (see [8] for the design space). Three frequently used schemes are illustrated in Fig. 3. The first one can be defined as follows for the `TU` case:

```
full_tdTU s x =
  (s x) `mappend` (allTU mappend mempty (full_tdTU s) x)
```

This reads as "apply $s$ to the term $x$, and then recurse into `all` immediate subterms of $x$ while combining the intermediate results with the binary operation `mappend` of a monoid using the 'unit' `mempty` as initial value". Assembling traversal schemes is actually a rather rare activity. Mostly, one reuses schemes defined in *StrategyLib*.

```
-- Synthesis of the traversal
findPerforms = applyTU ( full_tdTU step)
 where step = constTU [] `adhocTU` matchPerform

-- Type-specific case
matchPerform (Perform Nothing _ _ _ _ ) = return []
matchPerform (Perform (Just (Perform_procedure p thru)) _ _ _ _)
  = return [(procedure_name2string p,
              case thru of
               Nothing -> Nothing
               (Just (Through_label _ p')) -> Just (procedure_name2string p'))]
```

The function `findPerforms` performs a pattern match according to `matchPerform` all over the place, and it accumulates the identified invocations as a list using the `full_tdTU` traversal scheme. The function `matchPerform` extracts the referenced labels from a given `PERFORM` statement. The first equation covers an 'inline' `PERFORM` statement which does not refer to any label. Hence, the empty list `[]` is returned. The second equation deals with `PERFORM`s that actually invoke labelled code blocks. There is an extra `case` discrimination for the optional end label (see patterns `Nothing` and `Just` ...). So it returns a singleton list with a pair of the type `(String, Maybe String)` corresponding to the start label and an optional end label.

**Fig. 4.** Find all `PERFORM`s in Cobol program.

**Traversal design** The most frequently used *design patterns* for strategic programming [11] are to define 'rewrite steps' and to synthesise actual 'traversals'. A *type-specific* rewrite step is a monomorphic function that cares about problem-specific patterns. One obtains a *generic* rewrite step by 'lifting' one or more type-specific rewrite steps to the strategy level. That is, the steps are used to update a default strategy (recall `adhocTP` and `adhocTU`). An actual traversal is synthesised by simply passing a (generic) rewrite step as a parameter to the suitable traversal scheme. This is illustrated in the following Haskell code skeleton:

$$\begin{aligned} &\mathit{traversal}\ \mathit{term}\ =\ \mathit{apply}\ (\mathit{scheme}\ \mathit{step})\ \mathit{term} \\ &\quad \mathbf{where}\ \mathit{step}\ =\ \mathit{default}\ \text{`}\mathit{adhoc}\text{`}\ \mathit{mono}_1\ \cdots\ \text{`}\mathit{adhoc}\text{`}\ \mathit{mono}_n \end{aligned}$$

Here, *apply* is a place holder for an explicit application combinator `applyTP` or `applyTU` (needed for technical reasons). The place holder *scheme* can be resolved to a traversal scheme such as `full_tdTU` from above. The infix operator *adhoc* is a place holder for either `adhocTP` or `adhocTU`. The $mono_1, \ldots, mono_n$ are type-specific rewrite steps. Common *defaults* are the following:

- the identity strategy `idTP`,
- a constant strategy of the form `constTU` $u$, or
- the always failing strategy `failTP` or `failTU`.

In Fig. 4, the code skeleton is illustrated for our running example of a Cobol perform-graph extractor. The shown *traversal* `findPerforms` implements the first step of the extraction process: it collects all procedure invocations via the `full_tdTU` scheme. The generic rewrite *step* returns the empty list by *default* (cf. `constTU []`), and the single type-specific rewrite step `matchPerform` destructs actual `PERFORM` statements to retrieve the relevant labels. To summarise, this very concise and adaptive style of programming effectively focuses on the patterns that are relevant for a given problem.

```
-- The abstract syntax of ATerms
data ATerm = AAppl String [ATerm]       -- Application
           | AList [ATerm]              -- Lists
           | AInt Integer               -- Integers
           deriving (Read,Show,Eq,Ord)

-- Mediation between ATerms and algebraic datatypes
class ATermConvertible t where
  toATerm   :: t -> ATerm
  fromATerm :: ATerm -> t
```

While ATerms are suitable for import, export, run-time representation, and external storage, they are not directly suited for programming on syntaxes or formats because of the lack of typing. Due to the provisions of the *Strafunski* architecture, one can turn terms of any algebraic datatype into an ATerm (cf. member `toATerm`) and vice versa (cf. member `fromATerm`).

**Fig. 5.** ATerm support in Haskell.

## 3 External components

**Complementary interchange formats** The *Strafunski* architecture features integration of external components on the basis of the interchange formats XML and ATerms. For short, the low-level ATerm format is particularly suited for the integration of language processing components, whereas XML is favoured when we deal with application-specific data models (say, import/export formats). For XML, we rely on the type-based translation facilities of HaXML [16]. Given an XML document type definition (DTD), the HaXML tool *Dtd2Haskell* generates a corresponding system of Haskell datatypes, and instances of the *XmlContent* class. These instances, in combination with the HaXML library *XmlLib*, allow translation of terms over the generated datatypes to XML documents that adhere to the input DTD, and vice versa. The ATerm format for annotated terms [2] is a very simple, untyped interchange format that was designed specifically for component-based development of language tools. It is used in other language processing environments, too [1, 6]. The ATerm format supports data compression through maximal subterm sharing. The *Strafunski* architecture features two elements for ATerm support. Firstly, the Haskell ATerm library *ATermLib* provides a datatype for the representation of ATerms together with a class *ATermConvertible* for conversion between ATerms and algebraic datatypes (see Fig. 5). Secondly, the *DrIFT* preprocessor can be used to generate instances of the *ATermConvertible* class for any given Haskell datatype.

**Parser integration** A typical kind of external components that we need to integrate with Haskell components are *parsers*. In the *Strafunski* architecture, we provide support for one specific syntax definition formalism, namely SDF [5]. This is a suitable candidate for extending the capabilities of an otherwise Haskell-centred architecture for language processing. Firstly, SDF provides a very general grammar format. Secondly, SDF is supported by powerful parsing technology, namely (scannerless) generalised LR parsing with an implementation that matured inside the Meta-Environment [1].

```
– – SDF grammar fragment dealing with code blocks in Cobol
Paragraphs Section-with-header*         -> Sections {cons("Sections")}
Section-header "." Paragraphs           -> Headed-section {cons("Headed-section")}
Section-name "SECTION" Priority-number? -> Section-header {cons("Section-header")}
Sentence* Paragraph*                    -> Paragraphs {cons("Paragraphs")}
Paragraph-1                             -> Paragraph {cons("Paragraph-1")}
Altered-goto                            -> Paragraph {cons("Altered-goto")}
Paragraph-name "." Sentence*            -> Paragraph-1 {cons("Paragraph-11")}
Paragraph-name "." "GO" "TO"? "."       -> Altered-goto {cons("Altered-goto1")}
Statement-list "."+                     -> Sentence {cons("Sentence")}

– – Haskell counterpart (generated algebraic datatypes)
data Sections       = Sections Paragraphs [Headed_section]
data Headed_section = Section_with_header Section_header Paragraphs
data Section_header = Section_header Section_name (Maybe Priority_number)
data Paragraphs     = Paragraphs [Sentence] [Paragraph]
data Paragraph      = Paragraph_1 Paragraph_1
                    | Altered_goto Altered_goto
data Paragraph_1    = Paragraph_11 Paragraph_name [Sentence]
data Altered_goto   = Altered_goto1 Paragraph_name (Maybe ())
data Sentence       = Sentence Statement_list [()]
```

The derivation of algebraic datatypes from context-free grammars (EBNF, YACC, SDF, etc.) is largely straightforward. The above snippets illustrate the following techniques. The alternative productions that define a nonterminal amount to the different constructors of an algebraic datatype. Constructor annotations (`cons(...)`) in the grammar are used as proposals for constructor names in the algebraic datatypes. There are direct mappings for EBNF operators $+$, $*$, ? in Haskell, namely the `List` and the `Maybe` datatype. Keywords can be omitted from the algebraic representation.

**Fig. 6.** Fragment of the *VS COBOL II* grammar [10] and its Haskell counterpart.

Thirdly, we have access to an SDF-based grammar base[2] with grammars for several languages. Fourthly, SDF is supported by the Grammar Deployment Kit[3] (GDK) [7]. This kit can generate parsers for different parsing technologies, e.g., for YACC with backtracking [13], C-based combinator parsing as supported by GDK itself, generalised LR parsing, and Haskell-based combinator parsing.

In the *Strafunski* architecture, the tool *Sdf2Haskell* supports the derivation of algebraic Haskell datatypes from an SDF grammar. For a Cobol fragment, this correspondence is illustrated in Fig. 6. We assume that the external parsers emit parse trees in the ATerm format. Then, Haskell components can read in the parse trees relying on the *ATermConvertible* instances for the Haskell datatypes that correspond to the grammar. We dwell upon the fact that parser reuse is crucial by referring to Cobol again. A plain Cobol grammar has about 1000 productions (assuming EBNF operators for lists and optionals), not talking about extensions for SQL, CICS and others. Just this size rules out a manual approach to parser development – using maybe Haskell parser combinators. In fact, implementing a Cobol grammar specification is a challenge for any technology. To implement a scalable Cobol parser, one needs to resolve grammar conflicts or ambiguities, provide provision for error recovery and parse tree construction, and tweak for non-context-free constructs and performance.

[2] `http://www.program-transformation.org/gb/`
[3] `http://gdk.sourceforge.net/`

```
1  main = do prg      <- parseCobol                -- Parsing
2            dotGraph <- toPerformGraph prg        -- Graph synthesis
3            putStrLn dotGraph                     -- Output
4
5  toPerformGraph prg
6   = do name    <- getProgramName prg
7        procs   <- findPerforms prg               -- see Fig. 4
8        main    <- findMain prg
9        perproc <- findPerformsPerProc procs prg
10       inmain  <- findPerformsInMain main
11       return (mkGraph (name++" Perform Graph") (perproc++inmain))
```

We obtain the parse tree of the Cobol source program by the invocation of the external parser and converting its untyped ATerm output into a heterogeneously typed Haskell representation of the parse tree (line 1). The perform-graph generation synthesises the perform graph as a valid input string for the visualisation tool based on a simple API (line 2). The perform graph is then just written to the stdout (line 3). The actual synthesis of the perform graph consists of a number of steps (lines 6–11) as outlined on the paper's first page. The steps are arranged in a monadic do-sequence to be prepared for aspects such as I/O, failure or debugging.

**Fig. 7.** Top-level program structure of the perform-graph extractor for Cobol.

## 4 Case study I: Cobol reverse engineering

We complete our running example of a perform-graph extractor for Cobol. Its top-level functionality is shown in Fig. 7. We already described the approach to reusing an external Cobol parser, and the implementation of the first step in the synthesis of the perform graph, that is, to extract all `PERFORM`s. We will skip over the second step, that is, the identification of the main procedure of a Cobol program. Below we work out the third step, namely the identification of `PERFORM`s per procedure. In addition to this traversal functionality, we will also explain the integration of a graph visualisation tool.

**Find all `PERFORM`s per procedure** The simple traversal `findPerforms` (recall Fig. 4) provided us with the nodes in the perform graph. We shall now identify the edges in the graph, that is, we need to determine the outgoing procedure invocations for each previously identified procedure. The implementation of this idea is complicated by the fact that we need to deal with procedures *spanned* over several paragraphs or sections as triggered by `PERFORM` ... `THRU` ... statements. We basically have to look up intervals from lists of paragraphs and sections according to the identified procedure labels. Once we retrieved a relevant code block, we apply the simply traversal `findPerforms` to find all `PERFORM`s in the block. The corresponding piece of traversal functionality is found in Fig. 8.

**Graph visualisation** We integrate the `dot` tool as an external component for graph visualisation. We simply export the synthesised perform graph in the `dot` input format. We use a rather direct approach, that is, the corresponding API maps the given nodes (i.e., procedures) and edges (i.e., invocations) to plain strings adhering to the `dot` input format. The API is included in Fig. 9. This approach is very lightweight. Recall Fig. 1 where we illustrated the visual output of the extractor. In more demanding contexts, APIs preferably synthesise a public or opaque intermediate representation as opposed to plain strings. This adds type safety, and it enables subsequent processing of the intermediate representation.

```
1  findPerformsPerProc procs = applyTU ( full_tdTU step)
2   where
3    step = constTU [] `adhocTU` matchParagraph `adhocTU` matchParagraphList
4                      `adhocTU` matchSection   `adhocTU` matchSectionList
5
6    matchParagraph (Paragraph_11 pname sentences)
7     = do name <- return $ paragraph_name2string pname
8          singletonBlock name sentences
9
10   matchParagraphList (paragraphs::[Paragraph])
11    = do results <- mapM (rangeInParagraphs paragraphs) procs
12         return (concat results)
13
14   matchSection     ... = ... -- omitted for brevity
15   matchSectionList ... = ... -- omitted for brevity
16
17   singletonBlock name block
18    = if (name,Nothing) `elem` procs
19       then scanBlock name block else return []
20
21   scanBlock name block
22    = do procs  <- findPerforms block -- Find all PERFORMs in code block
23         node   <- return $ mkProcedure name
24         edges  <- return $ map (mkPerform name) procs
25         return (node:edges)
26
27   rangeInParagraphs _ (_, Nothing) = return [] -- no range but a singleton block
28   rangeInParagraphs paragraphs (start, Just end)
29    = let spanned = fromto ((==) start . getParagraphName)
30                           ((==) end   . getParagraphName)
31                           paragraphs
32      in scanBlock (mkRangeName start end) spanned
```

The traversal `findPerformsPerProc` employs the `full_tdTU` traversal scheme (line 1), and it exhibits type specific behaviour for paragraphs, sections, and lists thereof (lines 3–4). Given a single paragraph (see `matchParagraph`; lines 6–8), we investigate whether it constitutes a procedure (see `singletonBlock`; lines 17–19). In case it is, we scan this block for edges in the perform graph (see `scanBlock`; lines 21–25). The type-specific case for *lists* of paragraphs (see `matchParagraphList`; lines 10–12) maps over `procs` (line 11) to check for every element if it happens to refer to a range of paragraphs in the given list (see `rangeInParagraphs`; lines 27–32). To this end, we attempt to split up the list using the labels at hand as boundaries. Here we assume a helper `fromto` to select an interval of a list via predicates (used in line 29; definition omitted). Once we retrieved a code block consisting of a number of paragraphs, we invoke `scanBlock` (line 32) to scan this block for edges in the perform graph.

**Fig. 8.** Find `PERFORM`s per Cobol procedure.

```
mkGraph name ascii      = "digraph "++(quote name)++" {\n"++(concat ascii)++"}\n"
mkProcedure p           = (quote p)++" [ shape=box ]\n"
mkPerform f (t,Nothing) = (quote f)++" -> "++(quote t)++"\n"
mkPerform f (t,Just t') = (quote f)++" -> "++(quote (mkRangeName t t'))++"\n"
mkRangeName t t'        = t ++ ".." ++ t'
```

`mkGraph` completes the ASCII content of a `dot` graph (i.e., a list of strings for nodes and edges) into a complete `dot` input with the given `name`; `mkProcedure` and `mkPerform` derive the nodes and edges in ASCII from procedure names and `PERFORM` labels; `mkRangeName` builds an ASCII representation for labels in a `PERFORM` ... `THRU` ... statement.

**Fig. 9.** API for `dot`-file generation from Cobol perform graphs.

```
– – Relevant Java statement syntax in SDF
LabelledStatement                   -> Statement {cons("LabelledStatement")}
ClassDeclaration                    -> Statement {cons("ClassDeclaration")}
StatementWithoutTrailingSubstatement -> Statement {cons("WithoutTrailing")}
Block          -> StatementWithoutTrailingSubstatement {cons("Block")}
EmptyStatement -> StatementWithoutTrailingSubstatement {cons("EmptyStatement")}
";"                      -> EmptyStatement    {cons("semicolon")}
Identifier ":" Statement -> LabelledStatement {cons("colon")}

– – Counting statements; not counting certain constructors
1  statementCounter :: Term t => t -> Int
2  statementCounter = runIdentity . applyTU (full_tdTU step)
3   where
4    step = constTU 0 `adhocTU` statement `adhocTU` localVarDec
5    statement s = case s of
6     (WithoutTrailing (EmptyStatement _)) -> return 0
7     (WithoutTrailing (Block _))          -> return 0
8     (LabelledStatement _)                -> return 0
9     (ClassDeclaration _)                 -> return 0
10    _                                    -> return 1
11
12    localVarDec (_::LocalVariableDeclaration) = return 1
```

The type of the `statementCounter` (line 1) points out that this traversal can be applied to any term type `t`, and that the result type is an `Int`eger. The statement count is computed by performing a `full_tdTU` traversal with one 'tick' per statement. We do not count empty statements (`return` 0 in line 6). We also do not let a block statement, a labelled statement or a class declaration statement contribute to the statement count (lines 7–9), but only the statement(s) nested inside them. The catch-all case for `statement` 'ticks' for all other statements (`return` 1 in line 10). Finally, we want local variable declarations to contribute to the count, and hence an extra type-specific case for the traversal is needed (line 12).

**Fig. 10.** Counting Java statements.

## 5 Case study II: Java code metrics

In this section we discuss the implementation of the calculation of metrics for Java applications on the basis of source code. Such metrics are useful for determining volume and quality of Java code as required for the estimation of maintenance costs. We discuss the implementation of three metrics:

- *Statement count*: the number of statements.
- *Cyclometric complexity*: the number of conditionals (McCabe).
- *Nesting depth*: the maximal depth of nested conditionals.

We want to compute these metrics not only for an entire Java application, but also per method and per class or interface. Furthermore, we want to export the computed metrics in XML format for further processing. As in the case of Cobol, we employ an external parser component. Indeed, Java's grammar is available in the SDF grammar base.

**Metrics computation** The number of statements in any fragment of Java code can basically be computed by counting the number of nodes of type `Statement` in the corresponding parse tree. A full traversal is appropriate. A few exceptions are in place for the sake of precise counting. Fig. 10 shows the relevant productions from the Java grammar, and the implementation of statement count. The cyclometric complexity (or

```
1  mcCabeIndex :: Term t => t -> Int
2  mcCabeIndex = unJust . applyTU (full_tdTU step)
3   where
4    step = ifTU isConditional       -- potentially failing strategy
5                (const (constTU 1)) -- 'then' branch; value consumer
6                (constTU 0)         -- 'else' branch
7
8  isConditional
9   = failTU -- resolves to: const mzero = const Nothing
10           'adhocTU' (\(_::IfThenStatement)     -> return ())
11           'adhocTU' (\(_::IfThenElseStatement) -> return ())
12           'adhocTU' (\(_::WhileStatement)      -> return ())
13           'adhocTU' (\(_::ForStatement)        -> return ())
14           'adhocTU' (\(_::TryStatement)        -> return ())
```

In Java, the statements that contribute to the cyclometric complexity are not only conditionals and loops, but also the *try* statement associated to the exception handling mechanism. Recognition of a relevant statement is modelled via success and failure of the helper strategy `isConditional` (lines 8–14). Note that only types are matched but not patterns (see the type annotations '`::`'in lines 10–14). This is because of the particular format of the Java grammar that defines nonterminals for several statement forms. In the rewrite `step` for the full traversal (lines 4–6), success and failure behaviour is mapped to 1 vs. 0 by using a strategy combinator `ifTU`.

**Fig. 11.** Computing cyclometric complexity.

```
-- Java metrics via instantiation of generic metrics
nestingDepth :: Term t => t -> Int
nestingDepth = unJust . applyTU (depthWith isConditional)

-- Generic algorithm for depth of nesting
1  depthWith s
2    = recurse 'passTU'     -- Sequential composition
3        \depth_subterms ->
4            let max_subterms = maximum (0:depth_subterms)
5             in (ifTU s
6                      (const (constTU (max_subterms + 1)))
7                      (constTU max_subterms))
8   where
9    recurse = allTU (++) [] (depthWith s 'passTU' \depth -> constTU [depth])
```

Generic depth calculation works as follows. We first compute a list of depths for the various subterms (line 2) by recursing into them. The helper `recurse` does not employ any recursive traversal scheme, but we use *Strafunski*'s basic one-layer traversal combinator `allTU` (line 9) to apply the strategy for depth calculation to all *immediate* subterms. This setup for recursion leads to the needed awareness of nesting. From the list of depths, we compute the maximum depth (line 4), and then we complete this maximum to take the current term into account. If the recogniser succeeds for the term at hand, then we add 1 to the maximum (lines 5–7).

**Fig. 12.** Computing nesting depth of conditionals.

McCabe index) of a fragment of Java code can again be computed by a full traversal. This time we need to count the occurrences of conditional and looping constructs in the corresponding parse tree. The implementation is shown in Fig. 11. Note that the rewrite `step` for the traversal employs a strategy `isConditional` that merely serves as a 'recogniser' of relevant constructs as opposed to 'ticking'. This is expressed by a predicate-like result type `Maybe ()`. The actual ticking is done separately on the use site of `isConditional`. This style is more suitable for the reuse of the pattern recog-

```
<!DOCTYPE javaMetrics [
  <!ELEMENT javaMetrics (compilationunitMetric*) >
  <!ELEMENT compilationunitMetric (interfaceMetric | classMetric)* >
    <!ATTLIST compilationunitMetric name CDATA #REQUIRED>
  <!ELEMENT interfaceMetric EMPTY >
    <!ATTLIST interfaceMetric name        CDATA #REQUIRED
                              methodCount CDATA #REQUIRED
                              fieldCount  CDATA #REQUIRED>
  <!ELEMENT classMetric (methodMetric | classMetric)* >
    <!ATTLIST classMetric name       CDATA #REQUIRED
                          fieldCount CDATA #REQUIRED>
  <!ELEMENT methodMetric (classMetric)* >
    <!ATTLIST methodMetric name           CDATA #REQUIRED
                           statementCount CDATA #REQUIRED
                           mcCabe         CDATA #REQUIRED
                           nestingDepth   CDATA #REQUIRED>
]>
```

The structure of Java metrics documents roughly follows the syntactical structure of Java itself, but in a highly condensed manner. The attributes of the document elements contain the names of these elements and the values of various metrics.

**Fig. 13.** A DTD for Java metrics documents.

```
1  extractClassMetrics :: ClassDeclaration -> Maybe ClassMetric
2  extractClassMetrics (Class1 _ name extends implements body)
3   = do nestedClassMetrics <- mapM extractClassMetrics (getNestedClasses body)
4        methodMetrics      <- mapM extractMethodMetrics (getMethods body)
5        return $ ClassMetric
6                  ClassMetric_Attrs {
7                    classMetricName       = str2cdata name,
8                    classMetricFieldCount = int2cdata (length (getFields body)) }
9                  ((map ClassMetric_ClassMetric nestedClassMetrics)++
10                  (map ClassMetric_MethodMetric methodMetrics))
11
12 extractMethodMetrics :: MethodDeclaration -> Maybe MethodMetric
13 extractMethodMetrics (MethodHeader_MethodBody header body)
14  = do name               <- getMethodName header
15       statCount          <- statementCounter body
16       mcCabe             <- mcCabeIndex body
17       nestingDepth       <- nestingDepth body
18       nestedClasses      <- collectNestedClasses body
19       nestedClassMetrics <- mapM extractClassMetrics nestedClasses
20       return $ MethodMetric
21                 MethodMetric_Attrs {
22                   methodMetricName           = str2cdata name,
23                   methodMetricStatementCount = int2cdata statCount,
24                   methodMetricMcCabe         = int2cdata mcCabe,
25                   methodMetricNestingDepth   = int2cdata nestingDepth }
26                 nestedClassMetrics
27    where
28     collectNestedClasses = applyTU (stop_tdTU getClassDecl)
29     getClassDecl = failTU 'adhocTU' (\(cd::ClassDeclaration) -> return [cd])
```

The traversal for computing and storing metrics is structured as five cooperating functions, one for each DTD element. For brevity, we show the most interesting ones for class metrics and for method metrics. Trivial helper functions are omitted. The shown code performs little traversal on its own but pattern matching (lines 2 and 13) and list processing (lines 3,4,9,10,19) is usually sufficient. The only exception is to look up nested classes (line 28). Here we use a traversal with *stop* because we only want to gather immediate nested classes and not the transitive closure.

**Fig. 14.** Computing Java metrics and storing them in XML.

niser in the implementation of other metrics. Indeed, for the nesting-depth metric, the same statements are relevant as for cyclometric complexity, but the traversal behaviour is more involved. That is, we need to count levels of nesting rather than simply certain kinds of nodes. The implementation is shown in Fig. 12. The actual problem of counting levels of nesting is completely generic, and hence it is captured in a strategy combinator `depthWith` that is parameterised by a strategy for pattern recognition. The depth of a given term is the maximum of the depths of its children, possibly incremented by 1 if the term itself is relevant. Nesting depth for Java is then simply computed by passing `isConditional` to `depthWith`.

**Exporting to XML** To process metrics information by external components such as viewers, report generators, code browsers, and others, we use XML as interchange format. A DTD that describes the structure of Java metric documents is shown in Fig. 13. The *Dtd2Haskell* tool generates the corresponding system of Haskell datatypes. These datatypes are then used in our Haskell component to collect the results of our metrics calculations. The implementation is shown in Fig. 14. After the metrics have been computed, we invoke the conversion provided by *Dtd2Haskell* to export the metrics to an XML document that adheres to our metrics DTD.

## 6 Case study III: Haskell re-engineering

As a third object language for language processing we selected Haskell—not so much for the size of its grammar but rather because it is a complicated language with modules, overloading, type inference, nested scopes, and higher-orderness. While the other case studies concerned *analysis* problems, the Haskell case study deals with *transformation*. Two forms of dead code elimination shall be discussed:

- Elimination of dead local declarations in nested scopes.
- Elimination of dead top-level declarations in a chased module hierarchy.

The first form is a simple 'clean-up' refactoring while the second form generalises dead code elimination to the inter-modular level of a complete application. In fact, these transformations do not just serve a purpose in software re-engineering (in the sense of code improvement). They are also valuable for application extraction in software packaging, or for optimising compilation. Our tooling for the implementations of the transformations reuse available support for Haskell parsing and pretty printing (formerly called *hsparser* or *hssource*, now part of the Haskell Core Libraries — in the haskell-src package).

**Elimination of dead local declarations** In the given scope of a Haskell pattern match equation, a local 'where' declaration is dead if it is neither used by the right-hand side expression, nor by other declarations in the same group of bindings. Fig. 15, specifies the corresponding transformation by a generic traversal. Note that we rely on an analysis `hsFreeAndDeclared` for free and declared names in a Haskell program fragment. It is needed to decide whether a given abstraction is used. We also need a variant `hsFreeAndDeclaredGroup` that specifically deals with groups of bindings. The needed name analysis will be explained below.

```
1  elimDeadWheres :: Term t => t -> Maybe t
2  elimDeadWheres = applyTP (full_tdTP step)
3    where
4      step = idTP `adhocTP` match
5      match (HsMatch sl fun pats rhs wheres)
6        = do (pf,pd) <- hsFreeAndDeclared pats
7             (rf,rd) <- hsFreeAndDeclared rhs
8             (df,dd) <- hsFreeAndDeclaredGroup wheres
9             wheres' <- filterM (hsDeclUsed ((df `union` rf) \\ pd)) wheres
10            return (HsMatch sl fun pats rhs wheres')
11
12 hsDeclUsed names decl
13   = do (_,[name]) <- hsFreeAndDeclared decl
14        return $ name `elem` names
```

A full traversal is used (line 2) because declarations can be arbitrarily nested in Haskell. The rewrite step behaves like the identity function by default with a type-specific case for pattern match equations (line 4). Such equations are treated as follows (lines 5–10). We first destruct the `HsMatch` construct (line 5). Then, the free and declared names are determined for the various fragments in this scope (line 6 for the patterns on the left-hand side; line 7 for the expression on the right-hand side; line 8 for the local declarations). Then, we filter the declarations to only keep those that are actually used (line 9). Finally, we return the reconstructed pattern match equation with the filtered list of local declarations as the result of this rewrite step for transformation (line 10). The helper `hsDeclUsed` (lines 12–14) is a shorthand for determining the name defined by a declaration and performing a membership test with respect to a given set of names.

**Fig. 15.** Elimination of dead local declarations (meta-language = object-language = Haskell).

```
1  elimDeadTops ::       [(ModuleName,[ModuleName],HsModule)]
2               -> Maybe [(ModuleName,[ModuleName],HsModule)]
3
4  elimDeadTops l@(h:t)                                -- h is the main module
5    = do l' <- mapM worker t >>= return . (:) h       -- elimination per module
6         if l==l' then return l else elimDeadTops l' -- fixpoint by equality
7    where
8      worker (n,i,m@(HsModule n' i' e' ds))
9        = do clients <- return $ filter (\e@(_,i'',_) -> n `elem` i'') l
10            (imp,_) <- hsFreeAndDeclared clients
11            ds'     <- filterM (hsDeclUsed imp) ds
12            return (n,i,HsModule n' i' e' ds')
```

The elimination function operates on lists of modules which are tupled with, for convenience, the name of the module, and the imported modules (see the type in lines 1–2). The head of the list is the main module to be preserved as is. We continuously map a `worker` transformation over the chased modules (line 5) until no more top-level declarations are eliminated (line 6). The worker first determines all `clients` of the given module (line 9), that is, the modules that happen to import the given module. Then, we determine all top-level declarations used by these clients (line 10). Then, we filter away all dead top-level declarations of the given module accordingly (line 11). Finally, the module is reconstructed (line 12). The shown implementation only takes simple Haskell forms of module import into account (i.e., no selection, no re-export, and others).

**Fig. 16.** Inter-modular dead code elimination (meta-language = object-language = Haskell).

**Inter-modular dead code elimination** The transformation to eliminate dead top-level declarations uses the same machinery as above, but it operates on lists of modules. We eliminate dead top-level declarations with respect to a given main module. Fig. 16, specifies the corresponding transformation. Note that there is no need for a deep term traversal because we only deal with top-level declarations of modules.

```
1  hsFreeAndDeclared :: Term t => t -> Maybe ([HsQName],[HsQName])
2  hsFreeAndDeclared = applyTU (stop_tdTU step)
3   where
4    step = failTU `adhocTU` exp `adhocTU` pat `adhocTU` match ... `adhocTU` decls
5
6    exp (HsVar qn)             = return ([qn],[])
7    exp (HsCon qn)             = return ([qn],[])
8    exp (HsLambda pats body)   = do (pf,pd) <- hsFreeAndDeclared pats
9                                    (bf,bd) <- hsFreeAndDeclared body
10                                   return ((bf `union` pf) \\ pd,[])
11   ...
12   exp _                      = mzero  -- fail for all other expression forms
13
14   pat (HsPVar n)             = return ([],[UnQual n])
15   pat (HsPApp qn pats)       = addFree qn (hsFreeAndDeclared pats)
16   ...
17   pat _                      = mzero  -- fail for all other forms of patterns
18
19   match (HsMatch _ n pats rhs {-where-} decls)
20    = do (pf,pd) <- hsFreeAndDeclared pats
21         (rf,rd) <- hsFreeAndDeclared rhs
22         (df,dd) <- hsFreeAndDeclared decls
23         return (pf `union` (((rf \\ (dd `union` [n]) `union` df) \\ pd)), [n])
24
25   decls (ds::[HsDecl]) = do (f,d) <- hsFreeAndDeclaredGroup ds
26                             return (f \\ d,d)

-- Elaboration for groups of bindings
hsFreeAndDeclaredGroup ds = do names <- mapM hsFreeAndDeclared l
                               return ( foldr union [] (map fst names),
                                        foldr union [] (map snd names) )

-- Shorthand for adding one free name
addFree free mfd = mfd >>= \(f,d) -> return ([free] `union` f,d)
```

The analysis relies on a type-unifying traversal with *stop* (line 2). This is because we need to restart the traversal in a pattern-specific fashion (see the various recursive occurrences of `hsFreeAndDeclared`). There are type-specific cases for Haskell expressions, patterns (as in pattern-match equations), and groups of binding (i.e., lists of mutually recursive declarations). We omit a few cases that were needed for full Haskell. The equations for variables (line 6) and constructors (line 7) simply return the corresponding names as free. In the case of a lambda expression (line 8), we compute the free names from the free names of the body by subtracting the names that were declared (say bound) via the patterns. Note that there are no declared names that would escape from this scope. Other kinds of scope are illustrated in the functions `match` for pattern match equations (lines 19–23) and `decls` (lines 25–26) for groups of bindings.

**Fig. 17.** Free and declared names in Haskell program fragments.

**Name analysis** The notion of free and declared names as assumed above is essential for a broad class of language processing problems. Any analysis and transformation that deals with entities in modules and possibly nested scopes needs to be aware of scopes with their declared and free names. In Fig. 17, we define such an algorithm for Haskell. In the shown fragment, we focus on the core patterns such as lambdas, variables, nested scopes. The full algorithm deals with do-statements, list comprehensions, modules, and classes in largely the same manner. Note that generic traversal allows us to skip over many constructs that do not contribute directly to the set of free or declared names.

## 7 Related language processing setups

Let us leave the scope of functional programming (in Haskell) to compare *Strafunski* with other setups for language processing. We only discuss a few examples here while we are predominantly interested in the ways how these other approaches tackle the concerns of generic traversal and external components. This will also clarify the roots of our approach, and it will provide further evidence that functional programming in Haskell is in need of *Strafunski*'s contributions.

*The ASF+SDF Meta-Environment [1]* This is an interactive environment for the development of language processing tools. The ATerm format and the SDF formalism together with supporting tools were developed in the context of this project. A form of generic traversal has recently been added to the ASF term rewriting language, which is the central implementation language. The toolbus coordination language is offered for component integration. It is founded on process algebra, and it uses the ATerm format.

*XT [6]* This is a package for transformation tools, or more generally, for the development of language processors. The Stratego language for term rewriting with strategies plays a central role in XT's architecture. Stratego allows untyped generic programming. In fact, *Strafunski*'s support for functional strategies is largely inspired by Stratego, but realizes strategic programming in a statically typed higher-order functional programming context. XT also uses ATerms and SDF.

*Eli [3] and Cocktail [4]* These are prominent examples of attribute grammar systems. This paradigm specifically addresses language implementation, in particular semantic analysis, and translation from context-free structure to intermediate representations. Attribute grammars on their own normally fall short when applied to transformation tasks. This has been a typical application domain for rewriting technology. The aforementioned systems support integration of external components to some extent, e.g., by allowing semantic functions to be programmed in a general purpose programming language. Several non-trivial extensions of the basic attribute grammar formalism target at genericity (say, conciseness, and reusability).

*SmartTools [14]* This system supports language tool development based on two mainstream technologies, namely XML and Java. From an *abstract* syntax definition, it generates a development environment that includes a structure editor and some basic visitors that allow for generic graph traversals. SmartTools's foundation on XML makes integration of external components an easy task. If the user specifies additional syntactic sugar, a parser and a pretty printer are generated as well. In a designated simple language, the user can specify 'visitor profiles' to obtain more sophisticated visitors.

*JJForester and JJTraveler [15]* This is another architecture centred around Java. JJTraveler is basically a visitor framework including a library of reusable visitors. The specific approach provides full traversal control, basically because visitors can be combined in nearly the same way as *Strafunski*'s functional strategies. JJForester provides generative tool support to derive a Java class hierarchy from a given SDF grammar, and also the interface classes to use the JJTraveler visitor framework. Hence, JJForester corresponds to *Strafunski*'s employment of *DrIFT* for the generation of *Term* instances combined with the capabilities of the generator *Sdf2Haskell*.

## 8 The virtue of functional programming

So it is fair to say that generic traversal and external components are ubiquitous concerns in language processing. At the risk of saying the obvious, we want to argue that functional programming in Haskell has something to add when compared to other setups of language processing, that is: strong typing, higher-order functions, pattern matching, and Haskell's status of a general purpose language.

Lack of typing implies a tiresome amount of debugging when dealing with non-trivial syntaxes and formats in language processing. This is the case, for instance, for the Stratego language underlying XT [6]. Higher-orderness is basically the key to conciseness, composability, and reuse in our experience. We realise that certain readers are hard to convince but we refer to a 'benchmark' for genericity and conciseness in language processing [9]. We do not expect that the *Strafunski*-based reference solution can be outperformed by other approaches. To give an example, in a Java-based setting, one normally uses object composition, inheritance, object construction, and others to encode the combinator style of functional strategies. The merits of pattern matching in the context of language processing are obvious. The merits of a general purpose language are that the overhead for integrating external components only arises in the reuse context but not as an implication of lacking expressiveness. Also, *Strafunski* is very lightweight for this reason whereas setups that are based on attribute grammars or rewriting tend to necessitate a complete language implementation effort with all the known benefits (e.g., opportunities for designated checks and optimisations) and drawbacks (e.g., the need for a compiler, debugger, the need to deal with yet another notation, etc.).

## 9 Concluding remarks

This application letter substantiates that *typed functional programming* can be made fit to develop practical language processors in an integrated, concise, and scalable manner. To this end, we have spelled out the *Strafunski* architecture for language processing. This architecture is based on Haskell augmented with libraries and generators that provide support for *generic traversal* and the *integration of external components*. We have argued that these are the two crucial bits that are missing in plain functional programming. Generic traversal is founded on the *StrategyLib* library for *functional strategies* complemented by generative tool support. Generic traversal is essential to deal with only those language constructs that are relevant to the problem at hand. The integration of external components is supported by *Strafunski*'s ATerm library, by its connectivity to SDF parser generation, and by HaXML's support for Haskell-based XML processing.

We have applied this setup in three language processing case studies: reverse engineering of Cobol systems, computation of metrics for Java systems, and re-engineering of Haskell systems. Thus, our selection of case studies covers widely used languages from various paradigms and ages, and recurring problems of diverse algorithmic nature. The case studies clearly demonstrate that generic traversal is indispensable to achieve concise, scalable, and adaptive implementations. They also also prove that our approach to the integration of external components allows Haskell to be applied to previously alien applications: think of analysing and transforming huge Cobol portfolios.

## References


1. M. v. d. Brand et al. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Proc. of Compiler Construction 2001 (CC 2001)*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.
2. M. v. d. Brand, H. d. Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software—Practice & Experience*, 30(3):259–291, Mar. 2000.
3. R. Gray, V. Heuring, S. Levi, A. Sloane, and W. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM 35*, pages 121–131, Feb. 1992.
4. J. Grosch and H. Emmelmann. A Tool Box for Compiler Construction. In D. Hammer, editor, *Proc. of Compiler Compilers, Third International Workshop on Compiler Construction*, volume 477 of *Lecture Notes in Computer Science*, pages 106–116, Schwerin, Germany, 22–26 Oct. 1990. Springer, 1991.
5. 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.
6. M. d. Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M. v. d. Brand and D. Parigot, editors, *Proc. LDTA 2001*, volume 44 of *ENTCS*. Elsevier Science, 2001.
7. J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit–System Demonstration. In M. Brand and R. Lämmel, editors, *Proc. of LDTA'02*, volume 65 of *ENTCS*. Elsevier Science, 2002.
8. R. Lämmel. The Sketch of a Polymorphic Symphony. In B. Gramlich and S. Lucas, editors, *Proc. of International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *ENTCS*. Elsevier Science, 2002. 21 pages.
9. R. Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE 2002*, Pittsburgh, USA, 5 Oct. 2002. ACM Press. 14 pages.
10. R. Lämmel and C. Verhoef. *VS COBOL II grammar Version 1.0.3*, 1999. Available at: `http://www.cs.vu.nl/grammars/vs-cobol-ii/`.
11. R. Lämmel and J. Visser. Design Patterns for Functional Strategic Programming. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE 2002*, Pittsburgh, USA, 5 Oct. 2002. ACM Press. 14 pages.
12. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In S. Krishnamurthi and C. Ramakrishnan, editors, *Proc. of PADL 2002, Portland, OR, USA*, volume 2257 of *LNCS*. Springer-Verlag, Jan. 2002.
13. V. Maslov and C. Dodd. Btyacc—backtracking yacc, 1995-2001. `http://www.siber.org/btyacc/`.
14. D. Parigot, C. Courbis, P. Degenne, A. Fau, C. Pasquier, J. Fillon, C. Held, and I. Attali. Aspect and XML-oriented Semantic Framework Generator: SmartTools. In M. v. d. Brand and R. Lämmel, editors, *Proc. LDTA 2002*, volume 65 of *ENTCS*. Elsevier Science, 2002.
15. J. Visser. Visitor Combination and Traversal Control. *ACM SIGPLAN Notices*, 36(11):270–282, Nov. 2001. OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.
16. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? *ACM SIGPLAN Notices*, 34(9):148–159, Sept. 1999. Proceedings of ICFP'99.
17. N. Winstanley. A type-sensitive preprocessor for Haskell. In *Glasgow Workshop on Functional Programming, Ullapool*, 1997.