Scheme Program Source Code as a Semistructured Data

Kirill Lisovsky
MISA State Technological University
Moscow, Russia

Abstract

While traditional literate programming languages utilize a combination of a typesetting language and a programming language and are usually oriented towards a printed paper presentation of a program, the proposed technique combines a programming language with a semistructured markup and is intended for transformation of the Scheme source code into semistructured data that may be transformed or queried using usual semistructured data management methods and tools.

1 Introduction

As the structure of a software program may be considered as a "WEB" [8] made up of many interconnected pieces, application of semistructured data management technique looks like a natural solution.

Scheme itself is well suited for semistructured data management. Considered in this paper `Mole` [2] is an implementation of a Scheme source code management tool employing SXML [3] for its representation.

The representation of a Scheme program as a SXML tree makes it possible to use all of the SXML-based transformation and querying techniques for generation of meaningful reports from parsed Scheme sources.

If the source code contains a mark-up expressed by means of specific comments and indentation, then an additional semantic structure can be deducted and utilized for transformation of the source code into semistructured data.

2 Architecture

The proposed Scheme source code management system consists of three main components: the source code analyzer, the semistructured data repository, and the report synthesizer.

2.1 Source code analyzers

The purpose of the analyzers is to parse Scheme source code and transform it into SXML data. This task may be simplified significantly if the analyzer is aware of the "formatting style" employed in the parsed source code which is usually a set of rules for indentation and/or commenting in a Scheme program.

Current implementation of `Mole` provides a built-in source code parser but the proposed system does not depend upon any particular fixed source code formatting style. As long as the code is formatted and commented in accordance with some regulations like those described in [11], the implementation of the analyzer is not a difficult task.

Any source code which is already formatted for a Scheme literate programming tool is a good candidate for transformation into SXML semistructured data. The source code analyzer of the corresponding literal programming tool may be used as a starting point.

2.2 Semistructured data repository

`Mole` uses a SXML [3] tree as a repository for the parsed Scheme source code.

SXML is an instance of XML Infoset as S-expressions. It specifies the syntax and semantics for the representation of a well-formed XML document in the form of a Scheme S-expression. A stack of SXML based technologies, such as SXPath, is implemented in Scheme, which makes it convenient to use SXML-based transformation and querying techniques from synthesizers implemented in Scheme.

While SXML is the most natural way of semistructured data management in Scheme parsed sources may be stored in XML files, in a semistructured DBMS, and so on. For example, Scheme sources in the XML format may be stored in IBM DB2 Universal Database using DB2 XML Extender and queried using DB2's query language.

Current `Mole` simply creates a repository for one source file and uses it for generation of requested report or stores it as XML or SXML data. It may be reasonable to create a repository containing all the source code for a particular project or all available source code, or even a centralized or distributed repository accessible via the Internet with extended querying capabilities.

2.3 Synthesizers

A synthesizer generate a "report" from the source code stored in the semistructured data repository. This report may be a hypertext or printable documentation. This may be a comments-stripped source code or a source code merged with extra documentation using external tools, such as `l2t` [12] or `SLaTeX` [14], and so on.

Even trivial functionality may be really convenient. For example, the source code of SXPath library is seven times larger than its comments-, tests- and examples-stripped version.

Since this reports may be generated using querying tools, such as SXPath [7], it is possible to generate selective reports

extracting only certain parts from the information stored in the repository.

If the repository contains a complete implementation of a large library, such as SRFI-1, and only a few of its functions are necessary it may be reasonable to extract just the functions required, thus minimizing the application footprint.

Such a sophisticated generation of the source code from the repository may be used for implementation-specific code management, for creation of a distributive, for automated generation of the verifying code or exclusion/inclusion of examples, etc.

3 Data structure and source code formatting style

One of the primary design goals for `Mole`'s source code formatting style was minimization of the requirements for markup in the source file. Most of the source code structure is deducted from the indentation and the style of the comments.

At present `Mole` supports two types of the source code styles: semicolons-based, as proposed in "Good Lisp Programming Style" [11], and its own, separators-based style which introduces one additional intermediate level of source code hierarchy.

3.1 GLPS style

The Source code structure is expressed using comments marked with different number of leading semicolons, as proposed in [11]. The generated SXML data represents a three-level hierarchy `module-chapter-function`.

3.2 Separator-based style

The source code is parsed into structural elements using lines starting with certain characters as separators, as shown on Figure 1. Lines beginning with ";=" delimit chapters, with ";-" start sections. Source code structural parts separation is the sole function of such a line, and all the characters except the first two are disregarded.

A section is finished when the beginning of a new section or chapter is encountered, or it may be terminated explicitly using a line which begins with ";^", which makes it possible to mix section with units directly nested in a chapter.

Inside a section the source code is divided into "units" which may be functions, macros, binding, etc. Each unit starts with a "; " or "(" at the beginning of the line and is closed by the beginning of a new unit, section or chapter. Stand-alone blocks of comments are marked with leading ";;".

The first line after chapter or section separator is considered as its title, whereas the following commented lines constitute its description. Function name is extracted from its declaration. If a function is preceded by some commented lines then these lines are considered as its description.

3.3 Data structure

Data structure of the generated SXML tree depends on the source code formatting style and analyzer used. The primary style for current `Mole` is the separator-based one, where a generated SXML data represent a four-level hierarchy `module-chapter-section-unit`. Units may be nested in chapters directly, so the use of sections is optional.

While a `module` corresponds to source code file and `unit` is usually a function or macro definition, `chapters` and

```
;; Mole
; Transforms Scheme program to SXML document
; and generates some reports from it.

;================================================
; Analyzers
; This functions read the Scheme code, parse it to
; its structural elements, and store it in the SXML
; tree.

;------------------------------------------------
; Low-level

; Returns description of a given S-expression
; Possible types of unit: function, macro and app
(define (expr-type expr)
... code ...)

;------------------------------------------------
; Structural parts readers
; Readers for units, sections, chapters and entire
; source file. Beginning of each structural part
; in the source file is marked with the special
; comment line, which begins with:
;  ";=" - chapter
;  ";-" - section
;  "; " - unit

(define (read-scm-chapter)
... code ...)

(define (read-scm-section)
... code ...)

;================================================
; Synthesizers
; This functions generate different kinds of reports
; about the Scheme program stored in the SXML tree.

;------------------------------------------------
; Low-level

; Extract function declaration
(define (function-declaration s-expr)
... code ...)

;------------------------------------------------
; Chapter

; Chapter entry in the table of content
(define (toc-chapter c)
... code ...)

; Chapter content
(define (doc-chapter c)
... code ...)

;------------------------------------------------
; Section

; Section entry in the table of content
(define (toc-section s)
... code ...)
```

Figure 1: Outline of a Scheme source code formatted in separator-based style

```
(*TOP*
  (module
    (comment)
    (chapter
      (title)
      (description)
      (function
        (name)
        (description)
        (code))
      (section
        (title)
        (description)
        (function
          (name)
          (description)
          (code)))
        (macro
          (name)
          (description)
          (code)))
        (app
          (name)
          (description)
          (code)))))
```

Figure 2: DataGuide for the SXML tree generated by Mole

`sections` has no direct counterparts in the Scheme language. Those are defined by the means of `Mole` markup and provide semantically meaningful abstractions for larger groups of units.

Figure 2 provides the example structure of SXML tree generated by `Mole` from a source code formatted in separator-based style. It is described using strong DataGuide [6] represented in form of S-expression. DataGuide provides a descriptive schema of semistructured data and it may be used for formulation of meaningful queries.

4 Elucidative programming

Elucidative programming is a variant of literate programming which keeps the source program intact and free of lengthy documentation [10] and is based on relations between units of two documents: program source code and elucidative text.

`Mole` represents a Scheme source code as a semistructured data and provides means to address a desired structural part of the source code which makes it possible to link a unit of elucidative text to the relevant unit of the program source code. Hence, a program conforming to `Mole` source code formatting style may be elucidatively documented without any changes in the source code itself.

For example, such a system may be implemented using Mole for semistructured representation of the Scheme source code and combination of l2t [12] and SXML data management library [4] for embedding chunks of this source code in elucidative LaTeX document.

Such a system may be also implemented using XML format for elucidative document, a general purpose XML transformation tool, and DocBook [1] as a report generator.

5 Related works

The concept of literal programming has a wide recognition in the Scheme community, and a lot of literate programming tools have been developed for Scheme [12, 5, 9, 13].

Whereas most of these tools are intended for transformation of the Scheme source code into pretty-printed LaTeX or hyper-linked HTML documentation `Mole` transforms it into a SXML tree: source code parsing, parsed code querying and resulting report transformation are separated.

While most existing literate programming tools specify the layout of the resulting document in the source code, `Mole` is focused on the definition of the program structure in the source code.

Unlike any of the existing literate programming tools, `Mole` is based on general-purpose SXML transformation and querying tools. It may be used for implementation of a repository-based Scheme source code management system which is out of scope for traditional literal programming tools.

The described approach may be utilized for elucidative-style [10] documentation of the source code. `Mole` does not provide a complete elucidative programming environment comparable to Elucidator [9] but it provides a technology which enables one to attach some elucidative documentation to a Scheme source code. General purpose XML or SXML navigation and querying tools may be used for linking and embedding the structural parts of this two documents.

`Mole` itself is not focused on sophisticated source code parsing or high-quality reports generation. It is intended to be combined with external XML processors, Scheme literate programming tools, typesetting tools, and so on.

6 Conclusions

The principles and architecture of SXML-based Scheme source code management systems are described.

The main purpose of `Mole` (apart from the proof of concept) is to provide an infrastructure for development of user-defined source code parsers and report generators. However, it includes predefined analyzers and synthesizers which makes it practically useful as a generator of a customized source code and documentation without any additional programming. Different variants of `Mole` are already used, mostly for application deployment and documentation generation. `Mole` is Open Source software and available from [2].

Current implementation of `Mole` provides only a limited set of source code parsers and report generators. It does not provide any bindings for external tools, such as `l2t` [12] or `scmdoc` [5], etc., which may be used for improving the quality of generated reports and source code parsing. However some integration with external software is already possible using XML representation of `Mole` repository.

Classification of the unit types is in the preliminary state of the art, as well as the heuristic used for deducting the unit type and the name from the source code.

Application of the described approach to entire source code base management provides some functionality similar to a module system, or it may be used in conjunction with the module systems of the underlying Scheme implementations. Such a system is especially useful for deployment of portable applications and libraries.

It may be a good idea to standardize the semistructured data structure used for representation of Scheme program. Designing of an expressive and flexible data structure which may be used with a convenient variety of corresponding

source code formatting styles is a challenging problem, for which this paper provides a possible starting point.

7 Acknowledgments

Fruitful discussions with Oleg Kiselyov are gratefully acknowledged.

References

[1] DocBook.
`http://www.oasis-open.org/docbook/`.

[2] Mole.
`http://pair.com/lisovsky/scheme/lit/mole/`.

[3] SXML Specification.
`http://pobox.com/~oleg/ftp/Scheme/SXML.html`.

[4] SXMLP.
`http://pair.com/lisovsky/sxml/sxmlp/`.

[5] D. Boucher. A Scheme documentation generator.
`http://kaolin.unice.fr/~serrano/bigloo/contribs/scmdoc.tar.gz`.

[6] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. *VLDB*, 1997.

[7] O. Kiselyov. XML and Scheme.
`http://pobox.com/~oleg/ftp/Scheme/xml.html`.

[8] D. Knuth. *The CWEB System of Structure Documentation*. Addison-Wesley, 1994.

[9] K. Noermark. An Elucidative Programming Environment for scheme.
Nordic Workshop on Programming Environment Research. Limerick, Ireland., May 2000.

[10] K. Noermark. Requirements for an Elucidative Programming Environment.
International Workshop on Program Comprehension. Limerick, Ireland., June 2000.

[11] P. Norvig and K. Pitman. Tutorial on Good Lisp Programming Style. Lisp Users and Vendors Conference, August 1993.

[12] C. Queinnec. L2T: a Literate Programming tool.
`http://www-spi.lip6.fr/queinnec/WWW/l2t.html`,
January 2000.

[13] J. Ramsdell. SchemeWEB – WEB for Lisp.
`http://wuarchive.wustl.edu/packages/TeX/web/schemeweb/`.

[14] D. Sitaram. SLaTeX.
`http://www.cs.rice.edu/CS/PLT/packages/slatex/`.