Porting Scheme Programs

Dorai Sitaram
Verizon
40 Sylvan Road
Waltham, MA 02451

Abstract

The Scheme standard and the Scheme reports define not one but an entire family of programming languages. Programmers can still create useful programs in small dialect-specific extensions of the standardized Scheme language but porting such programs from one dialect to another requires tedious work. This paper presents SCMXLATE, a lightweight software tool that automates a large portion of this work.

1. ON THE PORTABILITY OF SCHEME

The existence of the IEEE Scheme Standard [6] appears to suggest that Scheme programmers can write a program once and run it everywhere. Unfortunately, appearances are deceiving. The Scheme standard and the Scheme reports [16, 15, 1, 2, 8] do not define a useful programming language for all platforms. Instead they—like the Algol 60 [9] report—define a family of programming languages that individual implementors can instantiate to a concrete programming language for a specific platform. As a result, Olin Shivers can publicly state that "Scheme is the least portable language I know" without expecting any contradictions from the authors of the standard or report documents.

Even though the Scheme standard and reports define a minimal language, it is still possible to write useful programs in small extensions of the standard language.[1] To understand the expressive power of standard Scheme plus a small library, take a look at SLaTEX [10], a package for rendering Scheme code in an Algol-like presentation style via TEX (approximately 2,600 lines of code), and TEX2page [11], a package for rendering TeX documents as HTML (approximately 9,200 lines of code).

To create a stand-alone application from a Scheme program in some different dialect of Scheme, programmers must often conduct a systematic three-stage transformation. First, they need installation-specific configuration code. Second, they add code for functions that the targeted dialect doesn't support. Finally, they must perform a number of tedious and labor-intensive surgery on the code itself.

This short paper presents SCMXLATE,[2] a program that assists programmers with the task of porting programs from one Scheme dialect to another. Specifically, the program assembles new packages from existing packages, libraries and directives. The program has been applied to a number of packages, including the above-mentioned SLaTEX and TEX2page.

The next section presents the general model of porting code. The third section describes the "surgery" directives that have proven useful for porting a number of large packages among several Scheme dialects. The last two sections discuss related and suggestions for future work.

2. PORTING PROGRAMS

SCMXLATE provides two services. First, it assists programmers with the tedium of porting Scheme programs from one dialect to another. Second, it provides the ability to configure a program into an installation-specific application.

In the first subsection, we present SCMXLATE's underlying assumptions about programs and the conversion process. In the second subsection, we illustrate an end-user's experience with SCMXLATE-based packages. In the third subsection, we describe how SCMXLATE translates a program from one Scheme dialect to another and how it assists with the creation of a full-fledged application.

2.1 Assumptions

Standard Scheme does not provide a module mechanism for partitioning a program into several components with well-specified dependencies. Instead, the Scheme standard implies that programmers treat files as components and combine them using Scheme's `load` instruction.

Since Scheme does not specify a method for describing directory paths in a platform-independent manner, SCMXLATE assumes that the programmer has placed all files into a single directory. Figure 1 displays an example. The sample program consists of three files, which are displayed in *italic*

[1] We use "Standard Scheme" for both the IEEE language and the language defined in the reports.

[2] We suggest reading the name as SCM × LATE and pronouncing it as "skim latte".

pgdir/	the package directory
apple	a source file
banana.ss	...
orange.scm	...
`dialects/`	the port directory
`files-to-be-ported.scm`	specifies the files in the parent directory that must be converted
`dialects-supported.scm`	specifies the target dialects
GUILE-APPLE	specifies dialect-specific instructions for a to-be-converted source file
GUILE-BANANA.SS	...
GUILE-ORANGE.SCM	...
SCMXLATE-APPLE	specifies installation-specific adjustments for a to-be-converted source file
SCMXLATE-BANANA.SS	...
SCMXLATE-ORANGE.SCM	...
my-apple	a generated target file
my-banana.ss	...
my-orange.scm	...

Figure 1: A sample directory organization

font. [Note to readers: please ignore the rest of the figure for now.]

A program is not an application. To create an application from a program, the installer must often specify some values that depend on the context in which the program runs. For example, a spelling program may need to know about some idiosyncratic words for a specific user. While an interactive approach works well for a spelling program, it is a terrible idea for a Unix-style filter, which transforms a text file in one format into another one. For such programs, it is best if users conduct a configuration process that creates the installation-specific defaults. SCMXLATE assumes that this configuration step should be a part of the installation and port process and therefore supports it in a minimal manner, too.

2.2 Installing a Package with SCMXLATE

Assume that a programmer has prepared some package for use on several Scheme dialects and possibly different platforms. Also imagine an installer who wishes to install the package for a Scheme dialect that is different from the source language and for a new platform. This installer must take two steps.

First, the user must install SCMXLATE on the target platform. Second, the user must configure the actual package. To do so, the user launches the target Scheme implementation in the package directory and types

```
(load "/usr/local/lib/scmxlate/scmxlate.scm")
```

where the `load` argument uses the full pathname for the directory that contains SCMXLATE. As it is loaded, SCMXLATE poses a few questions with a choice of possible answers, including a question that determines the target dialect,[3] though a knowledgeable user can provide different answers.

[3]The Scheme standard and reports do not provide a generic mechanism for Scheme programs to determine in which dialect they run.

When all the questions are answered, SCMXLATE creates the platform-specific and dialect-specific package. Naturally, the programmer can also prepare versions of a package for various dialects directly.

2.3 Preparing a Package for SCMXLATE

A programmer who wishes to distribute a package for use with different Scheme dialects creates a sub-directory with several files in the package directory. The files specify the pieces of the package that require translation, the dialects that are supported, and optional dialect-specific preambles for each file that is to be translated.

If the package also requires installation-specific configuration instructions, the programmer supplies files in the package directory. Specifically, the programmer creates one file per source file that requires special configurations. These additional files are independent of the target dialect but may contain SCMXLATE rewriting directives that process the corresponding source file (see the next section).

Let us refine our example from figure 1. Assume the source language is MzScheme and the file *apple* uses the library function

```
file-or-directory-modify-seconds
```

Also assume that the target language is Guile. Then the dialect-specific transformation file for *apple*—GUILE-APPLE in the figure—should contain the following Guile definition:

```
(define file-or-directory-modify-seconds
  (lambda (f)
    (vector-ref (stat f) 9)))
```

If the dialect-configuration file supplies a definition for a name that is also defined in the input file, then the output file contains the definition from the dialect-configuration file, not the input file. For example, suppose *apple* contains the

MzScheme definition

```
(define file-newer?
  (lambda (f1 f2)
    ;checks if f1 is newer than f2
    (> (file-or-directory-modify-seconds f1)
       (file-or-directory-modify-seconds f2))))
```

In Guile, this definition is expressed as

```
(define file-newer?
  (lambda (f1 f2)
    ;checks if f1 is newer than f2
    (> (vector-ref (stat f1) 9)
       (vector-ref (stat f2) 9))))
```

and this definition is therefore placed into GUILE-APPLE. Then SCMXLATE's translation of *apple* directly incorporates the Guile definition into the output file. That is, SCMXLATE doesn't even attempt to translate the MzScheme definition of the same name in the input file.

Let us revisit figure 1. In addition to the source files, the figure displays the complete directory structure for a specific example. SCMXLATE inspects the file and directory names in `type-writer` font for instructions on how to translate the source files in *pgdir*. In particular,

`files-to-be-ported.scm` contains strings that specify the names for those files that SCMXLATE must translate;

`dialects-supported.scm` contains symbols, which specify the names of the dialects for which the programmer has prepared translations; currently, SCMXLATE supports

BIGLOO,	PETITE,
CHEZ,	PSCHEME,
CL,	SCHEME48,
GAMBIT,	SCM,
GAUCHE,	SXM,
GUILE,	SCSH,
KAWA,	STK,
MITSCHEME,	STKLOS, and
MZSCHEME,	UMBSCHEME.

To provide file-specific adaptation code per dialect, the programmer creates a file name with a dialect-indicating prefix; in figure 1 these files are displayed in SMALL-CAPS font. Finally, installation-specific configuration code is in files whose names are prefixed with SCMXLATE-. All SMALL-CAP files are optional.

When SCMXLATE is run in the *pgdir* directory, it creates one file per source file. In figure 1, these files appear in *underlined italic* font. Figure 2 shows the structure of these generated files. The installation-specific code appears at the very top of the file; it is followed by the dialect-specific code. The bottom part of the file consists of the translated source code. The translation process is specified via directives that comes with the installation-specific and dialect-specific pieces.

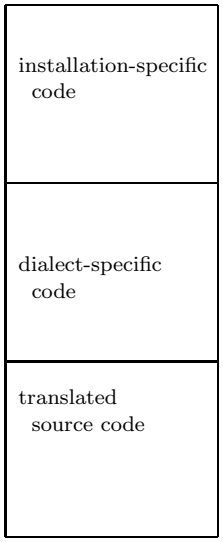

Figure 2: The file structure

3. THE DIRECTIVES

In addition to Scheme code intended to either augment or override code in the input file, the dialect-configuration and installation-configuration files can use a small set of directives to finely control the text that goes into the output file, and even specify actions that go beyond the mere creation of the output file. These directives are now described.

3.1 `scmxlate-insert`

As we saw, Scheme code in the dialect-configuration and installation-configuration files is transferred verbatim to the output file. Sometimes, we need to put into the output file arbitrary text that is not Scheme code. For instance, we may want the output file to start with a "shell magic" line, so that it can be used as a shell script. Such text can be written using the `scmxlate-insert` directive, which evaluates its subforms in Scheme and displays them on the output file.

Thus, if the following directive occurs at the top of GUILE-APPLE

```
(scmxlate-insert
  "#!/bin/sh
exec guile -s $0 \"$@\"
!#
")
```

the output file `my-apple` for the Guile-specific version of the package starts with the line

```
#!/bin/sh
exec guile -s $0 "$@"
!#
```

Note that the order of the code and `scmxlate-insert` text in the configuration file is preserved in the output file.

3.2 scmxlate-postamble

Typically, the original Scheme code (augmented with the code of `scmxlate-insert`s) occurs in the output file before the translated counterpart of input file's contents, and thus may be considered as *preamble* text. Sometimes we need to add *postamble* text, ie, things that go *after* the code from the input file. In order to do this, place the directive

```
(scmxlate-postamble)
```

after any preamble text in the dialect-configuration file. Everything following that line, whether ordinary Scheme code or `scmxlate-insert`s, shows up in the output file after the translated contents of the input file.

3.3 scmxlate-postprocess

One can also specify actions that need to performed after the output file has been written. Say we want the Guile output file for *apple* to be named *pear* rather than *my-apple*. We can enclose Scheme code for achieving this inside the SCMXLATE directive `scmxlate-postprocess`:

```
(scmxlate-postprocess
  (rename-file "my-apple" "pear"))
```

3.4 scmxlate-ignore-define

Sometimes the input file has a definition that the target dialect does not need, either because the target dialect already has it as a primitive, or because we wish to completely rewrite input code that uses that definition. That is, if the target dialect is MzScheme, which already contains `reverse!`, any definition of `reverse!` in the input file can be ignored.

```
(scmxlate-ignore-define reverse!)
```

The `scmxlate-ignore-define` form consumes any number of names, and all corresponding definitions are ignored.

3.5 scmxlate-rename

Sometimes we want to rename certain identifiers from the input file. One possible motivation is that these identifiers name nonstandard primitives that are provided under a different name in the target dialect. For instance, the MzScheme functions

```
current-directory ; -> String
file-or-directory-modify-seconds ; String -> Number
```

are equivalent to the Bigloo functions

```
chdir ; -> String
file-modification-time ; String -> Number
```

respectively. So if the MzScheme input file uses these functions, the Bigloo dialect-configuration file should contain

```
(scmxlate-rename
 (current-directory
  chdir)
 (file-or-directory-modify-seconds
  file-modification-time))
```

Note the syntax: `scmxlate-rename` has any number of twosomes as arguments. The left item is the name in the input file, and the right item is its proposed replacement.

3.6 scmxlate-rename-define

Sometimes the input file includes a definition for an operator that the target dialect already has as a primitive, but with a different name. That is, consider an input file that contains a definition for `nreverse`. MzScheme has the same operator but with name `reverse!`, which means that the MzScheme dialect-configuration file should contain the following directive:

```
(scmxlate-rename-define
  (nreverse reverse!))
```

Note that this is shorthand for

```
(scmxlate-ignore-define nreverse)
(scmxlate-rename
  (nreverse reverse!))
```

3.7 scmxlate-prefix

Another motivation for `scmxlate-rename` is to avoid polluting namespace. We may wish to have short names in the input file, but when we configure it, we want longer, "qualified" names. It is possible to use `scmxlate-rename` for this action, but the `scmxlate-prefix` is convenient when the newer names are all uniformly formed by adding a prefix.

Thus,

```
(scmxlate-prefix
  "regexp::"
  match
  substitute
  substitute-all)
```

renames

`match` to `regexp::match`,

`substitute` to `regexp::substitute`,

and

`substitute-all` to `regexp::substitute-all`, respectively.

The first argument of `scmxlate-prefix` is the string form of the prefix; the remaining arguments are the identifiers that should be renamed.

3.8 scmxlate-cond

Sometimes we want parts of the dialect-configuration file to be processed only when some condition holds. For instance, we can use the following `cond`-like conditional in a dialect-configuration file for MzScheme to write out a shell-magic line appropriate to the operating system:

```
(scmxlate-cond
  ((eqv? (system-type) 'unix)
   (scmxlate-insert *unix-shell-magic-line*))
  ((eqv? (system-type) 'windows)
   (scmxlate-insert *windows-shell-magic-line*)))
```

In this expression, the identifiers `*unix-shell-magic-line*` and `*windows-shell-magic-line*` must denote appropriate strings.

Note that while `scmxlate-cond` allows the `else` keyword for its final clause, it does not support the `=>` keyword of standard Scheme's `cond`.

3.9 `scmxlate-eval`

The test argument of `scmxlate-cond` and all the arguments of `scmxlate-insert` are evaluated in the Scheme global environment when SCMXLATE is running. This environment can be enhanced via `scmxlate-eval`. Thus, if we had

```
(scmxlate-eval
  (define *unix-shell-magic-line* <...>)
  (define *windows-shell-magic-line* <...>))
```

where the `<...>` stand for code that constructs appropriate strings, then we could use the two variables as arguments to `scmxlate-insert` in the above example for `scmxlate-cond`.

A `scmxlate-eval` expression can have any number of subexpressions. It evaluates all of them in the given order.

3.10 `scmxlate-compile`

`scmxlate-compile` can be used to tell if the output file is to be compiled. Typical usage is

```
(scmxlate-compile #t)      ;or
(scmxlate-compile #f)
```

The first forces compilation but only if the dialect supports it, and the second disables compilation even if the dialect supports it. The argument of `scmxlate-compile` can be any expression, which is evaluated only for its boolean significance.

Without a `scmxlate-compile` setting, SCMXLATE asks the user explicitly for advice, but only if the dialect supports compilation.

3.11 `scmxlate-include`

It is often convenient to keep some of the text that should go into a dialect-configuration file in a separate file. Some definitions may naturally be already written down somewhere else, or we may want the text to be shared across several dialect-configuration files (for different dialects). The call

```
(scmxlate-include "filename")
```

inserts the content of `"filename"` into the file.

3.12 `scmxlate-uncall`

It is sometimes necessary to skip a top-level call when translating an input file. For instance, the input file may be used as a script file whose scriptural action consists in calling a procedure called `main`. The target dialect may not allow the output file to be a script, so the user may prefer to load the output file into Scheme as a library and make other arrangements to invoke its functionality. To disable the call to `main` in the output file, add

```
(scmxlate-uncall main)
```

to the configuration file.

The `scmxlate-uncall` form consumes any number of symbol arguments. All top-level calls to these functions are disabled in the output.

4. RELATED WORK

SCMXLATE wouldn't be necessary if standard Scheme were a practical language. One way to achieve practicality is to equip a language with powerful, expressive libraries and extensions. Jaffer's SLIB [7] effort and the SRFI process [13] aim to supplement Scheme in just such a way. If they are successful, the various Scheme dialects will resemble each other as far as the source language itself is concerned, thus rendering a good part of SCMXLATE obsolete.

From a reasonably abstract perspective, SCMXLATE provides those services to Scheme that `autoconf` [5] provides to C. Both use preprocessing to conduct tests on the code, to assist the target compiler, and to create proper contexts for the ported program. Naturally, `autoconf` is a more expressive and more encompassing tool than SCMXLATE; it has been around for twice as long.

5. SUMMARY

The paper explains how SCMXLATE assists programmers with the translation of Scheme programs from one dialect to another. The software tool evolved due to the demand to translate various packages into a number of different dialects. It is now easy to use and robust. Indeed, SCMXLATE can now also translate Scheme programs into Common Lisp programs [14] though the resulting code is somewhat unnatural.

Although SCMXLATE has become an accessible product of its own, it still lacks good environmental support. A programmer preparing a package for SCMXLATE could make good use of sophisticated syntax coloring tools such as those provided in DrScheme [3] and refactoring tools such as Dr. Jones [4], especially if they are integrated with the programming environment.

SCMXLATE is available on the Web. For more information, the interested reader should consult the on-line manual [12].

6. ACKNOWLEDGMENT

I thank Matthias Felleisen for helpful discussions and for shaping the goals of SCMXLATE.

7. REFERENCES

[1] Clinger, W. The revised revised report on the algorithmic language Scheme. Joint technical report, Indiana University and MIT, 1985.

[2] Clinger, W. and J. Rees. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[3] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme.

Journal of Functional Programming, 12(2):159–182, 2002.

[4] Foltz, M. A. Dr. Jones: A software design explorer's crystal ball. Technical report, MIT AI Lab, expected in 2003.

[5] GNU. Autoconf, `http://www.gnu.org/software/autoconf/` 1998–2003.

[6] Haynes, C. Standard for the Scheme programming language. *IEEE Document P1178/D5*, October 1991.

[7] Jaffer, A. SLIB, `http://www.swiss.ai.mit.edu/~jaffer/SLIB.html` 1995–2003.

[8] Kelsey, R., W. Clinger and J. Rees (Editors). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[9] Naur, P. E. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[10] Sitaram, D. How to use SLaTEX, `http://www.ccs.neu.edu/home/dorai/slatex/slatxdoc.html` 1990.

[11] Sitaram, D. TEX2page, `http://www.ccs.neu.edu/home/dorai/tex2page/tex2page-doc.html` 2000.

[12] Sitaram, D. scmxlate, `http://www.ccs.neu.edu/home/dorai/scmxlate/scmxlate.html` 2000.

[13] Sperber, M., D. Rush, F. Solsona, S. Krishnamurthi and D. Mason. Scheme requests for implementation, `http://srfi.schemers.org/` 1998–2003.

[14] Steele Jr., G. L. *Common Lisp—The Language*. Digital Press, 1984.

[15] Steele Jr., G. L. and G. L. Sussman. The revised report on scheme, a dialect of lisp. Technical Report 452, MIT Artificial Intelligence Laboratory, 1978.

[16] Sussman, G. L. and G. L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Technical Report 349, MIT Artificial Intelligence Laboratory, 1975.