Distributing Applications with Petite Chez Scheme

R. Kent Dybvig

Cadence Research Systems
dyb@scheme.com

November 1998

Introduction

Petite Chez Scheme is a fast, reliable, and full-featured implementation of the Scheme programming language. It is fully compatible with the complete *Chez Scheme* system but uses a high-speed threaded interpreter in place of *Chez Scheme*'s incremental native-code compiler. Programs written for *Chez Scheme* run unchanged in *Petite Chez Scheme* as long as they do not depend specifically on the compiler. In fact, *Petite Chez Scheme* is built from the same sources as *Chez Scheme*, with all but the compiler sources included. *Petite Chez Scheme* may be used and redistributed without license fee or royalty for any purpose, including for resale as part of a commercial product. For details, see the *Petite Chez Scheme Software License Agreement*, a copy of which is included as an appendix to this note.

Although suitable for use as a stand-alone Scheme system, *Petite Chez Scheme* was conceived as a run-time system for compiled *Chez Scheme* applications. This note describes how to create and distribute such applications. The following section briefly describes the characteristics of *Petite Chez Scheme* and how it compares with *Chez Scheme*. The remaining sections detail how to prepare application source code, how to build and run applications, and how to distribute them.

Characteristics of Petite Chez Scheme

Although interpreter-based, *Petite Chez Scheme* evaluates Scheme source code faster than might be expected. Some of the reasons for this are listed below.

- The run-time system is fully compiled, so library implementations of primitives ranging from `+` and `car` to `sort` and `printf` are just as efficient as in *Chez Scheme*, although they cannot be open-coded as in code compiled by *Chez Scheme* at high levels of optimization.
- The interpreter is itself a compiled Scheme application. Because it is written in Scheme, it directly benefits from various characteristics of Scheme that would have to be dealt with explicitly and with additional overhead in most other languages, including proper treatment of tail calls, first-class procedures, automatic storage management, and continuations.
- The interpreter employs a preprocessor that performs various optimizations and converts the code into a form that can be interpreted efficiently. In fact, the preprocessor performs many of the same source-level optimizations as *Chez Scheme*'s compiler.

Nevertheless, compiled code is still far more efficient for most applications. The difference between the speed of interpreted and compiled code varies significantly from one application to another, but can amount to a factor of ten or more.

Two additional limitations result directly from the fact that *Petite Chez Scheme* does not include the compiler. First, since `foreign-procedure` forms result in the generation of machine code that is tailored to a specific set of argument and return value types, `foreign-procedure` forms cannot be processed by the interpreter. Compiled versions of `foreign-procedure` forms may be included in compiled code loaded into *Petite Chez Scheme*, however. Second, since inspector information is attached to code objects generated only by the compiler, source information and variable names are not available for interpreted procedures or continuations into interpreted procedures. This makes the inspector less effective for debugging interpreted code than it is for debugging compiled code.

Except as noted above, *Petite Chez Scheme* does not restrict what programs can do, and like *Chez Scheme*, it places essentially no limits on the size of programs or the memory images they create.

Preparing Application Code

While it is possible to distribute applications in source-code form, i.e., as a set of Scheme source files to be loaded into *Petite Chez Scheme* by the end user, distributing compiled code has two major advantages over distributing source code. First, compiled code is usually much more efficient, as discussed in the preceding section, and second, compiled code is in binary form and thus provides more protection for proprietary application code. For these reasons, we suggest that applications be compiled.

Application source code generally consists of a set of Scheme source files possibly augmented by foreign code developed specifically for the application and packaged up in shared libraries (also known as shared objects or, on Windows, dynamic link libraries). The following assumes that any shared library source code has been converted into object form; how to do this varies by platform. The result is a set of one or more shared libraries that are loaded explicitly by the Scheme source code during program initialization.

Once the shared libraries have been created, the next step is to compile the Scheme source files into a set of Scheme object files. Doing so typically involves simply invoking `compile-file` on each source (".ss") file to produce the corresponding object (".so") file. This may be done within a build script or "make" file via a command line such as the following:

```
echo '(compile-file "filename")' | scheme
```

which produces the object file `filename.so` from the source file `filename.ss`.

It may be necessary to make some adjustments to a file to be compiled if the file contains expressions or definitions that affect the compilation of subsequent forms in the file. One way to do this is to remove the forms that affect compilation and place them into a separate file that is loaded prior to compilation. Another way is to use `eval-when`, which is discussed in detail in the *Chez Scheme User's Guide*. It may also be that one file defines a set of syntactic abstractions or modules that must be present during the compilation of another file. In this case, the former file must be compiled first in the same session as the latter file, or "visited" via `visit` before the second file is compiled. Again, see the *Chez Scheme User's Guide* for details.

Although it is possible to intersperse initialization expressions and definitions at the top level of a Scheme source file, we suggest that initialization expressions be encapsulated in one or more initialization procedures that are explicitly invoked when the application is created or run. Initialization procedures to be invoked when the application is created, i.e., when the heap is created as described in the next section, may be invoked by placing explicit calls to them in the last file loaded into the system. Initialization procedures to be invoked when the application is run may be invoked by the Scheme startup procedure, which is described below.

The Scheme startup procedure determines what the system does when it is started from a saved heap. The default startup procedure loads the files listed on the command line (via `load`) and starts up a new café. The startup procedure may be changed via the parameter `scheme-start`. The following example demonstrates the installation of a variant of the default startup procedure that prints the name of each file before loading it.

```
(scheme-start
  (lambda fns
    (for-each
      (lambda (fn)
        (printf "loading ~a ..." fn)
        (load fn)
        (printf "~%"))
      fns)
    (new-cafe)))
```

A typical application startup procedure would first invoke the application's initialization procedure(s) and then start the application itself:

```
(scheme-start
  (lambda fns
    (initialize-application)
    (start-application fns)))
```

Any shared libraries that must be present during the running of an application must be loaded during initialization, whether they are loaded when the application is built or not. In addition, all foreign procedure expressions must be executed or reexecuted after the shared libraries are loaded so that the addresses of foreign routines are correctly recorded with the resulting foreign procedures. The following demonstrates one way in which initialization might be accomplished for an application that links to a foreign procedure `show_state` in the shared library `state.so`:

```
(define show-state)

(define app-init
  (lambda ()
    (load-shared-object "state.so")
    (set! show-state
      (foreign-procedure "show_state" (integer-32)
        integer-32))))

(scheme-start
  (lambda fns
    (app-init)
    (app-run fns)))
```

Building and Running the Application

Building and running an application is straightforward once all shared libraries have been built and Scheme source files have been compiled to object code.

Although not strictly necessary, we suggest that you concatenate your Scheme object files into a single object file. This may be done on Unix systems simply via the "`cat`" program or on Windows via `copy`. Placing all of the object code into a single file simplifies both building and distribution of applications.

With the Scheme object code contained within a single "boot" file, it is possible to run the application simply by loading the boot file into *Petite Chez Scheme*, e.g.:

```
petite app.boot
```

where `app.boot` is the name of the application boot file, and invoking the startup procedure:

```
> ((scheme-start))
```

It is usually preferable, however, to construct a *Petite Chez Scheme* heap that contains the application. The heap should be built on the system upon which the application will ultimately be run, i.e., on the end-user's computer system. The following command line constructs a heap file `app.heap` from an application boot file `app.boot`:

```
echo "(exit)" | petite app.boot -s1 app.heap
```

When using the Windows command shell, it may be necessary to omit the double quotes around the call to `exit`. Once the heap has been built, the application may be run simply by invoking *Petite Chez Scheme* as follows:

```
petite -h app.heap
```

The details of application start-up can be hidden from the end user via a shell script or batch file. A Unix shell script to invoke an application might look like:

```
# APPLIB is set during installation to reflect location of app heap file
APPLIB=/usr/local/lib/app
# export APPLIB so that the application can find other application
# library files
export APPLIB
petite -h $APPLIB/app.heap -- $*
```

The command-line argument "`--`" tells *Petite Chez Scheme* to pass along all remaining arguments as uninterpreted strings to the Scheme startup procedure.

Distributing the Application

Distributing an application involves creating a distribution package that includes, at a minimum, the following items:

- the *Petite Chez Scheme* distribution,
- the application boot file,
- any application-specific shared libraries,
- the application script or batch file (if not built from scratch by the installation script), and
- an application installation script.

The application installation script should allow the installation of *Petite Chez Scheme* if not already installed on the target system. It should create a directory to contain the application heap file, and create the heap file in that directory by invoking *Petite Chez Scheme* with the boot file and using the "-s1" option as shown in the preceding section. It should also install the application shared libraries, if any, either in the same location or in a standard location for shared libraries on the target system. Finally, it should build or edit the application script or batch file to reflect the location of the heap file and install the script or batch file on the target system. A sample installation script for Unix platforms is provided as an appendix to this note. For Windows, we suggest the use of an installation building program, such as *Wise Installation System* or *InstallShield*.

Contact us if you do not have a copy of the *Petite Chez Scheme* distribution or if the distribution you received combines both *Chez Scheme* and *Petite Chez Scheme*. Although *Petite Chez Scheme* is freely redistributable, the complete *Chez Scheme* system may be used only under direct license from Cadence Research Systems and may not be redistributed.

Appendix: Sample Unix Installation Script

The script below demonstrates how to perform a straightforward installation of a Scheme application on a Unix-based platform. The script makes the following assumptions, any of which may be changed by altering the script's application configuration parameters:

- the name of the application to install is `app`,
- the machine type upon which the installation will take place is `sps2` (Sparc Solaris 2.X),
- a single shared library, `libapp.so`, is included in the distribution, and
- a single boot file, `app.boot`, is included in the distribution.

The script also sets the default location for executables to `/usr/local/bin`, shared libraries to `/usr/local/lib`, and the application heap to `/usr/local/lib/app`. These settings would typically be open to change by the end user; a friendlier script would query the user to verify that these settings are appropriate.

The script first installs *Petite Chez Scheme*, then installs the shared libraries, then builds the heap and the executable shell script.

```
# set installation directories
bindir=/usr/local/bin
libdir=/usr/local/lib
applibdir=/usr/local/lib/${app}

# set application configuration
app=app
machine=sps2
libs=lib${app}.so
boot=${app}.boot
heap=${applibdir}/${app}.heap

# install Petite Chez Scheme
(cd csv6.0/custom; make petiteinstall m=${machine})

# install the shared libraries
(umask 022; mkdir -p ${libdir})
cp ${libs} ${libdir}
chmod 444 ${libs}
```

```
# create the application heap
(umask 022; mkdir -p ${applibdir})
echo | petite ${boot} -s1 ${heap}
chmod 444 ${heap}

# create the executable shell script
(umask 022; mkdir -p ${bindir})
echo '#! /bin/sh' > ${bindir}/${app}
echo 'exec petite -h' ${heap} '-- $*' >> ${bindir}/${app}
chmod 555 ${bindir}/${app}
```

Appendix: Petite Chez Scheme Software License Agreement

Licensee agrees that Cadence shall not be held to any liability with respect to any claim by Licensee or a third party arising from or on account of the use of the Licensed Product, regardless of the form of action, whether in contract or tort, including negligence. IN NO EVENT WILL CADENCE BE LIABLE FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES OF ANY NATURE WHATSOEVER, EVEN IF CADENCE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES IN ADVANCE.

6. US Government Customers

The Licensed Product is classified as "commercial computer software" developed at private expense. If delivered to the Department of Defense, the Licensed Product is delivered subject to the terms of this Agreement and either (i) in accordance with DFARS 227-7202-1(a) and 227.7202-3(a), or (ii) with restricted rights in accordance with DFARS 252.227-7013(c)(1)(ii) (Oct. 1988), as applicable. If delivered to any other Federal agency, the Licensed Product is restricted computer software delivered subject to the terms of this license agreement and (i) FAR 12.212(a), (ii) FAR 52.227-19, or (iii) FAR 52.227-14 (ALT III), as applicable.

7. Term and Termination

The effective date of this Agreement shall be the date of the first installation of the Licensed Product by Licensee, and its term is perpetual, except that Cadence may terminate this Agreement if Licensee fails to comply with any of the terms and conditions of this Agreement. Upon termination, Licensee shall cease use and redistribution of the Licensed Product and shall destroy or return to Cadence all copies of the Licensed Product. Licensee's obligations under Paragraph 5 shall survive any termination of this Agreement.

8. Governing Law

This Agreement shall be governed by the laws of the United States of America and the State of Indiana, both as to interpretation and performance. It constitutes the complete and exclusive statement of the Agreement between the parties with respect to the Licensed Product and supersedes all previous understandings, commitments or agreements, oral or written. The provisions of this Agreement are severable, and in the event that any provisions of this Agreement are determined to be invalid or unenforceable under any controlling body of law, such invalidity or unenforceability shall not in any way affect the validity or enforceability of the remaining provisions hereof. This Agreement may be modified only by a written agreement executed by both Cadence and Licensee.

BY INSTALLING THIS SOFTWARE, YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, THAT YOU UNDERSTAND IT, AND THAT YOU AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS.