ZK Installation Guide

Sentot Kromodimoeljo

July 2021

0.1 Prerequisites

0.1.1 Building from Source Code

If a pre-built ZK executable is not available for your platform, you will need to build it from source code and need one of the following ANSI Common Lisp implementations:

- Steel Bank Common Lisp (SBCL),
- Clozure Common Lisp (CCL), or
- Embeddable Common Lisp (ECL).

We recommend using a 64-bit Lisp implementation.

We have successfully compiled ZK using SBCL version 2.1.6, CCL version 1.12, and ECL version 21.2.1, on a notebook running Ubuntu 20.04.2 LTS. The ZK system built using SBCL is about twice as fast as that built using CCL, which is itself much faster than that built using ECL.

We have also successfully compiled ZK using SBCL version 2.1.6 on a macBook running macOS Big Sur 11.4 and using SBCL version 2.0.0 on a notebook running Windows 10 Pro 20H2. While the ZK system on the macBook was slightly faster than that running on the Ubuntu notebook, the ZK system running on the Windows notebook was significantly slower. All machines were running 64-bit AMD/Intel processors with similar specifications (3rd generation Intel I7 Quad Cores).

You ought to be able to build ZK for any major non-handheld OS platform, as SBCL is available for most OS and hardware architecture combinations. For Android and iOS, cross-compiling using ECL might be a solution, but we have not tried it and it will not be covered by this guide.

0.2 Directory Structure

Under the root directory of a ZK distribution, there are 6 subdirectories:

- `bin` subdirectory for pre-built ZK executables,
- `doc` subdirectory for available documentation,
- `examples` subdirectory for ZK examples,
- `library` subdirectory for an example ZK library,
- `sources` subdirectory for ZK source code including 3rd party code, and
- `system` subdirectory for build files and an Emacs ZK-mode .el file.

The main object files generated from compiling ZK are stored in subdirectories of `sources`, in `sources/SBCL`, `sources/CCL` or `sources/ECL` depending on the Lisp used.

Source files for 3rd party code are also stored in subdirectories of `sources`, e.g. `sources/chipz_0.8` for `chipz_0.8`. Object files generated from compiling `chipz_0.8` are stored in `sources/chipz_0.8/SBCL`, `sources/chipz_0.8/CCL` or `sources/chipz_0.8/ECL` depending on the Lisp used.

0.3 Compiling ZK

At a command line, e.g., in a bash terminal, go to the `system` subdirectory:

```
> cd <rootdir>/system
```

where `<rootdir>` is the root directory of your ZK distribution.

First compile the lexicon. You need to compile it separately because the Lisp package mechanism is a bit problematic. Invoke Lisp, for example:

```
> sbcl
```

(replace `sbcl` with e.g., `ccl64` or `ecl` when using another Lisp). At the Lisp prompt, load `compile-lexicon.lisp`:

```
* (load "compile-lexicon")
```

which compiles `sources/lexicon.lisp` and places the resulting object file in `sources/SBCL`. Once the compile is finished, exit Lisp:

```
* (quit)
```

You are now ready to compile the rest of the ZK system, including 3rd party libraries for flexible streams and compression/decompression. Invoke Lisp again:

```
> sbcl
```

Load `compile-all.lisp`:

```
* (load "compile-all")
```

It shouldn't take long for the compile to finish. The object files generated will be placed in the appropriate subdirectories. You are all done compiling and can exit Lisp:

```
* (quit)
```

0.4 Trying Out Compiled ZK

When you are done compiling ZK, you might want to try it out before making an executable ZK file. Again at the `system` subdirectory invoke Lisp:

```
> sbcl
```

Load `load-all.lisp`:

```
* (load "load-all")
```

When everything is loaded, you can run a regression test on the examples:

```
* (zk:run-all-tests)
```

The test took 1 minute and 17 seconds on a circa end-of-2013 macBook Pro. If everything is well ZK will print a bunch of `oks`. Otherwise ZK will report on differences from a previous run.

You are now ready to try out the ZK command line interface:

```
* (zk:zk-mode)
```

ZK will print out a banner with copyright notices, with the last few line looking as follows, ending with the `>` prompt:

```
ZK command line interface.
Type "(quit)" to exit.
Type "(help)" for help.

>
```

You can now try various ZK declarations and commands. The on-line help includes descriptions of the declarations and commands. When you are done you can exit the ZK command line interface by invoking the `QUIT` command:

```
> (quit)
```

ZK will ask you to confirm:

```
Exit ZK command processor?  (yes or no)
```

to which you will answer `yes`. Since you are running ZK not as a stand-alone executable, you will end up at the Lisp prompt, from which you can exit using quit:

```
* (quit)
```

0.5 Creating an Executable ZK

When you are satisfied that ZK is correctly compiled, you can create an executable file for ZK so you do not interact at the Lisp level at all. At the `system` subdirectory, start Lisp:

```
> sbcl --dynamic-space-size 16384
```

The dynamic space size option for SBCL is to make it less likely to run out of space when working on very large problems with proof logging turned on. This makes SBCL run with a virtual memory size of 16 GB instead of the default of approximately 1.2 GB. CCL, on the other hand, seems to run with a virtual memory size of 512 GB.

Load ZK:

```
* (load "load-all")
```

Save ZK as an executable. With SBCL you invoke the following:

```
* (save-lisp-and-die "zk" :toplevel #'zk:zk-mode :executable t
          :save-runtime-options t)
```

This will create an executable called `zk`. Under Windows, you would specify `"zk.exe"` rather than `"zk"`. If you compiled with CCL then you will invoke the following instead:

```
? (save-application "zk" :toplevel-function #'zk:zk-mode
          :prepend-kernel t)
```

Creating a ZK executable with ECL is not easy and has not been tried. ECL does not provide a way of saving a Lisp image. In a nutshell, you must recompile all source files using the "`:system-p t`" option and then use `c:build-program` to create an executable from the generated object files.

0.6 Installing ZK

Once you have an executable, either by building it from source code or having a pre-built executable available for your platform, you can properly install it. A good place to put a ZK executable on a unix system would be `/usr/local/bin`. In any case, you would want to have the directory where you placed the ZK executable to be in your `PATH` environment variable.

If you want ZK mode for the Emacs editor, then copy `zk.el` from the `system` subdirectory to your Emacs Lisp directory (e.g., `/home/user/.emacs.d/lisp`). Assuming you installed ZK in `/usr/local/bin` and your Emacs Lisp directory is `/home/user/.emacs.d/lisp` then add the following code snippet to your `.emacs` file:

```
(add-to-list 'load-path "/home/user/.emacs.d/lisp/")
(setq auto-mode-alist
      (append auto-mode-alist '(("\\.ver$" . zk-mode))))
(autoload 'zk-mode "zk" "ZK mode." t)
(autoload 'run-zk "zk" "Run ZK." t)
(setq zk-program "/usr/local/bin/zk")
```

0.7 ZK mode in Emacs

After you have successfully installed ZK and the ZK mode for Emacs, you can start developing ZK theories using Emacs as your combined editor/interface. When you edit a file with a `.ver` extension, the Emacs buffer for the file is automatically set to ZK mode. In addition to performing edits, you can send ZK declarations and commands in the file buffer to a ZK process running in a special `comint` buffer called `*zk*`.

The first time you edit a `.ver` file in an Emacs session, ZK will not be automatically run, so the first thing you should do is to start ZK running by typing `"C-c C-l"` (control-c followed by control-l). This will start ZK in the buffer `*zk*`.

In the file buffer, the main additions to standard editting is moving forward and backward by declaration or command, sending a declaration or command to ZK, sending a sequence of declarations and commands in a region to ZK, and inserting current proof into the file buffer. The relevant key bindings are:

- `"C-c C-f"` - move forward one declaration or command.
- `"C-c C-b"` - move back one declaration or command.
- `"C-c C-e"` - send declaration or command to ZK.
- `"C-c C-r"` - send declarations and commands in marked region to ZK.
- `"C-c C-s"` - insert current proof.

You can also interact directly with ZK in the `*zk*` buffer. Menu-based completions provided by Emacs `comint` are also available through the following key bindings:

- `"C-i"` or `"Tab"` - completion based on ZK keywords.
- `"C-c C-n"` - file name completion.

0.8 Getting Started with Using ZK

Although you don't need to use Emacs to use ZK, it is convenient to do ZK theory development using the ZK mode in Emacs. In any case on-line help is available through the `HELP` command. The entire on-line help can be printed by issuing the following command:

```
> (help manual)
```

Examples of ZK theory development are provided in the `examples` subdirectory, which uses an example ZK library provided in the `library` subdirectory. Sources for the examples have a `.ver` file extension. To use a ZK library, you must set the library using the `SET-LIBRARY` command. For example, if your current working directory is the `examples` subdirectory, you can use the example ZK library by issuing the following command:

```
> (set-library "../library/")
```

All sources for the example library units are in the `examples` subdirectory.

For your convenience, there is a `PRINT-WORKING-DIRECTORY` command and a `SET-WORKING-DIRECTORY` command in ZK. Except for strings, ZK is case-insensitive on input[1] with all letters converted to uppercase.

[1]You can use the `|` bracketing to get case-sensitive input for symbols, e.g., `|Foo|` but it would be pointless since ZK does not allow lower case letters in identifiers.