Using Yggdrasil to Generate Stand-alone Programs from Shen

Mark Tarver
30/08/2023
Shen Group

What is Yggdrasil?

Yggdrasil is a program for generating stand-alone programs from Shen programs. That means, if Yggdrasil has been set up for a given language *L*, you can instruct Yggdrasil to generate *L* code from your Shen program that can be run in *L* independently of Shen.

How Does It Work?

Yggdrasil uses a *tree-shaker*; that is a program that eliminates dead code. In this case, Yggdrasil eliminates all that part of the kernel and the standard library that you do not need to run your application. Hence the name; 'Yggdrasil' refers the World Tree from which hang the 9 realms of Norse mythology.

Will It Work on all Shen Programs?

Non-interactive programs are the best candidate. This implementation is of the date of writing; still experimental.

What are the Requirements for Yggdrasil?

You need Shen 34.6 or higher with the standard library installed.

What Languages will Yggdrasil Work On?

Currently Yggdrasil works only with Common Lisp.

How Do I Use It?

1. Download Yggdrasil and open shen and cd to the Yggdrasil directory.
2. Type (load "commonlisp.shen").
3. Type (load "yggdrasil.shen").
4. Create a folder in the Yggdrasil directory where you want your object code to go. For example, MyProgramFiles.
5. Now copy your program files (say myprogram.shen) to the Yggdrasil directory.
6. Enter (yggdrasil <*language key*> [<*list your Shen files here*>] <*your object code directory*>). For example, (yggdrasil "lsp" ["myprogram.shen"] "MyProgramFiles") would generate the Lisp for your program.
7. Now consult the documentation for how to run the object code generated. In the case of Common Lisp you will find a file driver.lsp in your object code directory; enter (load "driver.lsp") to load all the files in the correct order.

How Can I set Up Yggdrasil for Other Shen Ports?

If you look at the file commonlisp.shen you'll see an example. Basically, you need to set up a series of pointers. You also need to understand something of the port.

1. First decide on the file extension for your language; I chose "lsp" for Lisp. Let's call your chosen language Blub. We'll have an extension "blub".
2. Create a file blub.shen and open it. Enter what follows.
3. Create a pointer from "blub" to the Shen-to-Blub compiler by typing

   (put "blub" compiler (fn <*compiler-function*>))

   Remember that the keyword compiler is external to Yggdrasil so if you are packaging your blub.shen file remember to make compiler external.
4. The value of <*compiler-function*> should be function that returns a string. More precisely it will return the Blub object code generated from a Kλ expression where that object code has been placed in a string. To find the function that generates the Blub object code, you need to search the backend for the Shen-to-Blub port.
5. Now you need to create a pointer to the **boilerplate code**. This is the code that prefaces any attempt to load a Blub program generated from Shen. It contains all the needed settings and declarations that will be needed for your Blub code to work. Enter

   (put "blub" boilerplate <*string*>)

   <*string*> is a string that will contain this code. Remember boilerplate should be external to your package.
6. The next part requires you to get the primitives for the Blub port. Find the primitive files which implement the encoding of the 46 primitive functions for Kλ. Go to the directory Primitives in the Yggdrasil directory and open a new subdirectory Blub. Copy the primitive files into the Blub directory.
7. Now you need to set up pointers from each primitive function *F* of the 46 instructions that make up Kλ to the file(s) where that function is defined. For example, in Lisp I have

   ```
   (put if "lsp" ["Primitives/CL/if.lsp"])
   ```

   In the Common Lisp port each primitive function is given its own file in order to avoid importing unnecessary object code into your stand-alone. However, it may be that the porter has piled all the primitives into one file – say primitives.blub. In which case this command will work.

   ```
   (map (/. X (put X "blub" ["Primitives/primitives.blub"])) (value yggdrasil.*primitives*))
   ```

8. The last step is to set up the *driver function*. This is the function that creates the driver file that loads all the bits and pieces in the correct order (starting with the boilerplate file). Don't forget that driver is external! The format is

   (put "blub" driver (fn <*driverfunction*>))

   For example, in the Common Lisp port I have this declaration (put "lsp" driver (fn lispdriver)).

9. How is the driver function set up? The driver function takes 5 inputs.
   a. The name of the boilerplate file. Here it will be `boilerplate.blub`.
   b. The list of primitive files; i.e., the list of the files you gave in step 7.
   c. A global file `globals.blub`. This is generated automatically by Yggdrasil. It contains all and only the needed global declarations borrowed from the kernel in order for the stand-alone to work.
   d. The kernel file; `kernel.blub`. This is the Blub code generated from that part of the kernel needed to run your application.
   e. Blub application files. These are the files generated from your Shen application.

   Your driver function thus has the type *string* → (*list string*) → *string* → *string* → (*list string*) → *string*. The final result is a string which contains the code needed in the driver file which will be written when Yggdrasil executes. Here this file will be called `driver.blub`.

   For example, in the Common Lisp port my `lispdriver` function is defined thus.

```
(define lispdriver
  BoilerFile PrimFiles GlobalFile KernelFile UserFiles
  -> (make-string
    "(LOAD ~S)~%(MAPC 'LOAD (LIST ~A))~%(LOAD ~S)~%(LOAD ~S) ~%(MAPC 'LOAD (LIST ~A))"
              BoilerFile (files PrimFiles) GlobalFile KernelFile (files UserFiles)))

(define files
  [] -> ""
  [File | Files] -> (@s (str File) " " (files Files)))
```

   When I run Yggdrasil on the N queens program, my driver file looks like this.

```
(LOAD "boilerplate.lsp")
(MAPC 'LOAD (LIST "error-to-string.lsp" "and.lsp" "equal.lsp" "hd.lsp" "arith.lsp" "tl.lsp"
"simple-error.lsp" ))
(LOAD "globals.lsp")
(LOAD "kernel.lsp")
(MAPC 'LOAD (LIST "n queens.lsp"))
```

That's it. You're now set to go!