Javascript Backend for Gnu Guile

Ian Price

This proposal is an amendment and update to my proposal from two years ago, summarising work that was completed and what needs to be done this year.

1 Summary

To implement a Javascript backend to Gnu Guile, capable of compiling programs for execution in a modern web browser.

2 Benefits

Guile users will benefit from being able to run Guile scheme, and other languages implemented by Guile in the browser. Together with SXML, SCCS, and the Guile webserver, this will mean Guile is suitable for handling all aspects of a web site (at least for small scales).

Eventually, this could allow projects using Guile, like Lilypond or a future Emacs, to integrate into the browser.

It also further validates Guile's "compiler tower" design, showing that you can not only add new front end languages, but also back end languages.

3 What was accomplished two years ago

In a previous GSOC, I was able to compile a large portion of the cps language to Javascript. Many test programs of increasing complexity were written and successfully compiled (and with some manual intervention, ran).

Examples include (link to site)

- Most of boot-9.scm, including the macro expander
- Amb Based Search (output code available here - warning, very big)

- Mergesort (output code available here)
- Mutually recursive numerical programs (output code available here)

To enable this I wrote

- A JS intermediate language distinct from the ecmascript language support we had in Guile already, at the recommendation of Andy Wingo. This prevents a circularity in the compilation process.
- A compiler for (language cps) to the JS intermediate language.
- A large boot.js implementing most of the Guile VM builtins, and the basic Guile types.

4 Deliverables

1. A completed compiler for converting CPS forms into Javascript forms
2. A complete preamble script (equivalent to boot-9) containing JS representations of Scheme types, and basic functions on those types.
3. A script for compiling Guile Scheme files to Javascript, and importing the relevant builtins needed to run it.
4. Tests and Documentation

(2) is the most significant contribution. The largest remaining difficulty is Tail Calls, which will be handled using a "Cheney on the MTA" strategy[1], as implemented in Chicken Scheme, rather than the common trampoline strategy. This will allow it to use the JS stack, and integrate better with other Javascript code.

(2) is necessary, but straightforward, and will be filled in needed.

(3) will be a new addition to the scripts directory. It is important to be able to reduce the amount of the Guile implementation that needs to be imported along side the users program as this reduces bandwidth costs and reduces the time to load the program.

(4) Documentation changes will only be needed in two sections. "Guile Implementation" will need to be updated to describe the JS AST, and about how to compile to it. "Hello Guile" should also be updated with a small section on "Combining with JS" similar to the section for C.

[1]. `http://www.pipeline.com/~hbaker1/CheneyMTA.html`

5 Plan

Although a lot of progress was two years ago, there are still several important parts that need to be implemented.

First, and most important, is to be able to compile modules. The module system was a sticking point two years ago, and presented debugging challenges. It is necessary for this to be done as the module is the basic unit of modularity in Guile.

Second, is to implement the Cheney on the MTA strategy. Two years ago, we were able to get a lot done without proper tail recursion by aggressively inlining, but this is no permanent solution.

Third, to correctly implement all dynamic environment builtins. The dynamic environment, including fluids, handlers, and dynamic wind are used heavily in Guile Scheme code, particularly for implementing the exception system.

Once these are done, I can move on to writing scripts for compiling Guile code into embeddable javascript, so that we can finally have Guile on the web.

After all this is done, I will have room to experiment with how best to integrate mixed Javascript and Scheme code, so as to provide the best possible experience for Guile programmers writing web applications.

6 Communication

Although I have not been active in the Guile community recently, since I have been prioritising university work, I have been a regular contributor to it in the past, and am familiar with its usual methods of communication, namely its freenode IRC channel and its mailing lists.

These would be the primary means of communication with my mentor, as they were two years ago, although I would also like to meet more regularly via video chat (skype / google hangouts / etc.), as I had found this very helpful in the past.

If more formal communication is desired, I can provide updates to the Guile user list, as I had done two years ago.

I would prefer for code to be available as a branch on the official repository, but I can host it on my private website or a public code repository site like Gitorious.

7 Qualifications

I participated in the Google Summer of Code two years ago, working on Scheme to Javascript compilation, and while incomplete, made significant progress.

I have been involved to various degrees in the Guile project since February 2011 (just prior to the 2.0 release), and have contributed fixes to many pieces of the source tree. In particular, I have posted fixes for the Lua and Tree-IL languages in Guile, and along with the previous work on Javascript compilation, am very familiar with the (language ...) subtree and its design decisions.

As I said two years ago, compilers are the most interesting pieces of software, and I still firmly believe this, and want to use this software to further my knowledge of both guile, compilation techniques in general, and modern Javascript implementations in particular , and I plan to continue writing them, whether as part of GSOC, Guile, or not.

Last year, although I unfortunately did not make the cut for the google summer of code, I did work on lower level software. I spent that summer designing and writing practicals for the University of Aberdeen's Computer Architecture course, which decided to adopt the use of the Raspberry Pi and ARM assembly language. I adopted a "translation based" approach, showing how simple rewrites and heuristics could be to used translate snippets of high level languages into assembly language. I believe this was informed by my interest and experience writing compilers both for fun, and as part of the summer of code two years ago.