Laconic ReadMe

Adam Yedidia

April 27, 2016

This document is intended for users who want to know how to run or compile a Laconic file, without necessarily knowing how to program one themselves. For a quick introduction to Laconic programming, read `laconic_quick_start.pdf`; for an exhaustive list of primitive operations in Laconic, read `laconic_ops.pdf`.

Laconic code can be processed in two ways. First, it can be *interpreted*; that is, it can be directly evaluated statement-by-statement. This is generally done to verify a program's correct behavior, and to correct errors which would result in thrown exceptions in the interpreter but might lead to undefined behavior in the compiled Turing machine (because the compiled Turing machine is optimized for parsimony, whereas the interpreter need not be).

Second, Laconic code can be *compiled* down to a variety of different lower-level representations. It is highly recommended, however, to first interpret any piece of Laconic code before compiling it, because the interpreter is much better for catching programming errors.

The compiler is general-purpose and not restricted to compiling the programs discussed in this thesis. It is optimized to minimize the number of states in the resulting Turing machine, not to make the resulting Turing machine time- or space-efficient.

1 Preparation

Before a Laconic file can be processed in any way, first it must exist in a place where it can be found. Make sure that the Laconic file is present, with the extension `.lac`, in the directory:

`parsimony/src/laconic/laconic_files/`

Then, BEFORE running any of the commands described below, navigate to:

`parsimony/src/laconic/laconic_meta/`

2 Interpretation

To interpret a Laconic file—to run the Laconic program as though it was in a normal programming language, and see what would happen if the Laconic program's commands were run explicitly—run the command:

`python laconic_interpreter.py` [name of Laconic program without `.lac` extension]

After you run this command, the interpreter will evaluate the program until an error is thrown or the program halts. It will also print any variables that you ask it to print.

3 Compilation

To compile a Laconic program—to transform it into a 2-symbol Turing machine—run the command:

`python laconic_to_2s_tm_compiler.py` [name of Laconic program without `.lac` extension]

This will cause a parsimonious 2-symbol Turing machine with equivalent behavior to your Laconic program to appear at:

`parsimony/src/tm/tm2/tm2_files/`[name of Laconic program without `.lac` extension]`.tm2`

If you'd prefer to transform your Laconic program to a 4-symbol Turing machine, run the command:

`python laconic_to_4s_tm_compiler.py` [name of Laconic program without `.lac` extension]

This will cause a 4-symbol Turing machine with equivalent behavior to your Laconic program to appear at:

`parsimony/src/tm/tm4/tm4_files/`[name of Laconic program without `.lac` extension]`.tm4`

Note that unlike the 2-symbol compiler, the 4-symbol compiler is not parsimonious; the 4-symbol Turing machine you'll end up with will have plenty of states. This option is primarily intended for if you want to watch a Turing machine execute which can be thought of as an equivalently-behaved but much-easier-to-read version of the 2-symbol machine. The lack of efficiency in states in the 4-symbol machine comes from the fact that the introspection algorithm is harder to implement with 4 symbols than with 2, and people are generally not interested in results relating to 4-symbol Turing machines in any case.

Finally, you can compile a Laconic program just to TMD:

`python laconic_to_tmd_compiler.py` [name of Laconic program without `.lac` extension]