Laconic to TMD Compilation

Adam Yedidia

May 11, 2016

This document is an explanation of the Laconic-to-TMD compilation process. It won't be useful to users who only want to know how to use Laconic files, or how to write Laconic files of their own. Rather, it is intended for an audience who is simply curious about how the algorithm works, or wants to write their own language which will compile down to TMD.

Each part of the logic in a Laconic program transforms into an equivalent section of TMD code. In the below sections, it is explained how this happens for each Laconic command.

Every Laconic compiler presented in this repository makes use of the algorithm described herein. So, for example, the compiler that transforms Laconic programs into 2-symbol Turing machines just calls this compiler, and then calls the compiler that transforms TMD programs into 2-symbol Turing machines. The same is true for compilation to 4-symbol Turing machines.

1 Variable Values

In Laconic, variables can have any value corresponding to an integer, a list of integers, or a list of lists of integers. In TMD, "variables" and "tapes" refer to the same thing, and correspondingly the "value" of a "variable" is simply the symbols currently on the tape.

Recall that in TMD, the tape alphabet is (`_`, `1`, `E`), and that tapes must take the form $\_(1|E)^*\_^\infty$. We call the second symbol on the tape the "home position" of that tape.

1.1 `int` Values

1.1.1 Non-Negative Integers

To represent the value of an `int x` with value $x \geq 0$, the tape representation is:

_1^{x}E_$^{\infty}$

So, for example, the value 5 would be represented as:

_11111E____...

1.1.2 Negative Integers

To represent the value of an `int x` with value $x < 0$, the tape representation is:

_E1^{-x}E_$^{\infty}$

So, for example, the value -3 would be represented as:

_E111E____...

1.2 `list` Values

To represent the value of a `list` that contains integer values e_1, e_2, e_3, and so on, the tape representation is:

_E(($e_1 > 0$)?1 : E)1$^{|e_1|}$E(($e_2 > 0$)?1 : E)1$^{|e_2|}$E(($e_3 > 0$)?1 : E)1$^{|e_3|}$E . . . _$^{\infty}$

In other words, the tape representation of the `list` begins with an `E` in the home position, and then for each element of the `list`, there would be a `1` if the element positive and an `E` otherwise, followed by a number of `1`'s equal to the magnitude of the number, followed by an `E`. So, for example, the value $[5, -2, 0]$ would be represented as:

_E111111EE11EEE____...

1.3 `list2` Values

To represent the value of a `list2` that contains `list` values l_1, l_2, and so on, each of which contains `int` values e_{11}, e_{12}, and so on, and e_{21}, e_{22}, and so on, and so on, the tape representation is:

$\texttt{\_EE1}((e_{11} > 0)?\texttt{1} : \texttt{E1})\texttt{1}^{|e_{11}|}\texttt{E1}((e_{12} > 0)?\texttt{1} : \texttt{E1})\texttt{1}^{|e_{12}|}\texttt{E1}\ldots\texttt{EE1}((e_{11} > 0)?\texttt{1} : \texttt{E1})\texttt{1}^{|e_{11}|}\texttt{E1}((e_{11} > 0)?\texttt{1} : \texttt{E1})\texttt{1}^{|e_{11}|}\texttt{E1}\ldots\ldots\texttt{E\_}^{\infty}$

In other words, the tape representation of each list in the `list2` begins with `EE1`, and then for each `list` in the `list2`, the `int`s in those `list`s are preceded by a `1` if the number is positive and a `E1` otherwise, followed by a number of `1`'s equal to the magnitude of the number, followed by a `E1`. Then, to signal the end of a `list`, the sequence `EE1` is used. Finally, to signal the end of the entire `list2`, the symbol `E` is used, followed by infinite underscores. So, for example, the value $[[3, -1], [], [0, 4]]$ would be represented as:

```
_EE11111E1E11E1EE1EE1E1E111111E1E____...
```

2 Laconic Operations

Each operation in Laconic has a corresponding TMD function which performs the operation. As an example, the Laconic operation `a*b` will compile down to a call to the TMD function `BUILTIN_multiply`.

The following is the full mapping between Laconic operations and TMD functions:

Operation	Laconic symbol	TMD function
Addition	`+`	`BUILTIN_add`
Subtraction	`-`	`BUILTIN_subtract`
Multiplication	`*`	`BUILTIN_multiply`
Integer Division	`/`	`BUILTIN_divide`
Negation		`BUILTIN_intneg`
Equality	`==`	`BUILTIN_equal`
Inequality	`!=`	`BUILTIN_notEqual`
Greater Than	`>`	`BUILTIN_greaterThan`
Less Than	`<`	`BUILTIN_greaterThan` (arguments reversed)
Greater or Equal	`>=`	`BUILTIN_greaterOrEqual`
Less Than or Equal	`<=`	`BUILTIN_greaterOrEqual` (arguments reversed)
And	`&`	`BUILTIN_and`
Or	`|`	`BUILTIN_or`
Not	`!`	`BUILTIN_assignNot`
List Index	`@`	`BUILTIN_listindex`
List2 Index	`@*`	`BUILTIN_list2index`
List Append	`^`	`BUILTIN_append`
List2 Append	`^*`	`BUILTIN_list2append`
List Length	`#`	`BUILTIN_len`
List2 Length	`#*`	`BUILTIN_len2`
List Concatenation	`||`	`BUILTIN_concatenate`
Explicit Integer	—	`BUILTIN_assign*`
Explicit List Description	`[]`	`BUILTIN_listAssemble*`
Explicit List2 Description	`::`	`BUILTIN_list2Assemble*`

The above TMD functions are all available for perusal at:

```
parsimony/src/tmd/laconic_std_library/
```

Note that for explicit descriptions of `ints`, `lists`, or `list2s`, there are separate functions callable based on the size of the relevant value. This means that it is important to avoid putting integers larger than 20 or so, or lists of similar length, into the program explicitly. (Building them up implicitly, of course, is fine.) I would recommend against doing this, because probably at this point it is more parsimonious to describe the number or list implicitly, but if necessary one can use the programs `assemblexgen.py` and `assignxgen.py`, which can be found in the `tmd_meta` directory, to generate larger explicit-description functions.

The Laconic Standard Library has more than just the functions shown in the table above. It also contains many helper functions which are called by the functions above.

The Laconic Standard Library's functions all begin with the string "`BUILTIN_`". It is therefore a very bad idea to begin your function names in TMD or Laconic with that string.

Every function in the Laconic Standard Library was written by me in order to compute the desired operation. Each one is composed entirely of explicit tape commands, return statements, and calls to other built-in functions.

3 Complex Expressions

Complex expressions are expressions that are made up of multiple operations, such as (a+b)*c, for example. There is no order of operations to worry about, since full parenthesization is required.

Compiling a complex expression is somewhat more tricky than compiling a single operation, since the built-in functions can only handle one operation at a time. The solution is to hold temporary results in *holder variables*, which are created automatically by the compiler. They are created as necessary and reused when their contents become irrelevant. They are recognizable by the fact that their names all begin with the `!` symbol. They exist only in the compiled TMD, of course.

When a variable is assigned to the value of a (potentially complex) expression, a series of function calls to built-in functions (potentially involving holder variables) is the result in the compiled TMD. Then, the built-in function `BUILTIN_assign` is called to transfer the held value of the result of the complex expression to the variable.

4 If Statements

In Laconic, an `if` statement has the following form:

`if ([`*expression*`]){[`*function body*`]}`

This is compiled down to a section of TMD code that looks like the following:

[Sequence of commands designed to load the value of *expression* into the holder variable h]
`[`h`] E (IF_STATE_`i`_FALSE); 1 ()`
[Compiled version of *function body*]
`IF_STATE_`i`_FALSE:` [Continuation of the program]

In a Laconic program, the function body is evaluated if and only if the expression evaluates to a positive integer (and the expression is forbidden from being a `list` or `list2`). As can be seen above, in the compiled TMD, the function body is evaluated if and only if the home-position tape symbol in the result of the expression is not an `E`. As can be seen from Section 1, this corresponds precisely to the scenarios when the value of the expression is equal to a positive integer.

5 While Statements

In Laconic, a `while` statement has the following form:

`while ([`*expression*`]){[`*function body*`]}`

This is compiled down to a section of TMD code that looks like the following:

`WHILE_TEST_`i`:` [Sequence of commands designed to load the value of *expression* into the holder variable h]
`[`h`] E (WHILE_STATE_`i`_FALSE); 1 ()`
[Compiled version of *function body*]
`[`h`] E (WHILE_TEST_`i`); 1 (WHILE_TEST_`i`)`
`WHILE_STATE_`i`_FALSE:` [Continuation of the program]

In a Laconic program, the function body is evaluated repeatedly until the expression evaluates to a non-positive integer (and the expression is forbidden from being a `list` or `list2`). As can be seen above, in the compiled TMD, the function body is evaluated repeatedly until the home-position tape symbol in the result of the expression is an `E`. As can be seen from Section 1, this corresponds precisely to the scenarios when the value of the expression is equal to a non-positive integer.

6 Function and Variable Compilation

Each Laconic function defined within a Laconic program is given its own corresponding TMD function file. There is additionally a TMD function created named `main`, which is given its own function file and contains all of the logic that exists in the Laconic program outside of function definitions. The function `main` is called first in the TMD program, and is accordingly put at the top of the `functions.tff` file.

The input line of `main.tmd` contains every variable declared in the Laconic file, along with every holder variable needed in `main` or in any of the other functions defined in the Laconic program.

Once `main` is fully converted to TMD as described in the sections above, we find all of `main`'s "dependencies" on other functions, by going through each function called by `main`. For each of those functions, we include them in `functions.tff`, and we recursively transform them to TMD as above if necessary, and import their dependencies as well. Some of these functions may come directly from the Laconic Standard Library, in which case they are copied to the target directory; others will be functions defined within the Laconic program, in which case they need

The functions are `functions.tff` are ordered from top to bottom in decreasing order of how many times each is called, with more frequently-called functions towards the top of the file. (Of course, `main` is the top function in the list.)