Proposal: The Towel Programming Language

Zihang Chen (zc2324), Baochan Zheng (bz2269), Guanlin Chen (gc2666)

September 29, 2015

Contents

1 What is it anyway

A towel, is about the most massively useful thing an interstellar hitchhiker can have.

The Hitchhiker's Guide to the Galaxy

The Towel programming language is a general-purposed, stack-based, statically typed, functional, postfix-syntaxed programming language. It is inspired by some famous functional programming language, like Erlang, Haskell and Lisp.

1.1 General-purposed

Towel aims at being able to program everything, from scripting to get your files backed up, to large projects such as web applications. *It's also very good at sending stress signals to strags.* This means that the syntax should be expressive (yet concise). And *maybe* a foreign function interface should also be provided to extend the ability of Towel.

1.2 Stack-based

Stack-based languages are simple to implement, yet powerful and relatively efficient to use. It's the best if you want to implement a language in three months.

1.3 Statically typed and functional

Being statically typed, means that everything has a type. Futhermore, these types are tagged to names before we can execute the program. With this being done, things run more efficiently. Types also help Towel to figure out the order of computation, we'll see that in a moment.

However, this does not mean that you have to tag types to the names yourself, Towel does that for you (most times). The rationale is that, in functional programming, programs are simply expressions made up by functions (the operators are functions too), and the values they are applied with. Since functions are values too, they have types too, thus we can infer from the function type the types of arguments, thus the types of the names.

We don't plan to implement sophisticated type system in Towel. One thing is that most times, simplicity is good and all you need. The other thing is that we don't believe we have enough related mathematic knowledge.
It is worth mentioning here that the built-in types of Towel will be `Int` and `Float`, which are subclasses of `Number`, `String`, `List` and finally the superclass of every type `Any`. Subclasses are said to be "compatible with" their superclasses.

Literals are provided with these built-in types.

1.4 Postfix-syntaxed

We have all seen prefix-syntaxed and infix-syntaxed languages, a lot.

What about postfix-syntaxed ones? We rarely see them. There seems to have much less language based on postfix expression. Why? Is it not popular for a certain reason? We'll find it out with Towel.

2 And now for something completely different...

To see Towel in action, let me present some code in Towel.

```
‘‘this is a comment.’’
bind SomeLiterals [1 ‘‘numbers: FixedInt, Float, Int and’’
                     ‘‘their superclass Number’’
                   (2 3 +) ‘‘a sequence to be evaluated,’’
                           ‘‘i.e., a special anonymous function’’
                   (2 *) ‘‘a partial anonymous function’’
                   'spam spam\nspam' ‘‘strings are single-quoted’’
                   arthur-dent ‘‘Atoms are not capitalized,’’
                               ‘‘whereas names are’’] ‘‘Lists’’
in
  SomeLiterals@3 Printn. ‘‘print the third element (the atom) with and \n’’

bind Quicksort function L,
  match
    []:;
    Head Tail &:
      @ Tail Head < Filter Quicksort
        [Head]
        Tail Head >= Filter Quicksort
        ++!
```

From now on, it's just technically memo which shows to some extent how things will work in Towel.

2.1 The compilation

Overall, the program (with extension “t”) is compiled into slightly-compiled-towel program (with extension “o”) which looks a bit like stack-based assembly source code. Then the compiler simulates running the slightly-compiled-towel to inflate the source code with type information and order of computation. This results in files called more-slightly-compiled-towel programs (with extension “w”).

With more information in “w” files, we can construct a more meaningful abstract syntax tree (the AST made from “o” files would probably be some linked lists), and be able to perform some optimizations (which probably won’t be implemented due to limited time). The compiler will then convert the AST in plain text form. This procedure produces the “e” files, called optimized-towel programs.

Finally, the compiler grabs the “e” files and produces bytecode version programs called binary-towel programs, with extension “l”. One thing to clarify is that you get one “o”, “w”, “e”, “l” file per “t” source code file.

The rationale that we divide the whole procedure into multiple layers is to simplify it. This layering reduces the responsibility of each layer and thus leads to quicker prototyping and codes that are less prone to bugs as well.

For more compilation details, see A Appendix I.

2.2 More technical details

Now suppose we skip the “o” files and examine the source code directly: a function is bound to the name `Quicksort`. And the function takes an argument named **L** which, by type inferring, is determined to be of type `List`. When `Quicksort` is applied with a list, the Towel pushes the name `L` onto the stack.

When pushing a name, Towel simply copies the name and push the copy onto the stack for later use. At this time, the name `L` and the copied name on the stack both reference to a list value maintained in a global value table. This implies that destroying the name on the stack **does not affect** you referencing to the value to which `L` references.

It should be pointed out that `match` is a special form keyword that pattern-matches the top element of the stack, which is like this:

```
match pattern-1: branch-expr-1;
      pattern-2: branch-expr-2;
      ...
      pattern-k: branch-expr-k;
      ...
```

By knowing this, the top element on the stack is pattern-matched against the patterns. More precisely, whatever `L` is referencing to is matched against `[]` and `Head Tail &` in a one-by-one manner. If the matching against `[]` succeeds, the `match` special form evaluates to its first branch expression, i.e., an empty expression, in other words, it’s the top element of the stack, which is an empty list. The rest branches are simply ignored.

Patterns are simply functions with return type of List with the keyword `pattern`, rather than `function`. The elements of the returned list are bound to the names in the patterns.

The other case is that the matching against `[]` fails, whereas `Head Tail &` succeeds, then the head of `L` is bound to `Head`, while the rest of `L` is bound to `Tail`, just like OCaml and Haskell. And the second branch expression is evaluated under these name bindings.

Then, the compiler pushes the whole bunch of

```
@ Tail Head < Filter Quicksort [Head] Tail Head >= Filter Quicksort ++
```

onto the stack. And when Towel reads an expression terminator, it pushes an `eval'` instruction onto the stack. Terminator includes comma, semicolon, exclamation mark and period. Quick hint, if you wish to manually change the order of computation, add any of the terminators (except period and semicolon if you are in `match` form) to appropriate position(s). In the example above, the terminator is an exclamation mark whose meaning will be revealed later.

When computing on the stack and instruction `eval'` is reached, whatever next to eval is evaluated, which is, in most times, a function. To evaluate a function, Towel continues to pop more elements one by one. For any element just popped from the stack, Towel examines the type of it to see if it agrees with the type constraint of the function. This leads to three kinds of results:

- The types simply agree with each other. Then the compiler continues to pop more.
- The types does not agree, however, the element is a function whose return type meets the type constraint. In this case, the compiler then saves the current status, and turns to evaluate this element. (And inserts `eval'` instruction to related position in "w" file to mark this priority.)
- The types does not agree at all. Compiler complaints about this, refuses to proceed and exits.

Once evaluated, the value is pushed back onto the stack and the procedure proceeds until all of the arguments are satisfied. In case of a function, the return value of the function is whatever left over on top of the stack.

After all of the expressions evaluated, the compiler retrospects what happened during the simulation and attaches important information to the "o" files. This procedure produces the "w" files. For example, the order of computation is represented by `eval'`s inserted during evaluations of the expressions. In other words, most times Towel will decide your operator (or even function) priority for you with respect to the type information. If you wish to manually control the order, add   which stands for instruction `eval'`.

What about the piece of code `Tail Head < Filter 'eval`, or anonymous partial functions? Let's do a simulation here:

1. Pop `eval'` and evaluate `Filter` which is of type `List -> (Any -> Bool) -> List`. The compiler begins to find the last argument (since we're on a stack) for `Filter`
2. Pop `<` and examine its type: it's of type `Number -> Number -> Bool` which doesn't agree with `Any -> Bool`. However, the second case of type constraint applies here: their return types are compatible. The compiler then examines if `Number -> Number -> Bool` has any chance to be compatible with `Any -> Bool`. Since class `Number` (not the object-oriented term `class`) is a subclass of class `Any` by default, it can be inferred that if one more `Number` is applied to `<`, we could form an anonymous partial function of type `Number -> Bool` which complies with type `Any -> Bool`.
3. Pop `Head` and form an anonymous partial function with `<`. The compiler record this discovery in "w" file.
4. Push the new function onto stack.
5. Current evaluation is done. Resume to the upper one.

The same rules apply to the rest of the program, thus we will elaborate no more here.

Wait, what's that `@` over there? Glad you asked. Suppose I have a variadic function, how many arguments will I supply to it? `@` comes in handy in such situation. When evaluation terminator is an exclamation mark, everything between the at sign and the function name is applied to the function. For example, when `++` (function for concatenating lists) is terminated with period, it is a binary function, on the other hand, when it is terminated with exclamation mark, it takes as many arguments as possible until it reaches `@`.

2.3 Towel for Vogons

Vogons may like Towel too, because they know we respect how they like to write poems and plan to provide for them an alternative lexer which understands programs written in a Vogon-poem manner.

The lexer simply reads the first character of each word, ignores the rest of them and converts the recognized character (not all characters are meaningful as a token) into a valid token. And the rest is the same with what is shown above.

Maybe the quicksort program can be written as

```
behold my Quickly fried Fish,
  moistly empty, emptiness;
  Hard as comsuming a Rock,
    while Rock Hard is Less than Filter Quickly
          oh Hard
          Rock Hard Greater Filter Quickly
          Consume!
```

Technically this[1] is equivalent to the following:

```
b(ind) Quickly f(unction) Fish,
  m(atch) [](i.e. e(mpty)), [](i.e. e(mptiness));
  Hard c(onstructs with) Rock,
    while(@) Rock Hard <(Less) Filter Quickly
    [Hard](o means list literal here)
    Rock Hard Greater Filter Quickly
    ++(Consume)!
```

One thing to notice is that irrelevant words are stripped out, such as words starting with "a", "i" and "m".

If you would like to write poems describing something else, you can configure the lexer to recognize other characters as valid tokens.

A Much more technical details

"o" files are somewhat simple lists of instruction in a text file format. For the quicksort example, the "o" file looks like this:

```
bind Quicksort to
function L
match []
eval
ret
match Head :& Tail
push @
push Tail
push Head
push <
push Filter
```

[1] *Pffft..., this may be even worse than Vogon peoms.*

```
push Quicksort
push [Head]
push Tail
push Head
push >=
push Filter
push Quicksort
push ++
eval-variadic
ret
end-function
push [4 5 1 3 2]
push Quicksort
eval
end-bind
```

When outputting "w" files, the compiler simulates the running process on "o" files and see if types are compatible. The pseudo code is here:

```
push all the "o" program into stack
init a exec stack
pop top element to x
write x to "w"
push type of x to exec stack
while stack is not empty:
    let x = top element of the stack
    examine if x is type-compatible to the top of exec stack
    if no:
        examine if x is a function
        (i.e., needs a new stack and has high priority)
        if yes:
            examine if return type of x is type-compatible to\
            the top of exec stack
            if yes:
                calculate how many arguments it need to be fully\
                compatible with the return type
                generate a new id for the upcoming partial function, say P
                write make-partial-function-end :P
                write eval' ;; we calculate this first
                write x
                write and pop as many next xs as needed
                write make-partial-function-marker :P
                cancel out correspond part of the type at the top\
                of exec stack
            else:
                reject
        else:
            reject
    else: ;; it's a type-compatible non-function value
        write x
        cancel out correspond part of the type at the top of exec stack
reverse the written output to "w"
```

It's a bit complicated, because we have to consider two scenarioes, where x is a function or a non-function. The resulted "w" file looks like this, which is more runnable to virtual machine.

```
                make-stack 0, 0
                make-scope

                add-to-scope ??, Quicksort

:f1             make-stack 0, 1 ;; because Quicksort, as its definition,
                                ;; has one argument
                make-scope
                add-to-scope 0, L ;; bind the top element to L
                                  ;; in current scope
                ;; we are done for function initialization

:f1-m1          match-init
:f1-m1-1        match-empty
                eval'
                jump :f1-m1-end
:f1-m1-2        match Head Tail &
                push variadic-marker
                push Tail
                make-partial-function-marker' :f1-anonyf1
                push Head
                push <
                eval'
                make-partial-function-end' :f1-anonyf1
                push Filter
                eval' ;; when evaluating a function (the control sequence
                      ;; jumping to other location),
                      ;; `eval'` pushes current IP to return address stack
                push Quicksort
                eval'
                push [H]
                eval'
                push Tail
                make-partial-function-marker' :f1-anonyf2
                push H
                push >=
                eval'
                make-partial-function-end' :f1-anonyf2
                push Filter
                eval'
                push Quicksort
                eval'
                push ++
                eval-variadic'
:f1-m1-end      ret ;; ret copies whatever is on top of current stack
                    ;; to the upper stack, destroyes current stack and
                    ;; jump to previous location saved in return
                    ;; address stack

                push [3, 1, 4, 2, 5]
                push Quicksort
                eval'
                exit
```

One kind of optimization that transformations from ‘w‘ to ‘e‘ actually do is reducing some of the ‘eval’‘s, for example, the ones that evaluates partial functions. Thus reduces the work needed to be done at runtime.

The instructions between a make-partial-function-marker’ and the corresponding end instruction is the actual function body of the partial function, which is to be scraped out and put to a new place along with some function initialization boilerplate.

The output looks like this:

```
                make-stack 0, 0
                make-scope

                add-to-scope :f1, Quicksort ;; resolves previously
                                            ;; unresolved label

:f1             make-stack 0, 1 ;; because Quicksort, as its definition,
                                ;; has one argument
                make-scope
                add-to-scope 0, L ;; bind the top element to L in
                                  ;; current scope
                ;; we are done for function initialization

:f1-m1          match-init
:f1-m1-1        match-empty
                eval'
                jump :f1-m1-end
:f1-m1-2        match Head Tail &
                push variadic-marker
                push Tail
                push @:f1-anonyf1 ;; reduce [Head < eval'] to an
                                  ;; anonymous partial function
                push Filter
                eval'
                push Quicksort
                eval'
                push [H]
                eval'
                push Tail
                push @:f1-anonyf2
                push Filter
                eval'
                push Quicksort
                eval'
                push ++
                eval-variadic'
:f1-m1-end      ret ;; ret copies whatever is on top of current stack
                    ;; to the upper stack, destroyes current stack and
                    ;; jump to previous location saved in jump address stack

                push [3, 1, 4, 2, 5]
                push Quicksort
                eval'
                exit
```

```
:f1-anonyf      make-stack 0, 1 ;; according to type constraint information,
                                ;; Towel calculates that only one argument
                                ;; is needed
                make-scope
                add-to-scope 0, :f1-anonyf1-arg1
                push Head
                push <
                eval'
                ret

:f1-anonyf2     make-stack 0, 1 ;; according to type constraint information,
                                ;; Towel calculates that only one argument
                                ;; is needed
                make-scope
                add-to-scope 0, :f1-anonyf1-arg1
                push Head
                push >=
                eval'
                ret
```

B Recognising different parts of the Towel

For the purpose of being maintainable and collaboratible, Towel will be decomposed into following parts which may contain several modules:

- General utilities
- The compiler
- The virtual machine (we are not going to generate any real assembly or code, though translating "e" files to C seems easy)
- The standard library

General utilities is used by both the compiler and the virtual machine. It may contain:

- Type system
- Maintenance of object reference table
- Maintenance of names and scopes
- Garbage collection
- Package and module control
- etc.

The compiler may contain:

- Module import handling
- Module that transform source code ("t" files) to slightly-compiled-towel ("o" files)
- Module that transform "o" files to more-slightly-compiled-towel ("w" files)
- Module that transform "w" files to optimized-towel ("e" files)
- Module that transform "e" files to bytecode-towel ("l" files)

- Vogon lexer

The virtual machine may contain:

- Module that reads “e” and “l” files and streams instructions
- Module that execute the instructions
- Module that interoperates with the foreign function interface
- Debug utilities

The standard library (called rack in Towel) may contain:

- Arithmetic operations
- String utilities
- IO utilities
- etc.

Implementation of each module will be (hopefully evenly) distributed to each member of the team. By the `helloworld` demo, all modules of compiler, certain parts of general utilities and virtual machine, and IO utilities of the standard library shall be implemented without error.