The Towel Programming Language

W4115 PLT, Fall 2015

Zihang Chen (zc2324) Baochan Zheng (bc2269) Guanlin Chen (gc2666)
December 21, 2015

Columbia University
Introduction

What is Towel anyway?
It is ...

- \( \lambda \) : tail recursion, function as first-class citizen, etc.
- Stack-based and postfix-syntaxed
- Dynamically strong-typed
- General-purpose

42 !println
What does it look like?

```plaintext
import 'std' @

bind Fold-left ,\ Acc Xs Fun,
   (Xs ?# ift Acc,
    (Acc Xs #hd Fun Xs #tl Fun` Fold-left@))
also Sum (0 (+` Fold-left /flip))
then ([1 10 11 20] Sum !println)
```
Let me do some syntax-highlighting here.

```plaintext
import 'std' @

bind Fold-left ,\ Acc Xs Fun,
    (Xs ?# ift Acc,
        (Acc Xs #hd Fun Xs #tl Fun` Fold-left@))
also Sum (0 (+` Fold-left /flip))
then ([1 10 11 20] Sum !println)
```

- **Language Structures**: Sequence, if forms, Function, Backquote, bind-then forms, import form, export form
Let me do some syntax-highlighting here.

```plaintext
import 'std' @

bind Fold-left ,\ Acc Xs Fun,
  (Xs ?# ift Acc,
    (Acc Xs #hd Fun Xs #tl Fun` Fold-left@))
also Sum (0 (+` Fold-left /flip))
then ([1 10 11 20] Sum !println)
```

Tail recursive function call

- **Language Structures**: Sequence, if forms, Function, Backquote, bind-then forms, import form, export form
How to recognize different parts of the example

Let me do some syntax-highlighting here.

```plaintext
import 'std' @

bind Fold-left ,\ Acc Xs Fun,
    (Xs ?# ift Acc,
        (Acc Xs #hd Fun Xs #tl Fun` Fold-left@))
also Sum (0 (+` Fold-left /flip))
then ([1 10 11 20] Sum !println)
```

Partial function application

- **Language Structures**: Sequence, if forms, Function, Backquote, bind-then forms, import form, export form
How to recognize different parts of the example

Let me do some syntax-highlighting here.

```plaintext
import 'std' @

bind Fold-left ,\ Acc Xs Fun,
    (Xs ?# ift Acc,
        (Acc Xs #hd Fun Xs #tl Fun` Fold-left@))
also Sum (0 (+` Fold-left /flip))
then ([1 10 11 20] Sum !println)
```

- **Literals:** literals for atoms, numbers, strings, lists, tuples are supported
How to recognize different parts of the example

Let me do some syntax-highlighting here.

```plaintext
import 'std' @

bind Fold-left ,\ Acc Xs Fun,
   (Xs ?# ift Acc,
    (Acc Xs #hd Fun Xs #tl Fun` Fold-left@))
also Sum (0 (+` Fold-left /flip))
then ([1 10 11 20] Sum !println)
```

- **Names**: extensive characters supported, flexible naming
How to recognize different parts of the example

Let me do some syntax-highlighting here.

```
import 'std' @

bind Fold-left ,\ Acc Xs Fun,  
  (Xs ?# ift Acc,  
   (Acc Xs #hd Fun Xs #tl Fun` Fold-left@))  
also Sum (0 (+` Fold-left /flip))  
then ([1 10 11 20] Sum !println)
```

- **Language Structures**
- **Literals**
- **Names**

The above three are what we call **words** in Towel. A program in Towel is essentially a **sentence** of words.
Types in Towel

Towel supports the following type:

- **Int** → Big integer
- **FixedInt** → Signed 64-bit integer
- **UFixedInt** → Unsigned 64-bit integer
- **Float** → IEEE754 floating point
- **Atom**
  → A constant with a name (see also Erlang atoms)
- **String**
  → String (one of the Enumerable types)
- **List**
  → Linked list (one of the Enumerable types)
- **Tuple**
  → Fixed, random accessible enumerable data type
- **Function**
  → Passing around a piece of code
In module **Std**, you will find ...

- Arithmetic Functions: +, -, etc. So **no operators**.
- Conversion and Reflection Functions: ~int, ~str, etc.
- Routines: functions with side(or stack)-effects, e.g. !println, !!pop, !!dup, etc.
- Functions that work with enumerables: #hd, #tl, #cons, etc.
- The Fun Functions: /foldl, /map, /filter, etc.
- Variadic Functions: a paceman that eats arguments until the stack is empty. See manual for more detail.
How *weave* compiles a piece of *towel*: it ...

1. **Source → Tokens**
   tokenizes the source code using a scanner

2. **Tokens → AST**
   parses the tokens with a parser

3. **AST → IR AST**
   traverses and transforms AST to IR AST (along with some scope analysis that will detect unbound names)

4. **IR AST → Bytecode**
   compiles IR AST into bytecode representation

*Bytecode* is runnable via the Towel Virtual Machine!
The Towel Virtual Machine

The Towel Virtual Machine is a piece of software that ...

1. **Bytecode** → **IR AST**
   decompiles bytecode to IR AST

2. **IR AST** → **42**
   interprets the IR AST (essentially an array of instructions) one by one so you can get the answer

You can use the **Extension** mechanism to call OCaml functions from within the Towel Virtual Machine! See manual for more detail.
The future of Towel

- A native compiler that compiles IR to C code.
- Better error messages, both for the compiler and the virtual machine.
- Better debugging facilities: need to make use the dynamicness feature of Towel.
- Enrich the standard library so that it’s batteries-included and general-purpose.
- Statically typed Towel!
  A stack-based language is very dynamic due to its uncleanness of the data (i.e. type) flow. A static-typed Towel could be made by analyzing each function’s stack-effect.
And now for something completely different...

The DEMO

- Partial function application
- Tail calls
- Standard library
- Extensions to the Towel Virtual Machine
- The test suite
- Anything you would like to ask