Shen: A Sufficiently Advanced Lisp

Aditya Siram (@deech), Intoximeters, Inc.

September 20, 2014

Outline

Functions

- Defining Function ...

```
(define welcome
 X nil -> (output "Welcome to ~A" X)
 X Y   -> (output "Welcome to ~A and ~A" X Y))
(welcome "Strange Loop" nil) =>
     "Welcome to Strange Loop"
(welcome "Strange Loop" "St. Louis") =>
     "Welcome to Strange Loop and St. Louis"
```

Typed Functions

- Turn on typechecking, type signatures are required . . .

  ```
  (tc +)
  ```

- Simple Types

  ```
  (define is-positive
    { number --> boolean }
    X -> (and (number? X) (>= X 0)))
  (is-positive -1) => false : boolean
  ```

Typed Functions

- Polymorphic Types

```
(define map
  { (A --> B) --> (list A) --> (list B) }
  F [] -> []
  F [ X | XS ] -> [(F X) | (map F XS)])

(map (function (+ 1)) [1 2 3]) =>
  [2 3 4] : (list number)
```

- Note the currying above ...

Untyped Functions

- Turn off typechecking, type signatures are optional ...

```
(tc -)
(define is-positive
  X -> (and (number? X) (>= X 0)))
(define map
  { (A --> B) --> (list A) --> (list B) }
  F [] -> []
  F [ X | XS ] -> [(F X) | (map F XS)])

(is-positive 10) => true
(map (/. X (+ X 1)) [1 2 3]) => [2 3 4]
```

- Notice no types in results & the alternate lambda syntax

Features

- Optional typechecking
- Very flexible type specification
- (Prolog) Pattern matching

```
(define same
  X X -> true
  _ _ -> false)
```

Features

- Full Prolog engine
- Full parsing engine
- Decompiles to KLambda (Scheme-like)
- Your Shen = A Scheme interpreter + 45 primitive functions
  - if, and, or, cons ...
- "A unifying meta-language ..."

Web Application Demo

- Backend in Ruby (Sinatra)
- Frontend in JavaScript (Qooxdoo)
- Common code in Shen
- This is the important bit.

Form Demo

- Simple Login Screen
- Check form inputs using identical Shen code.

Form Demo

- Validation data-structure

```
(set *model*
  (@v (@p "username"
          (function (parses (function <alphas>))))
      (@p "password"
          (function parse-password))
   <>))
```

- This is a global variable!
- And a function that runs the validator . . .

```
(define run-validator
  nil   Key -> false
  Value Key ->
  (let Validator (lookup (value *model*) Key)
    (Validator Value)))
```

Form Demo

- The lookup function ...

```
(define lookup
 (@v (@p Key Validator) _) Key -> Validator
 (@v _ PS) Key -> (lookup PS Key)
 <> Key -> (error "No validator for ~A." Key))
```

- Notice use of unification!

Form Demo

- The username parser is standard

```
(defcc <alphas>
 <alpha> <alphas> := [<alpha> | <alphas> ];
 <alpha> := [<alpha>])
```

```
(compile (function <alphas>)
         (string->bytes "abcdef")) =>
   [97 98 99 100 101 102]
```

Check Parser Output

- Check the parser runs

```
(define parses
  Parser Input ->
  (trap-error
    (do
     (compile
      (function Parser)
      (shen.string->bytes Input))
     true)
    (/. E false)))
```

- ‘do‘ macro for imperative programming
- ‘trap-error‘ for exceptions

Form Demo

- The password parser is more interesting

```
(defcc <password>
 shen.<digit> <password>
      := [(function add-digit) | <password>];
 X <password>
      := [(function add-special) | <password>]
           where (element? X (shen.string->bytes "_!"));
 shen.<alpha> <password>
      := [(function add-alpha) | <password>];
 <e> := []
 )
```

  - User input is translated into function!

Form Demo

- Parsing forms a chain of functions

```
"Hi123!_"
   =>
[
 (function add-alpha)
 (function add-alpha)
 (function add-digit)
 (function add-digit)
 (function add-digit)
 (function add-special)
 (function add-special)
]
```

Form Demo

- Add functions

```
(define add-alpha
  (@p Alpha Digit Special) ->
    (@p (+ Alpha 1) Digit Special))
(define add-digit
  (@p Alpha Digit Special) ->
    (@p Alpha (+ 1 Digit) Special))
(define add-special
  (@p Alpha Digit Special) ->
    (@p Alpha Digit (+ 1 Special)))
```

Form Demo

- Compose them!

```
(shen.compose
  [
   (function add-alpha)
   (function add-digit)
    ...
   ]
   (@p 0 0 0))

"Hi123_!" => (@p 2 3 2)

(define validate-password
  (@p Alpha Digit Special) ->
    (and (= Alpha Digit)
         (not (= Special 0))))
```

Form Demo

- Ruby/JavaScript hookup

```
Shen.call_by_name(
   "run-validator",
   [value, "password"]
 )
```

```
shen.send(
 "run-validator",
 password, "password"
 )
```

Counter

- Increment a counter over HTTP
- Make it type safe.
- Shen functions . . .

```
(define string->positive-number
  { (readable string) --> positive-number }
  X -> (make-positive (toNumber X)))
(define add-positive
  { positive-number --> positive-number -->
    positive-number }
  X Y -> (+ X Y))
(define is-positive
  { number --> boolean }
  X -> (and (number? X) (>= X 0)))
```

Counter

- Type specification

```
(datatype positive-number
  X : positive-number;
  Y : positive-number;
  ________________________
  (+ X Y) : positive-number;

  ...
 )
```

- To show '(+ X Y)' inhabits 'positive-number'
  - Show 'X' inhabits 'positive-number' and
  - Show 'Y' inhabits 'positive-number'

Counter

- So how do you make a ‘positive-number‘? Verified types . . .

```
(datatype positive-number
   ...

   ______________________________________________
   (is-positive X) : verified >> X : positive-number;
 )
```

- Note nothing above the bar
- If ‘(is-positive X)’ is in the list of assumptions,
  - ‘X : positive-number‘ is proven!

Counter

- Basically include a '(is-positive X)' guard ...

```
(define make-positive
  { number --> positive-number }
    X -> X where (is-positive X)
    X -> (error "~A is not a positive number" X))
```

- Typechecker 'sees' the (is-positive X) and rubber-stamps 'make-positive'.

Counter

- Parsing a string to a number using Shen's own parser . . .

```
(defcc <digits>
  <digit> <digits> := [<digit> | <digits> ];
  <digit> := [<digit>])

(defcc <trimmed-digits>
  <whitespaces> <digits> <whitespaces> := <digits>;
  <digits> <whitespaces> := <digits>;
  <whitespaces> <digits> := <digits>;
  <digits> := <digits>)

(compile (function <trimmed-digits>)
         (string->bytes "  123 ")) =>
   [1 2 3]
```

Counter

- Run the parser

```
(define number-string-p
  X -> (let Result
         (trap-error
           (compile
            (function <trimmed-digits>)
            (shen.string->bytes X))
           (/. E (do (pr E) (fail))))
         (not (= Result (fail)))))

(declare number-string-p [string --> boolean])
```

- Note the type signature after the fact.
- Tell the typechecker the type of dynamic code!

Counter

- Now add the final rule to the datatype . . .

```
(datatype positive-number
    ....
    ________________________________
    (number-string-p X) : verified >>
    X : (readable string);
)
(define make-readable
  { string --> (readable string) }
  X -> X where (number-string-p X)
  X -> (error "~A is not a number" X))
```

- Everything along the chain is typechecked!

JSON Typing

- JSON -> Internal Representation

```
{                                [object
 auth:                            (@p "auth"
  {                                 [object
    user: "someuser",                 (@p "user" "someuser")
    pass: "somepass"                  (@p "pass" "somepass")
  }                                 ])
 first: "Joe",                    (@p "first" "Joe")
 last: "User",                    (@p "last" "User")
 phone: [314 555 5555]            (@p "phone" [314 555 5555])
}                                ]
```

JSON Typing

- Internal representation -> type

```
[object
 (@p "auth"
   [object
     (@p "user" "someuser")
     (@p "pass" "somepass")])
 (@p "first" "Joe")
 (@p "last" "User")
 (@p "phone" [314 555 5555])]
```

```
(object
 ((kv (auth *
  (object
    ((kv (user * string))
     (kv (pass * string))))))
 ((kv (first * string))
 ((kv (last * string))
  (kv (phone * (list number)))))))
```

JSON Typing

- Iterating over '[object . . .]'

```
(datatype object-iterator
   ...
   let Separated (shen.cons_form
                   (separate (shen.decons XS)))
   Separated : KVS;

   ________________________
   [object | XS] : (object KVS);
 )
 (define separate
   [] -> []
   [X] -> [end X]
   [X | XS] -> [X , | (separate XS)]
   )

[object a b c] => [ a , b , end c]
```

JSON Typing

- Iterating over '[object ...]'

```
(datatype object-iterator
 ...

 X  : KV;
 XS  : KVS;
 ======================
 [X , | XS] : (KV KVS);

 X  : KV;
 ======================
 [end X] : KV;
)
```

```
[object a b c] => (object (a-type (b-type (c-type))))
```

JSON Typing

- Key value datatype:

```
(datatype kv
  X : (A * B);
  ==============
  [kv X] : (kv (A * B));
  )
```

```
[kv (@p "hello" 1)] => (kv (string * number))
```

JSON Typing

- Datatype for ‘user‘:

```
(datatype user-type
   X : (string * B);
   X : (user * B) >> P;
   ________________________________________________
   (= user (string->symbol (fst X))) : verified >> P;)
```

- Function to create ‘user‘:

```
(define make-user
 { (string * A) --> (kv (user * A)) }
 X -> [kv X] where (= user (string->symbol (fst X))))
```

JSON Typing

- Generate the ‘user‘ key types & functions:

```
(define key-type
  Key -> (string->symbol (make-string "~A-type" Key)))
(define key-maker
  Key -> (string->symbol (make-string "make-~A" Key)))

(define register-key
  Key ->
   (let Type (key-type Key)
        Func (key-maker Key)
     ...
     )
   )

(register-key "user"): Type: user-type, Func: make-user
```

JSON Typing

- Generate the ‘user‘ key types & functions:

```
(define register-key
  Key ->
   (let ...
    (do
     (eval
      [datatype Type
        X : [string * B];
        X : [Key * B] >> P;
        ________________________________________________
        [= Key [string->symbol [fst X]]] : verified >> P;]
      )
     ...)))
```

JSON Typing

- Generate the ‘user‘ key types & functions:

```
(define register-key
  Key ->
   (let ...
     (do
      ...
      (eval
       [define Func
         FArg ->
          (shen.cons_form [kv FArg])
             where [= Key [string->symbol [fst FArg]]]
        ])
      ...)))
```

JSON Typing

- Generate the ‘user‘ key types & functions:

```
(define register-key
  Key ->
   (let ...
     (do
      ...
      (declare Func [[string * A] --> [kv [Key * A]]])
      nil)))
```

JSON Typing

- Generate all the keys

```
[object
  (@p "auth"
    [object
      (@p "user" "someuser")
      (@p "pass" "somepass")])
  (@p "first" "Joe")
  (@p "last" "User")
  (@p "phone" [314 555 5555])]

(map
  (function register-key)
  [auth user pass first last phone])
```

JSON Typing

- Need to translate

```
[object
  (@p "auth"
    [object
      (@p "user" "someuser")
      (@p "pass" "somepass")])]
  =>
[object
   (make-auth (@p "auth
      [object
        (make-user (@p "user" "someuser"))
        (make-pass (@p "pass" "somepass"))]))]
```

JSON Typing

- Need a macro!

```
(defmacro my-awesome-obj-macro
  [obj-macro [cons object Pairs]] ->
    (shen.cons_form [object |
      (map
       (function obj-macro-helper)
       (shen.decons Pairs))]))

(define obj-macro-helper
  [@p X Y] ->
    [(key-maker X)
     [@p X (obj-macro-helper Y)]]
  [object PS] -> (obj-macro [object PS])
  XS -> (map (function obj-macro-helper) XS)
        where (cons? XS)
  X -> X)
```

JSON Typing

- Invoking

```
(obj-macro [object (@p "auth" ...) ..])
```

- And it works!

```
(object
 ((kv (auth *
  (object
    ((kv (user * string))
     (kv (pass * string))))))
 ((kv (first * string))
 ((kv (last * string))
  (kv (phone * (list number)))))))
```

JSON Typing

- Further work:
  - Equality?
  - Subtyping?
  - Row polymorphism?

JSON Typing

- Type level equality!

```
(datatype object-iterator
  ...
  let ListA (kv->list (shen.decons A))
  let ListB (kv->list (shen.decons B))
  if (and (= (length ListA) (length ListB))
          (same-keys-p ListA ListB))
  ________________________________
  X : (object B) >> Y : (object A);
  )

(define same-keys-p
  [] _ -> true
  [X | XS] B -> (if (element? X B)
                    (same-keys-p XS B)
                    false))
```

JSON Typing

- Test it:

```
(define works?
  { (object ((kv (key-a * number))
             (kv (key-b * number)))) -->
    (object ((kv (key-b * number))
             (kv (key-a * number)))) }
  X -> X)
```

- It works!

```
(map (function register-key) [key-a key-b])
> (works? (make-object [object (@p "key-a" 1)
                              (@p "key-b" 1)]))
[object ..] : (object ((kv (key-b * number))
                       (kv (key-a * number))))
```

Resources

- The Shen website - http://www.shenlanguage.org/
- The Book Of Shen - http://www.shenlanguage.org/tbos.html
- Shen Google Group - https://groups.google.com/group/qilang?hl=en

A Brief Interlude . . .

- Type Error . . .

```
(+ 1 "hello")
 => type error
```

- Shen has a type level debugger!

```
(spy +)
```

A Brief Interlude

- A type debugging session ...

```
(+ 1 "hello") =>
___________________________ 3 inferences
?- ((+ 1) "hello") : Var2

__________________________ 12 inferences
?- (+ 1) : (Var5 --> Var2)
```

A Brief Interlude

- Type debugging continued ...

```
___________________________________17 inferences
?- + : (Var7 --> (Var5 --> Var2))

___________________________________20 inferences
?- 1 : number

___________________________________24 inferences
?- "hello" : number

>
type error
```

Side Effecting Type Declarations

- Debug output in type declaration

```
(datatype positive-number

   if (do (output "I'm here!") (is-positive X))
   -------------------
   X : positive-number;)
```

Interop

- Serializing/de-serializing

```
(define native-map->shen
  _ N     -> N    where (isNumber N)
  ...
  Lang O ->
     (let PairsOrEmpty (nativePairs O)
        (if (absvector? PairsOrEmpty)
            (vector-map
               (/. X
                  (with-snd
                    (function (native-map->shen Lang))
                    X))
               (nativePairs O))
            PairsOrEmpty))
        where (isObject O)
  _ Wut  -> (error "Unrecognized: ~A" Wut))
```

Interop

- Convert between Ruby/JSON to Shen

```
{:x => [1,2,3],
 :y => {:ya => {},
         :yb => nil      (@v
        }           =>    (@p "x" [1 2 3])
}                         (@p "y" (@v
{x: [1,2 3],                        (@p "ya" (@v <>))
 y: {ya: {},                        (@p "yb" nil))))
     yb: null
    }
}
```

Interop

- On the Ruby side...

```
class << shen
  def isObject(x)
    x.is_a?(Hash)
  end
end
```

- On the JavaScript side ...

```
Shen.defun("isObject",1,function(args){
  // Yes, I know this is buggy!
  return typeof(args[0]) === "object";
});
Shen.defun("isObject",1,function(args){
  return _.isObject(args[0]);
});
```

Interop

- Calling Shen from Ruby/JavaScript ...

```
shen.send(
  "native-map->shen",
  "rb",
   {:x => [1,2,3],
    :y => {:ya => {},
           :yb => nil
          }
   })
```

```
Shen.call_by_name(
  "native-map->shen", [
   "js",
   {x: [1,2 3],
    y: {ya: {},
        yb: null
       }
   }]);
```

Interop

- The FFI is not standardized
- This is good!
- Ports can do whatever they want
  - Ruby coerces Arrays to Shen lists
  - JavaScript does not, but has equality issues
  - Cyclomatic complexity ensues
- Equal representation is hard and annoying!

JSON Typing

- Type check at run-time!

```
(define user-input-key-value ->
   (let Key (input)
        Value (input)
        Pair (@p (make-string "~A" Key) Value)
        Form [object | [(helper Pair)]]
     (shen.typecheck (shen.cons_form Form) A)))
```

```
> (user-input-key-value)
phone
[314 555 5555]
=>
[object [kv [phone * [list number]]]]
```