Shen Trick Shots

Aditya Siram

May 27, 2016

Outline

Overview

- A Lisp
- Pattern matching
- Optional Types
- Built in YACC

Lenses

- Feature the YACC parser
- Functional updates of a JSON structure
- Yak shave some lenses

Lenses

- Getting

```
(from-json [get a-key 0] "{ \"a-key\" : [1,2,3,4] }")
           ^^^^^^^^^^^^^^

  => 1
```

- Setting

```
((from-json [set a-key 0] "{ \"a-key\" : [1,2,3,4] }")
            ^^^^^^^^^^^^^^
            (+ 1))
  => [json.object [(@p a-key [2 2 3 4])]]
```

Lenses

- Tokenized by Shen's own reader!

```
(read-from-string "{ \"a-key\" : [1,2,3,4] }")
=> [{ "a-key" : [cons 1 [cons , [cons 2
                  [cons , [cons 3 [cons ,
                     [cons 4 []]]]]]]] }]
```

- Then built-in parser takes over

```
(compile <object>
  (compile <uncons>
    (...)))
 => [object (@p a-key [1 2 3 4])]
```

Lenses

```
(defcc <uncons>
  [cons X Xs] <uncons> := [(eval [cons X Xs]) | <uncons> ] ;
  X <uncons> := [X | <uncons> ];
  X := [X])
```

Lenses

- Notice how much this ...

```
(defcc <object>
  { <members> } := [object | <members> ];
  {}            := [object];
  { }           := [object];)

(defcc <members>
  <pair> , <members> := [<pair> | <members>];
  <pair>             := [<pair>];)

(defcc <pair>
  String : <value> := (@p (intern String) <value>);)

(defcc <array>
  [ <elements> ] := <elements>;
  ...
```

- Looks like . . .

```
object
   {}
   { members }
members
   pair
   pair , members
pair
   string : value
array
   []
   [ elements ]
```

Lenses

- A lens for objects

```
(define object-lens
  Key [object | KVs] ->
    (@p (get-key Key KVs)
        (set-key Key KVs)))
```

- `(set-key ...)` is curried!

```
(set-key Key KVs)
  == (/. Object V (set-key Key KVs Object V))
```

- A lens to the 'a-key' key

```
(object-lens a-key)
```

Lenses

- A lens for arrays

```
(define array-lens
  Index Array ->
    (@p (get-index Index Array)
        (set-index Index Array)))
```

- A lens to the 3rd element

```
(array-lens 2)
```

Lenses

- Combine two lenses

```
(define compose
   Lens1F Lens2F Json ->
     (let Lens1 (Lens1F Json)
          Lens2 (Lens2F (fst Lens1))
          (@p (fst Lens2)
              (/. V ((snd Lens1) ((snd Lens2) V))))))
```

Lenses

- Combine many lenses

```
(define starter-lens
  X -> (@p X (/. V V)))

(define fold-lenses
   [] -> starter-lens
   [Lens | Lenses] ->
      (fold-lenses-helper
         (compose starter-lens Lens)
         Lenses))
```

Lenses

- Run a lens

```
(define modify
   LensF Json G ->
   (let Lens (LensF Json)
        ((snd Lens) (G (fst Lens)))))

(define access
   LensF Json -> (fst (LensF Json)))
```

Lenses

- Adding 1 to the first element

```
{ "a-key" : [  1  ,2,3,4]}
              ^^^^^

(modify (fold-lenses [(object-lens a-key)
                      (array-lens  0    )])
                     (+ 1))
```

Lenses

- Add 1 to a deeply-nested element

```
{ "a-key" : [1,2, { "another-key" : [3,  4  ,5,6]},7]}
                                        ^^^^^

(modify (fold-lenses [(object-lens a-key      )
                      (array-lens  2          )
                      (object-lens another-key)
                      (array-lens  1          )])
                     (+ 1))
```

Lenses

- The UI is messy, what I want is:

```
[set a-key 2 another-key 1]
=> (modify (fold-lenses [(object-lens a-key      )
                          (array-lens  2          )
                          (object-lens another-key)
                          (array-lens  1          )])
                         ...)
```

Lenses

- Describe the composition as a grammar!

```
(defcc <action>
  set <chain-lenses> :=
      ((function modify) (fold-lenses <chain-lenses>));
  get <chain-lenses> :=
      ((function access) (fold-lenses <chain-lenses>));)

(defcc <chain-lenses>
   <lens> <chain-lenses> := [<lens> | <chain-lenses>];
   <lens>                := [<lens>];)

(defcc <lens>
   X := (array-lens X) where (number? X);
   X := (object-lens X) where (symbol? X);)
```

Lenses

- Putting it all together:

```
(define from-json
  Path JsonString ->
    ((compile <action> Path)
     (compile <object>
     (compile <uncons> (read-from-string JsonString)))))
```

Lenses

- Given the JSON

```
{ "a-key" : [1,2,{ "another-key" : [3,4,  5  ,6] },7]}
                                          ^^^^^
```

- Add 1 to 5

```
((from-json
    [set a-key 2 another-key 2]
    "{\"a-key\":[1,2,{\"another-key\":[3,4,5,6]},7]}")
    (+ 1))
```

- Results in ...

```
[object [
   (@p a-key [1 2 [object [
                    (@p another-key [3 4  6  6])]
                                      ^^^^^
                  ] 7])]]
```

Coins - Typed

- Initial glance at the type system
- Debugging at the type level
- Inserting coins into a coin store

Coins - Typed

- Typed coin store example

```
(insert-coin penny)
=> [penny] : (list coin)

(insert-coin dime)
=> [penny dime] : (list coin)
```

Coins - Typed

- Structure of a Shen datatype

```
(datatype ....
   things-that-need-to-be-true;
   ____________________________
   want-to-prove;

   things-that-need-to-be-true;
   ____________________________
   want-to-prove;
   ...
)
```

Coins - Typed

- Coin type

```
(datatype coin

  ___________
  penny : coin;

  ___________
  nickel: coin;

  ___________
  dime : coin;

  ___________
  quarter: coin;
  ... )
```

- Roughly the same as

```
data Coin = Penny | Nickel | Dime | Quarter
```

Coins - Typed

- Types for storing

```
____________________
*store*: (list coin);

X : A;
______________
(value X): A;

Y : A;
_____________
(set X Y) : A;)
```

Coins - Typed

- Inserting into global store

```
(define insert-coin
  { coin --> (list coin) }
  Coin -> (set *store* (append (value *store*) Coin)))
```

Coins - Typed

- Running

```
(set *store* [])
=> []
(insert-coin penny)
=> type error
```

Coins - Typed

- Step through the typechecker

  ```
  (spy +)
  ```

- Stepping session

  ```
   _______________________________________ 3 inferences
  ?- (define insert-coin ... ) : Var2
  >
  ________________________________________ 23 inferences
  ?- &&Coin : coin

  1. &&Coin : Var10
  2. insert-coin : (coin --> (list coin))
  ```

...

Coins - Typed

- Current expression

```
(set *store* (append (value *store*) Coin))
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

- Step session

```
__________________________________________ 90 inferences
?- ((append ...) &&Coin) : (list coin)

1. &&Coin : coin
2. insert-coin : (coin --> (list coin))
...
```

Coins - Typed

- Current expression

```
(set *store* (append (value *store*) Coin))
                                     ^^^^
```

- Step session - contradiction! `(list coin) !== coin`

```
_______________________________________ 156 inferences
?- &&Coin : (list coin)

1. &&Coin : coin
2. insert-coin : (coin --> (list coin))

>
type error in rule 1 of insert-coin
```

Coins - Typed

```
(define insert-coin
  { coin --> (list coin) }
  Coin -> (set *store* (append (value *store*) [Coin])))
                                               ^^^^^^
```

Coins - Typed

- Datatypes also take side-conditions

```
(datatype coins
  if (= 1 1)
  ___________
  penny : coin;
  ...)
```

Coins - Typed

- Which run arbitrary code!

```
(datatype coins
  if (do (output "Hurr-durr, I'm a penny!~%") true)
  ___________
  penny : coin;
  ...)
```

- Type level println!

```
(insert-coin penny)
 => "Hurr-durr, I'm a penny!"
    [penny] : (list coin)
```

Coins - Typed

- Ad hoc hole driven development!

```
(datatype <<HOLE>>

  if (do (output (make-string "<<HOLE>> : ~A~%" X)) true)
  ______________
  <<HOLE>> : X;)
```

Coins - Typed

- Load with typechecking

```
(define insert-coin
  { coin --> (list coin) }
  Coin -> (set *store* <<HOLE>>))
=> <<HOLE>> : [list coin]
   insert-coin : (coin --> (list coin))
```

- Don't run this or you'll get:

```
(insert-coin penny)
=> [<<HOLE>>]
```

Coins - Untyped

- Use the typechecker for runtime reflection
- Grow a datatype at runtime!

Coins - Untyped

- Add and make coins.

```
(with-store penny)
=> "penny is not a coin."
(with-store [make penny])
=> type#coin
(with-store penny)
=> [penny]
(with-store [remove penny])
=> type#coin
(with-store penny)
=> "penny is not a coin"
```

Coins - Untyped

- Use the typechecker for runtime reflection!

```
(define with-store
  ...
  Coin ->
    (if (= (shen.typecheck Coin coin) coin)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
       ( ... )
      (make-string "~A is not a coin." Coin)))
```

- A simple example

```
(shen.typecheck "hello world" string)
=> string
(shen.typecheck "hello world" number)
=> false
```

Coins - Untyped

- Add or remove from the global list of coin types

```
(define to-coin
  make Coin     -> (append (value *coins*) [Coin])
  remove Coin   -> (remove Coin (value *coins*)))
```

- Eval a fresh datatype with only those types!

```
(define with-store
  [Action Coin] ->
   ...
    (do
      ...
      (set *current-datatype* (create-datatype NewCoins))
      (eval (value *current-datatype*)))))
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  Coin -> ...)
```

Coins - Untyped

- Creating the datatype

```
(define create-datatype
  Coins ->
    (append
      [datatype coin]
      (mapcan (/. Coin [

                       ___________
                       Coin : coin;
                     ])
            Coins)))
```

Coins - Untyped

- Example Run

```
(create-datatype [penny dime])
=> [datatype coin

      __________
      penny : coin;

      __________
      dime : coin;]
```

Coins - Untyped

- Examine datatype at runtime!

```
(value *current-datatype*)
  => [datatype coin

      __________
      penny : coin;

      __________
      dime : coin;]
```

API - Dump

- Use built-in functions to inspect source code.
- DIY Hoogle.

API - Dump

- Don't need to give typecheck a concrete type!

  ```
  (shen.typecheck 1 A)
  => number
  ```

- `A` is unified with the type

API - Dump

- An 'undefined' type

```
(datatype undefined
    _______________
    ??? : X;
)
```

API - Dump

- Some sample functions with fake datatypes

```
(define a-b-c
  { a --> b --> c }
  _ _ -> ??? )

(define b-c-d
  { b --> c --> d }
  _ _ -> ??? )

(define c-d
  { c --> d }
  _ -> ??? )

(define a-e
  { a --> e }
  _ -> ??? )
```

API - Dump

- Extract the type signatures!

```
(dump "test.shen")
=> [[a-b-c [a --> [b --> c]]]
    [b-c-d [b --> [c --> d]]]
    [c-d   [c --> d]]
    [a-e   [a --> e]]
    [b-f   [b --> f]]]
```

- Roll you own semver!

API - Dump

- Extraction code - by Shen's author, adapted from mailing list post.

```
(define dump
  Shen ->
     (let Defs (mapcan (function def) (read-file Shen))
          Types (map get-sig Defs)
       Types))

(define def
  [define F | _] -> [F]
  _ -> [])

(define get-sig
  Def -> [Def (shen.typecheck Def (protect A))])
```

API - Dump

- Hoogle style search!

```
(find-signature [a --> b --> X]
                (dump "Type-Tetris-Test.shen"))
=> [[a-b-c [a --> b --> c]]]
```

API - Dump

- Generate a grammar at runtime.

```
(define find-signature
  Signature ... ->
     (let ...
          SigParserAST (append
                          [defcc SigParserName]
                            Signature [:= true;]
                            [_ := false;])
          _ (eval SigParserAST)
        (...)))
```

- Generated grammar

```
(defcc Parser12345
  a --> b --> X := true;
  _ := false;)
```

Rank N Types

- Emulate Rank N Types in Shen!
- This fails to typecheck

```
(define foo
   { (A --> A) --> (number * symbol) }
   F -> (@p (F 1) (F a)))
```

- The type variable `A` needs to be determined by application.

Rank N Types

- Neat hack by Shen author, Mark Tarver.
- This works!

```
(define rank-n-stein
  {(forall A (A --> A)) --> (number * symbol)}
  F -> (@p (F 1) (F a)))
```

Rank N Types

- Substitute out free variable in forall

```
let C (subst (gensym &&) A B)
X : C;
_____________________________
X : (mode (forall A B) -);
```

- Mode declaration disallows two way binding (unification)

```
C => (&&12345 --> &&12345)
```

Rank N Types

- Typechecking `(F 1)`
- `(forall A (A --> A)) -> (free-var --> free-var)`
- Type system can now unify `free-var` with `number`
- `(forall A (A --> B)) -> (free-var --> B)`

Rank N Types

- When `forall ...` is in the environment ...
- Replace with `S`.

```
(scheme A B S V);
X : S >> P;
_______________________________
X : (mode (forall A B) -) >> P;
```

Rank N Types

- Recurse over `(A --> A)`
- Build up `(D | E)`.

```
!;
(scheme A B D F);
(scheme A C E F);
_____________________________
(scheme A (B | C) (D | E) F);
```

Rank N Types

- If `A` is found substitute with `V`

```
!;
_________________
(scheme A A V V);
```

- The `!` is a cut. No backtracking.
- In the end, just return:

```
_________________
(scheme A B B _);
```

Rank N Types

- (Very) roughly like:

```
scheme(A [B | C] [D | E] F) :-
   scheme(A B D F);
   scheme(A C E F).
scheme (A B B _).
scheme (A A V V).
```

Rank N Types

- The Book Of Shen (1st & 2nd edition)
- The Shen mailing list
- Questions?