

# Typed routing with continuations

Daniel Patterson

Position Development  
[positiondev.com](http://positiondev.com)

<https://dbp.io/static/fn-continuations-haskell-meetup-2016.pdf>

# What are Fn routes?

```
1 app :: Ctxt -> IO (Maybe Response)
2 app ctxt =
3     route ctxt
4         [end                                     ==> indexH
5         ,path "add" // segment // segment       ==> addH
6         ,path "add" // param "a" // param "b"    ==> addH
7         ]
8 indexH :: Ctxt -> IO (Maybe Response)
9 indexH ctxt = ...
10
11 addH :: Ctxt -> Int -> Int -> IO (Maybe Response)
12 addH ctxt a b = ...
```

# Let's try without continuations.

```
1 data UriPart = Constant Text | Segment | Param Text | End
2 type UriPattern = [UriPart]
3
4 pattern ==> hndlr = \url -> .. produce response
```

Type of ==> is...? UriPattern -> **Handler** -> Uri -> Response

T1 -> T2 -> ... -> Response

# What goes wrong?

`pattern = [p1, p2, ... pn]`

`Handler :: T1 -> T2 -> ... -> Tn -> Response`

n must be the same, so the  
type of the handler depends  
on the value of the pattern.

# Review: polymorphism

```
id_int :: Int -> Int
id_int i = i
```

```
id_str :: String -> String
id_str s = s
```

```
id :: a -> a
id a = a
```

```
id 10      :: Int    -- a is Int
id "Hello" :: String -- a is String
```

Type variables are “instantiated”  
with concrete types

# Continuations!

Or, type dependency with plain-old-polymorphism.

```
add :: Int -> Int -> Int
add a b = a + b
```

Becomes

```
add_k :: (Int -> a) -> Int -> Int -> a
add_k k a b = k (a + b)
```

Can use as:

```
add :: Int -> Int -> Int
add a b = add_k id
```

```
add_print :: Int -> Int -> IO ()
add_print a b = add_k print
```

etc...

# Related problem: printf

`printf :: FormatString -> Arg -> ... -> Arg -> IO ()`

Specification of  
number & types of  
args - `Hi %s, %d`

Must match - `String -> Int`

Similar to: must match

`(==>) :: Route -> (Arg -> ... -> Arg -> IO (Maybe Response))`



# printf

Build format string out  
of “combinators”:

```
main = do printf (c"Hi " % s % c", " % d) "there" 10
          printf (c"foo" % d) 10
```

```
> Hi there, 10
> foo10
```

```
printf (c"Hi " % s % c", " % d) :: String -> Int -> IO ()
printf (c"foo" % d) :: Int -> IO ()
```

Polymorphism and continuations used to make this work.



# printf

Format will take continuation of where to send formatted string.

```
printf :: ((String -> IO ()) -> a) -> a
printf format = format putStrLn
```

Type variable accomplishes dependency. Will instantiate as:

```
IO ()
String -> IO ()
String -> Int -> IO ()
etc.
```

# start with c - constants

```
printf (c"Hello there")
```

```
> Hello there
```

```
printf :: ((String -> IO ()) -> a) -> a
```

```
"Hello there"  
c :: String -> ((String -> IO ()) -> IO ())
```

`(String -> IO ()) -> IO ()` vs. `(String -> b) -> b`

a particular continuation  
(that can do IO)

any continuation  
(more flexible)

# c - implementing it

```
c :: String -> ((String -> b) -> b)
```

```
c str = \k -> k str
```

Let's see the types get instantiated:

```
printf :: ((String -> IO ()) -> a) -> a
```

```
printf (c"hello there") :: a = IO ()
```

But why?

```
printf expects: ((String -> IO ()) -> a)
(c"hello there") :: ((String -> b) -> b)
```

`b = IO ()` `b = a` so `a = IO ()`

# c - using it

```
printf (c"hello world")
```

```
(c"hello world") putStrLn
```

```
((\str -> \k -> k str) "hello world") putStrLn
```

```
(\k -> k "hello world") putStrLn
```

```
putStrLn "hello world"
```

# d - integer format

```
printf d 10
```

```
> 10
```

```
printf :: ((String -> IO ()) -> a) -> a
```

try to write like c (implementation first):

```
d = \k -> ... k (show (int :: Int))
```

where does this come from?

# d - integer format

```
printf d :: Int -> IO ()
```

```
d = \k -> ... k (show (int :: Int))
```

let's learn from c:

```
printf (c"hello there") :: IO ()
```

```
(c"hello there") :: ((String -> b) -> b)
```

```
c str = \k -> k str
```

we want: `Int -> IO ()`


```
d = \k -> \int -> k (show (int :: Int))
```

# d - using it

```
printf d 10
```

```
d putStrLn 10
```

```
(\k -> \int -> k (show (int :: Int)) putStrLn 10
```



```
(\int -> putStrLn (show int :: Int)) 10
```



```
putStrLn (show 10 :: Int)
```

# s - string format

pretty much the same as d:

```
s :: (String -> a) -> String -> a
```

```
s = \k -> \str -> k str
```

```
printf s "hello there"
```

```
printf d 100
```

```
printf (c"hello")
```

```
> hello there
```

```
> 100
```

```
> hello
```



# Combining them

think in terms of continuations (rather than types)

$f1 \% f2 = \text{\texttt{\textbackslash k}} \rightarrow \dots$

where combined, formatted string will go

can use  $f1$  and  $f2$  by passing them continuations:

$f1 \% f2 = \text{\texttt{\textbackslash k}} \rightarrow f1 (\text{\texttt{\textbackslash s1}} \rightarrow \dots f2 (\text{\texttt{\textbackslash s2}} \rightarrow \dots))$

string from  $f1$

string from  $f2$

$k$  needs combined output, i.e.  $s1 ++ s2$ :

$f1 \% f2 = \text{\texttt{\textbackslash k}} \rightarrow f1 (\text{\texttt{\textbackslash s1}} \rightarrow f2 (\text{\texttt{\textbackslash s2}} \rightarrow k (s1 ++ s2)))$

# (yes, that works)

but let's check the types anyway.

A format has type:  $(\text{String} \rightarrow a) \rightarrow F[a]$  (not a real type)

=

$T1 \rightarrow T2 \rightarrow \dots \rightarrow a$

We want something like this “type”:

```
(%) :: ((String -> a) -> F[a])  
      -> ((String -> b) -> F[b])  
      -> (String -> c) -> F[c]
```

# type-checking %

$f1 \% f2 = \text{\texttt{\textbackslash k}} \rightarrow f1 (\text{\texttt{\textbackslash s1}} \rightarrow f2 (\text{\texttt{\textbackslash s2}} \rightarrow k (s1 ++ s2)))$

$(\%) :: ((String \rightarrow a) \rightarrow F[a])$   
 $\rightarrow ((String \rightarrow b) \rightarrow F[b])$   
 $\rightarrow (String \rightarrow c) \rightarrow F[c]$

becomes:

$(\%) :: ((String \rightarrow a) \rightarrow c)$   
 $\rightarrow ((String \rightarrow b) \rightarrow a)$   
 $\rightarrow (String \rightarrow b) \rightarrow c$

```
type Str = String
```

# using % (types)

let's see how printf (d % s) types as `Int -> String -> IO ()`

```
(%) :: ((Str -> a) -> c)      d :: (Str -> f) -> Int -> f
      -> ((Str -> b) -> a)
      -> (Str -> b) -> c
```

```
(%) :: ((Str -> f) -> Int -> f)  s :: (Str -> g) -> Str -> g
      -> ((Str -> b) -> f)
      -> (Str -> b) -> Int -> f
```

```
(%) :: ((Str -> Str -> g) -> Int -> Str -> g)
      -> ((Str -> g) -> Str -> g)
      -> (Str -> g) -> Int -> Str -> g
```

```
(d % s) :: (Str -> g) -> Int -> Str -> g    so g = IO () and
printf :: (Str -> IO ()) -> a -> a           a = Int -> Str -> IO ()
```

# using % (values)

```
printf (d % s) 10 "hi"
```

```
(d % s) putStrLn 10 "hi"
```

```
(\k -> d (\s1 -> s (\s2 -> k(s1++s2)))) putStrLn 10 "hi"
```




```
(d (\s1 -> s (\s2 -> putStrLn (s1 ++ s2)))) 10 "hi"
```

```
(\k -> \int -> k (show int))  
  (\s1 -> s (\s2 -> putStrLn (s1 ++ s2))) 10 "hi"
```

```
(\int -> (\s1 -> s (\s2 -> putStrLn (s1 ++ s2)))  
  (show int))  
  10 "hi"
```

# using % (values)

```
(\int -> (\s1 -> s (\s2 -> putStrLn (s1 ++ s2)))  
  (show int))  
10 "hi"
```



```
(\int -> s (\s2 -> putStrLn ((show int) ++ s2)))  
10 "hi"
```

```
(\int -> (\k -> \str -> k str)  
  (\s2 -> putStrLn ((show int) ++ s2)))  
10 "hi"
```

```
(\int -> \str ->  
  (\s2 -> putStrLn ((show int) ++ s2)) str)  
10 "hi"
```

# using % (values)

```
(\int -> \str ->  
  (\s2 -> putStrLn ((show int) ++ s2)) str)  
10 "hi"
```

```
(\int -> \str -> putStrLn ((show int) ++ str))  
10 "hi"
```

```
putStrLn ((show 10) ++ "hi"))
```

# complete printf

```
c str = \k -> k str
```

```
d = \k -> (\int -> k (show (int::Int)))
```

```
s = \k -> (\str -> k (str::String))
```

```
printf format = format putStrLn
```

```
(%) f1 f2 = \k -> f1 (\s1 -> f2 (\s2 -> k (s1 ++ s2)))
```

because inference will figure types out...



# How fn routing differs

- Overall, very similar.
- Routes can fail to match, so combinators return Maybe and short circuit if Nothing.
- A Request is threaded through, so that segment can remove matched portions.
- Building a single argument function that expects to get a multi-argument handler, rather than a multi-argument function. This means // does function composition (between a function that will pass some arguments to handler and another that will pass more).

# Other approaches

- Type families: build type level list, use it to construct function of right arity (Spock does this).
- Define typed DSL (needs GADTs, I believe) for patterns, write interpreter for it that while running consumes a Request. In OCaml: <http://rgrinberg.com/blog/2014/12/13/primitive-type-safe-routing/>

# Questions?