Stephanie Weirich

University of Pennsylvania
https://github.com/sweirich/dth
@fancytypes

Dependent Types in Haskell

What is Dependent Type Theory?

- Originally, logical foundation for mathematics (Martin-Löf)
- Now, basis of modern proof assistants such as Coq, Agda, and Lean
- Connected to programming through the Curry-Howard isomorphism: propositions are types, proofs are programs

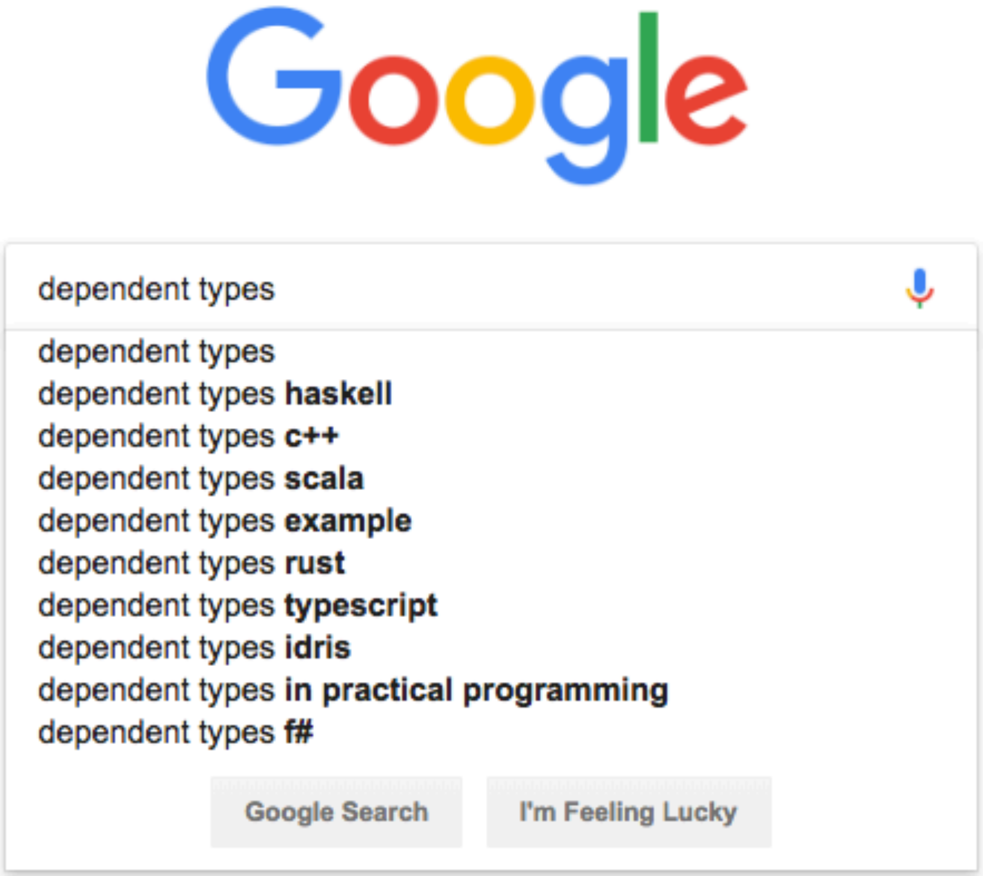
Π

What is Haskell?

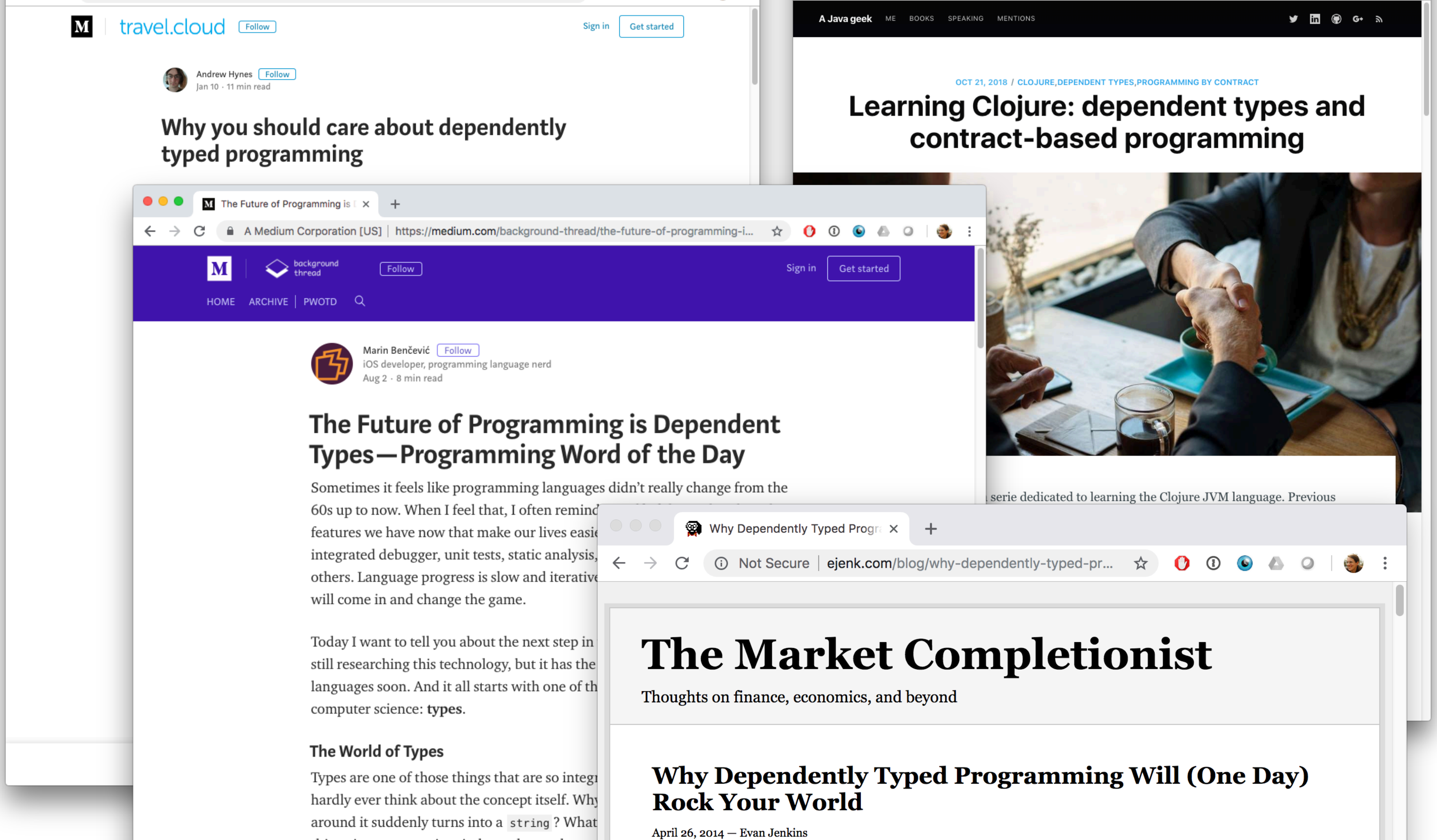
- Originally, research programming language (Hudak, Wadler, Peyton Jones, et al. 1990)
- Now, research programming language with users (industrial users, researchers, educators, hobbyists…)
- Influential
  - New languages based on Haskell (Elm, PureScript, Eta, Frege)
  - Existing languages adopt ideas from Haskell (HKT, type classes, purity, ADTs, …)

Dependent types in Haskell?

Dependent types and programming


dependent types

dependent types
dependent types **haskell**
dependent types **c++**
dependent types **scala**
dependent types **example**
dependent types **rust**
dependent types **typescript**
dependent types **idris**
dependent types **in practical programming**
dependent types **f#**



travel.cloud Follow Sign in Get started

Andrew Hynes Follow
Jan 10 · 11 min read

Why you should care about dependently typed programming

A Java geek ME BOOKS SPEAKING MENTIONS

OCT 21, 2018 / CLOJURE, DEPENDENT TYPES, PROGRAMMING BY CONTRACT

Learning Clojure: dependent types and contract-based programming

serie dedicated to learning the Clojure JVM language. Previous

The Future of Programming is

A Medium Corporation [US] | https://medium.com/background-thread/the-future-of-programming-i...

background thread Follow Sign in Get started

HOME ARCHIVE | PWOTD

Marin Benčević Follow
iOS developer, programming language nerd
Aug 2 · 8 min read

The Future of Programming is Dependent Types—Programming Word of the Day

Sometimes it feels like programming languages didn't really change from the 60s up to now. When I feel that, I often remind features we have now that make our lives easie integrated debugger, unit tests, static analysis, others. Language progress is slow and iterative will come in and change the game.

Today I want to tell you about the next step in still researching this technology, but it has the languages soon. And it all starts with one of th computer science: **types**.

The World of Types

Types are one of those things that are so integr hardly ever think about the concept itself. Why around it suddenly turns into a `string`? What

Why Dependently Typed Progr

Not Secure | ejenk.com/blog/why-dependently-typed-pr...

The Market Completionist

Thoughts on finance, economics, and beyond

Why Dependently Typed Programming Will (One Day) Rock Your World

April 26, 2014 — Evan Jenkins

Dependent Haskell

A set of language extensions for GHC that provides the ability to program *as if* the language had dependent types

```
{-# LANGUAGE DataKinds, TypeFamilies, PolyKinds, TypeInType,
    GADTs, RankNTypes, ScopedTypeVariables, TypeApplications,
    TemplateHaskell, UndecidableInstances, InstanceSigs,
    TypeSynonymInstances, TypeOperators, KindSignatures,
    MultiParamTypeClasses, FunctionalDependencies,
    TypeFamilyDependencies, AllowAmbiguousTypes,
    FlexibleContexts, FlexibleInstances #-}
```

Why Dependent Types?

Domain-specific type checkers

Regular expression capture groups

- Use regexps to recognize and parse a file path
  `"dth/regexp/Example.hs"`
- Return captured results in a dictionary
  - Basename `"Example"`
  - Extension `"hs"`
  - Directories in path `"dth"` `"regexp"`
- Challenge: Type system verifies dictionary access

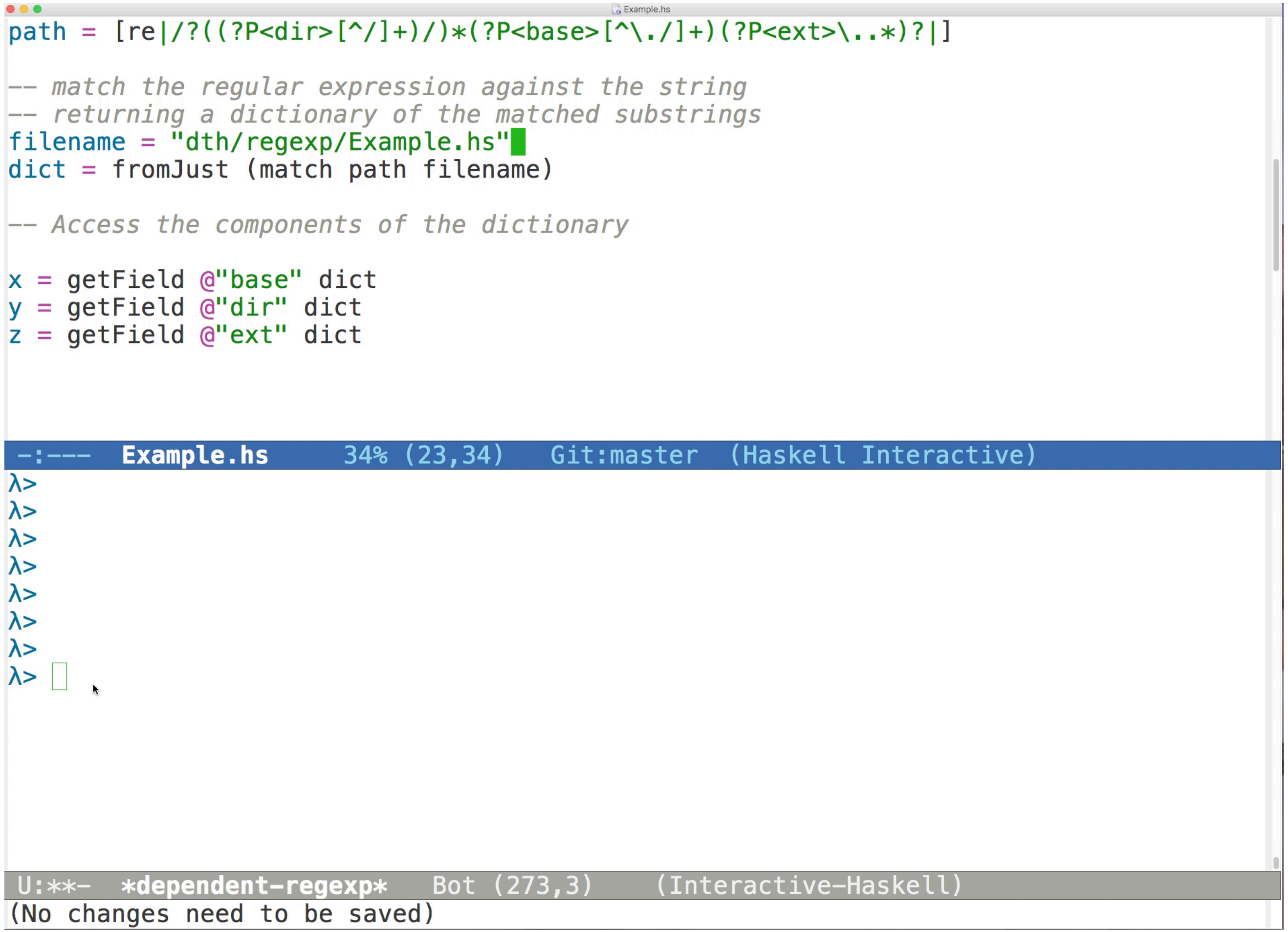
Example: a regexp for parsing file paths

```
/?                          -- optional leading "/"
((?P<dir>[^/]+)/)*          -- any number of dirs
(?P<base>[^\./]+)           -- basename
(?P<ext>\..*)?              -- optional extension
```

Named capture groups marked by `(?P<`*`name`*`>`*`regexp`*`)`

Demo

```
path =
   [re|/?((?P<dir>[^/]+)/)*(?P<base>[^\./]+)(?P<ext>\..*)?|]
filename =
   "dth/regexp/Example.hs"
```



Example.hs

```
path = [re|/?((?P<dir>[^/]+)/)*(?P<base>[^\./]+)(?P<ext>\..*)?|]

-- match the regular expression against the string
-- returning a dictionary of the matched substrings
filename = "dth/regexp/Example.hs"
dict = fromJust (match path filename)

-- Access the components of the dictionary

x = getField @"base" dict
y = getField @"dir" dict
z = getField @"ext" dict
```

```
-:---  Example.hs     34% (23,34)    Git:master   (Haskell Interactive)
```

```
λ>
λ>
λ>
λ>
λ>
λ>
λ>
λ>
```

```
U:**-  *dependent-regexp*    Bot (273,3)     (Interactive-Haskell)
(No changes need to be saved)
```

What are we asking for, when we ask for dependent types?

Four Capabilities of Dependent Type Systems

1. Type computation
2. Indexed types
3. Double-duty data
4. Equivalence proofs

Type Computation

We can use the type system to implement a domain-specific compile-time analysis

How does this work?

```
λ> path =
   [re|/?((?P<dir>[^/]+)/)*(?P<base>[^/.]+)(?P<ext>\..*)?|]
λ> :t path
RE '['("base", Once), '("dir", Many), '("ext", Opt)]
```

Regular expression type includes a "Occurrence Map" computed by the type checker

```
data Occ = Once | Opt | Many
```

How does this work? 1. Compile-time parsing

```
λ> path =
  [re|/?((?P<dir>[^/]+)/)*(?P<base>[^/.]+)(?P<ext>\..*)?|]

λ> :t path
RE '[ '("base", Once), '("dir", Many), '("ext", Opt)]
```

```
> path = ropt (rchar '/')
    `rseq` rstar (rmark @"dir" (rplus (rnot "/")) `rseq` rchar '/')
    `rseq` rmark @"base" (rplus (rnot "./"))
    `rseq` ropt (rmark @"ext" (rchar '.' `rseq` rstar rany))
```

2. Type functions run by type checker

```
-- accepts single char only, captures nothing
rchar  :: Char -> RE '[]
-- sequence  r₁r₂
rseq   :: RE s1 -> RE s2 -> RE (Merge s1 s2)
-- iteration r*
rstar  :: RE s -> RE (Repeat s)
-- marked subexpression
rmark  :: ∀k s. RE s -> RE (Merge (One k) s)
```

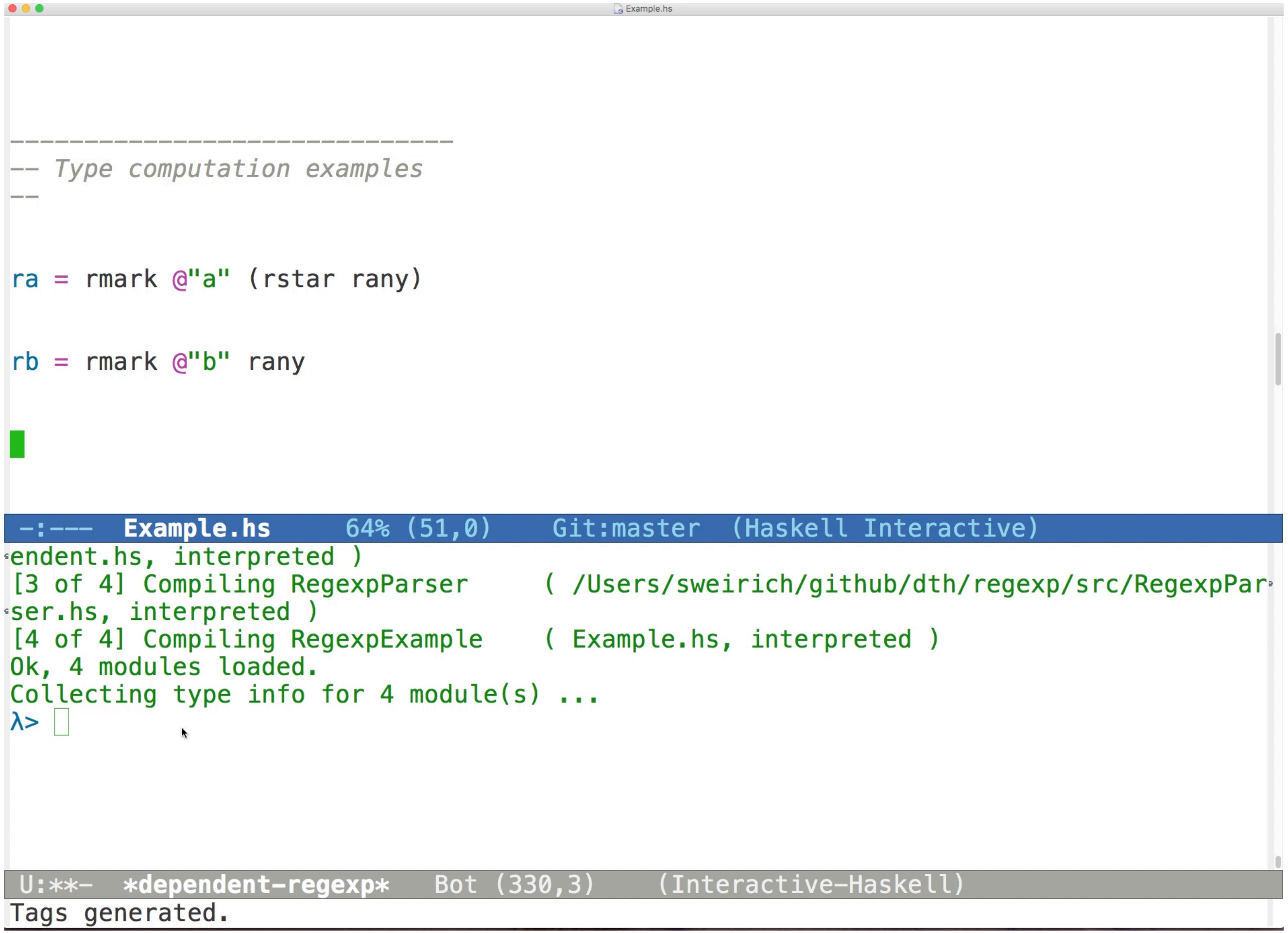
Type functions via type families

```
-- iteration r*
rstar  :: RE s -> RE (Repeat s)

type family Repeat (s :: OccMap) :: OccMap
  where
    Repeat '[]           = '[]
    Repeat ((k,o) : t) = (k, Many) : Repeat t
```

Demo

```
r1 = rmark @"a" (rstar rany)
r2 = rmark @"b" rany
ex1 = r1 `rseq` r2
```



Example.hs

```
-------------------------------
-- Type computation examples
--


ra = rmark @"a" (rstar rany)


rb = rmark @"b" rany


```

```
-:---  Example.hs      64% (51,0)    Git:master  (Haskell Interactive)
```

```
endent.hs, interpreted )
[3 of 4] Compiling RegexpParser     ( /Users/sweirich/github/dth/regexp/src/RegexpPar
ser.hs, interpreted )
[4 of 4] Compiling RegexpExample    ( Example.hs, interpreted )
Ok, 4 modules loaded.
Collecting type info for 4 module(s) ...
λ>
```

```
U:**-  *dependent-regexp*   Bot (330,3)    (Interactive-Haskell)
Tags generated.
```

Indexed types

Type indices constrain values and guide computation

How does this work?

```
λ> :t dict
Dict '['("base", Once),'("dir", Many), '("ext", Opt)]

λ> getField @"ext" dict
Just "hs"

λ> getField @"f" dict
<interactive>:28:1: error:
    • I couldn't find a capture group named 'f' in
          {base, dir, ext}
```

Access resolved at compile time by type-level symbol

Custom error message

Types Constrain Data

```
λ> :t dict
   Dict '['("base", Once),'("dir", Many),'("ext", Opt)]
```

- Know `dict` must be a sequence of entries

```
E "Example" :> E ["dth","regexp"] :> E (Just "hs") :> Nil
```

- Entries do not store keys
  - From type, know `"base"` is **first** entry
  - Field access resolved at ***compile time***

Types Constrain Data with GADTs

```
λ> :t dict
   Dict '['("base", Once),'("dir", Many),'("ext", Opt)]
```

```
data Dict :: OccMap -> Type where
   Nil  :: Dict '[]
   (:>) :: Entry s o -> Dict tl -> Dict ('(s,o) : tl)
```

- Know `dict` must be a sequence of entries

```
E "Example" :> E ["dth","regexp"] :> E (Just "hs") :> Nil
```

Types Constrain Data with Type Families

```
x :: Entry "ext" Opt
x = E (Just ".hs")


data Entry :: Symbol -> Occ -> Type
  where
    E :: OT o -> Entry k o
```

```
type family OT (o :: Occ)
  where
    OT Once = String
    OT Opt  = Maybe String
    OT Many = [String]
```

Double-duty data

We can use the same data in types and at runtime

How does this work?

```
dict :: Dict '['("base", Once),'("dir", Many),'("ext", Opt)]
dict =
 E "Example" :> E ["dth", "regexp"] :> E (Just "hs") :> Nil

λ> print dict
{ base="Example", dir=["dth","regexp"], ext=Just ".hs" }
```

Dependent types: Π

```
showEntry :: Π k -> Π o -> Entry k o -> String
showEntry k o (E x) = showSym k ++ "=" ++ showData o x

showData :: Π o -> OT o -> String
showData  Once = show  :: String -> String
showData  Opt  = show  :: Maybe String -> String
showData  Many = show  :: [String] -> String
```

GHC's take: Singletons

```
showEntry :: Sing k -> Sing o -> Entry k o -> String
showEntry k o (E x) = showSym k ++ "=" ++ showData o x

showData :: Sing o -> OT o -> String
showData SOnce = show
showData SOpt  = show
showData SMany = show
```

```
data instance Sing (o :: Occ) where
   SOnce :: Sing Once
   SOpt  :: Sing Opt
   SMany :: Sing Many
```

Equivalence proofs

Type checker must reason about program equivalence, and sometimes needs help

Working with type indices

```
data RE :: OccMap -> Type  where
   Rempty :: RE '[]
   Rseq   :: RE s1 -> RE s2 -> RE (Merge s1 s2)
   Rstar  :: RE s -> RE (Repeat s)
   …

rseq :: RE s1 -> RE s2 -> RE (Merge s1 s2)
rseq Rempty r2 = r2  -- Merge '[] s2 ~ s2
rseq r1 Rempty = r1
rseq r1 r2     = Rseq r1 r2
```

Working with type indices

```
type family Repeat (s :: OccMap) :: OccMap where
   Repeat '[]           = '[]
   Repeat ((k,o) : t) = (k, Many) : Repeat t
```

```
rstar :: RE s -> RE (Repeat s)
rstar Rempty     = Rempty    -- need: Repeat '[] ~ '[]
rstar (Rstar r) = Rstar r  -- oops!
rstar r          = Rstar r   Could not deduce: Repeat s ~ s
                              from the context: s ~ Repeat s1
```

Need: `Repeat (Repeat s1) ~ Repeat s1`

Not true by definition. But provable!

Type classes to the rescue

```
class (Repeat (Repeat s) ~ Repeat s)
      => Wf (s :: OccMap)
instance Wf '[]                       -- base case
instance (Wf s) => Wf ('(n,o) : s) -- inductive step

rstar :: Wf s => RE s -> RE (Repeat s)
rstar Rempty    = Rempty
rstar (Rstar r) = Rstar r
    -- have: Repeat (Repeat s1) ~ Repeat s1
rstar r         = Rstar r
```

Type classes to the rescue

```
class (Repeat (Repeat s) ~ Repeat s,
       s ~ Alt s s,
       Merge s (Repeat s) ~ Repeat s)
       => Wf (s :: OccMap)
instance Wf '[]                         -- base case
instance (Wf s) => Wf ('(n,o) : s) -- inductive step
```

Summary: Dependent types have a lot to offer

1. Type computation
2. Indexed types
3. Double-duty data
4. Equivalence proofs

Haskell is a good fit for dependent types

- Similarities make integration possible
  - Computation based on polymorphic lambda calculus
  - Type system encourages purity
- Differences tell us about the design space
  - Full language available for programming, many examples in-the-wild
  - Lack of termination analysis discourages proof-heavy use, pushes for new approaches

https://github.com/sweirich/dth

Thanks to: Simon Peyton Jones, Richard Eisenberg, Dimitrios Vytiniotis, Vilhelm Sjöberg, Brent Yorgey, Chris Casinghino, Geoffrey Washburn, Iavor Diatchki, Conor McBride, Adam Gundry, Joachim Breitner, Julien Cretin, José Pedro Magalhães, Steve Zdancewic, Joachim Breitner, Antoine Voizard, Pedro Amorim and NSF

fin