The Influence of Dependent Types

Stephanie Weirich
University of Pennsylvania
How has Dependent Type Theory influenced the design of the Haskell type system?
Dependent Haskell

A set of compiler extensions for the GHC compiler 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 #-}
"What have you done to Haskell?"
Showcase ~10 years of language extensions that conspire to make GHC "dependently-typed"

"If you are interested in dependent types, why Haskell?"
Demonstrate the benefits of studying dependent types in the context of the Haskell ecosystem (Haskell-specific features, different design space, industrial-strength compiler, ready-made user base, awesome collaborators)
Why Dependent Haskell?

Answer: Domain-specific type checkers
A type system for regular expressions

• Task: Use regexp capture groups to recognize a file path and extract its parts
  "dth/popl17/Regexp.hs"
  - Basename "Regexp"
  - Extension "hs"
  - Directories in path "dth" "popl17"

• Return all captured results in a data structure

• Challenge: Type system allows only "sensible" access to the data structure

• http://www.github.com/sweirich/dth/popl17/

Inspiration: Spishak, Dietl, Ernst "A type system for regular expressions"
A regular expression for file paths

```
/\?
  (((?P<d>[^/]+)\//))*
  (?P<b>[^\./]+)
  (?P<e>\.\.*\?)?
```

-- optional / 
-- directories 
-- basename 
-- extension

Caveats:

- Uses Python syntax but captures all strings under a *, not the most recently matched one
- Only named capture groups, not numbered
Demo

path =
    [re|/?((?P<d>[^/]+)*)((?P<b>[^\./]+)(?P<e>\..*)?)|]
filename =
    "dth/popl17/Regexp.hs"
Four Features of Dependently Typed Programs

1. Types compute
2. Indices constrain values
3. Double-duty data
4. Equivalence matters
We can use dependent types to implement a domain-specific compile-time analysis.
Type aware implementation

```haskell
> path =
  [re|/?(?P<d>[^/]+)//(*(?P<b>[^/\.]++)(?P<e>\..*)?|)]

> dict = fromJust (match path "dth/popl17/Regexp.hs")

> :t dict
Dict ['("b", 'Once), '("d", 'Many), '("e", 'Opt)]

> :t path
R ['("b", 'Once), '("d", 'Many), '("e", 'Opt)]
```

DataKinds [Yorgey, Weirich, Cretin, Peyton Jones, Magalhães TLDI 2012]
Type-level symbols [Diatcki, HS 2015]
How does this work? Compile time parsing

```
> path =
[re]/?((?P<d>[/^/]+)/)*(?P<b>[^\./]+)/(?P<e>\..*/)?)
> :t path
R ['("b", 'Once), ('"d", 'Many), ('"e", 'Opt)]
```

```
> path = ralt rempty (rchars "/") `rseq`
  rstar (rmark @"d" (rplus (rnot "/"))) `rseq`
  `rseq` rchars "/"
  `rseq` rmark @"b" (rplus (rnot "./")) `rseq`
  ralt rempty (rmark @"e"
  (rchars "." `rseq` rstar rany))
> :t path
R ['("b", 'Once), ('"d", 'Many), ('"e", 'Opt)]
```

TypeApplications [Eisenberg, Weirich, Hamidhasan, ESOP 2016]
TemplateHaskell [Sheard & Peyton Jones, HW 2002]
--- accepts empty string only
rempy :: R '[]

--- accepts single char only
rchar :: Char -> R '[]

--- alternative \( r_1 | r_2 \)
ralt :: R s1 -> R s2 -> R (Alt s1 s2)

--- sequence \( r_1 r_2 \)
rseq :: R s1 -> R s2 -> R (Merge s1 s2)

--- iteration \( r^* \)
rstar :: R s -> R (Repeat s)

--- marked subexpression
rmark :: \( \forall n \ s. \ R \ s \rightarrow R \ (\text{Merge } '(n,\text{Once}) \ s) \)
Computing with types

data Occ = Once | Opt | Many

Represent maps by lists of pairs, ordered by first component (name of the capture group)

type SM = [(Symbol, Occ)]

type family Merge (s1 :: SM) (s2 :: SM) :: SM where

  Merge s '[] = s
  Merge '[] s = s
  Merge ('(n1, o1):t1) ('(n2, o2):t2) =
    If (n1 ::== n2) ('(n1, 'Many) : Merge t1 t2)
    (If (n1 ::<= n2)
     ('(n1, o1) : Merge t1 ('(n2, o2):t2))
     ('(n2, o2) : Merge ('(n1, o1):t1) t2)
GHC's take on type-level computation

• Differences
  • Type functions are arbitrary computation and need not be terminating (cf. Merge)
  • Backwards compatible with HM type inference (no search & no higher-order unification)

• What's next for GHC?
  • Anonymous type-level functions,
  • More flexibility in higher-order polymorphism,
  • Uniform syntax for type and term functions
Indices constrain values

We can use compile-time computation to define type structure and guide the type checker.
How does this work?

> :t d

Dict [('b', 'Once'), ('d', 'Many'), ('e', 'Opt')]

> get @$e$ d

Just "hs"

> get @$f$ d

<interactive>:28:1: error:
  • I couldn't find a group named 'f' in {b, d, e}

Overloaded access, resolved by type-level symbol

Custom error message
Types constrain data

```haskell
data Dict :: SM -> Type where
  Nil :: Dict '[]
  (==> :: Entry '(n,o) -> Dict tl
    -> Dict '(n,o) : tl)

data Entry :: (Symbol,Occ) -> Type where
  E :: \n  o. OccType o -> Entry '(n,o)

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

GADTs [Peyton Jones, Vytiniotis, Washburn, Weirich ICFP 2006]
Types Constrain Data

```haskell
dict ::
Dict '[("b", 'Once), ("d", 'Many), ("e", 'Opt)]
```

- The dict must be of the form
  
  ```haskell
  E someString
  => E someListOfStrings
  => E someMaybeString => Nil
  ```

- Type checker knows group for "b" comes first, and that the stored value is a string
- Type checker knows that a value for "f" is not present in the dict
GHC's take on indexed types

• Overloaded access to dictionary

\[
\text{get} :: \forall n \ r \ a. \ \text{Has} \ n \ r \ a \Rightarrow r \rightarrow a
\]

• Compile-time constraint solving guided by a type-level "Find" function, which calculates offset into the dictionary

\[
\text{instance} \ (\text{Get} \ (\text{Find} \ n \ s :: \text{Index} \ n \ o \ s),
\text{a} \sim \text{OccType} \ o) \Rightarrow \text{Has} \ n \ (\text{Dict} \ s) \ a \ \text{where}
\text{get} = \ldots
\]

• If Find function fails, custom type error is triggered

Custom Type Errors [Augusttson, HS 2015]
ClosedTypeFamilies [Eisenberg, Peyton Jones, Weirich POPL 2014]
TypeInType [Weirich, Hsu, Eisenberg, ICFP 2013]
Double-duty data

We can use the same data for compile time and runtime computation.
How does this work?

data Dict :: SM -> Type where
   Nil :: Dict '[]
   (,:) :: Entry '(n,o) -> Dict tl
       -> Dict ('(n,o):tl)

data Entry :: (Symbol,Occ) -> Type where
   E :: ∀n o. OccType o -> Entry '(n,o)

d :: Dict ['("b", Once),('"d", Many),('"e", Opt)]

d = E "Regexp" :> E ["dth", "popl17"]
     :> E (Just "hs") :> Nil

> show d
{b="Regexp",d=["dth","popl17"],e=Just ".hs"}
Dependent types: \( \Pi \)

\[
\text{showEntry} :: \Pi n \to \Pi o \to \text{Entry } '(n,o) \to \text{String}
\]

\[
\text{showEntry } n \ o \ (E \ x) = 
\begin{align*}
\text{show } n & + + "=\" + + \text{showData } o \ x \ \text{where} \\
\text{showData} :: \Pi o \to \text{OccType } o \to \text{String}
\end{align*}
\]

\[
\text{showData } \text{Once } x = \text{show } x \quad \text{-- for String} \\
\text{showData } \text{Opt } x = \text{show } x \quad \text{-- for Maybe String} \\
\text{showData } \text{Many } x = \text{show } x \quad \text{-- for [String]}
\]

\[
\text{show} :: \text{Show } a \to a \to \text{String}
\]

\[
\text{instance Show Symbol where show } = \ldots
\]
GHC's take: Singletons

showEntry :: Sing n -> Sing o -> Entry '(n,o) -> String

showEntry n o (E x) =
  show n ++ "=" ++ showData o x where
  showData :: Sing o -> OccType o -> String
  showData SOnce x = show x  -- for String
  showData SOpt x = show x   -- for Maybe String
  showData SMany x = show x  -- for [String]

instance Show (Sing (n :: Symbol)) where show = ...

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

Boilerplate automated by Singletons library
[Eisenberg and Weirich, HS 2012]
Singletons are "easyish"

• Uniform type for all singletons, indexed by kinds

```haskell
type Sing (a :: k) ...
```

• Type class supplies singletons via type inference

```haskell
class SingI (a :: k) where
    sing :: Sing a

instance (SingI n, SingI o) => Show (Entry (n,o))
    where show = showEntry sing sing

instance (SingI s) => Show (Dict s)
    where show = showDict sing
```

• What's next? Richard Eisenberg close to adding a true \( \Pi \) type to GHC
Equivalence matters

Type checking depends on an expressive definition of program equality
Regular Expression datatype (no indices)

data R where

  Rempty :: R   -- \( \varepsilon \) (accepts empty string)
  Rchar  :: Char \rightarrow R  -- accepts single char
  Ralt   :: R \rightarrow R \rightarrow R  -- alternative \( r_1 | r_2 \)
  Rseq   :: R \rightarrow R \rightarrow R  -- sequence \( r_1 r_2 \)
  Rstar  :: R \rightarrow R  -- iteration \( r^* \)
  Rvoid  :: R  -- \( \varnothing \) (always fails)
  Rmark  :: String \rightarrow String \rightarrow R \rightarrow R

rseq :: R \rightarrow R \rightarrow R
rseq Rvoid r2  =  Rvoid
rseq r1 Rvoid  =  Rvoid
rseq Rempty r2 =  r2
rseq r1 Rempty =  r1
rseq r1 r2     =  Rseq r1 r2

"Smart constructors" optimize regexp creation
Regeps with type indices

data R s where
  Rempty :: R '[]
  Rchar :: Char -> R '[]
  Ralt :: R s1 -> R s2 -> R (Alt s1 s2)
  Rseq :: R s1 -> R s2 -> R (Merge s1 s2)
  Rstar :: R s -> R (Repeat s)
  Rvoid :: R s
  Rmark :: Sing n -> String
    -> R s -> R (Merge '(n,Once) s)

rseq :: R s1 -> R s2 -> R (Merge s1 s2)
rseq Rvoid r2 = Rvoid  -- need Rvoid :: R (Merge s1 s2)
rseq r1 Rvoid = Rvoid
rseq Rempty r2 = r2    -- Merge '[] s2 ~ s2 (by def)
rseq r1 Rempty = r1
rseq r1 r2 = Rseq r1 r2
type family Repeat (s :: SM) :: SM where
    Repeat '[] = '[]
    Repeat ('(n,o) : t) = '(n, Many) : Repeat t

rstar :: R s -> R (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!
code

```haskell
class (Repeat (Repeat s) ~ Repeat s) => Wf (s :: SM)

instance Wf '[] -- base case

instance (Wf s) => Wf ('(n,o) : s) -- inductive step

data R s where
  Ralt :: (Wf s1, Wf s2) => R s1 -> R s2 -> R (Merge s1 s2)

  Rstar :: (Wf s) => R s -> R (Repeat s)

  ...

rstar :: Wf s => R s -> R (Repeat s)

rstar Rempty = Rempty -- have: Repeat '[] ~ '[]

rstar (Rstar r) = Rstar r

  -- have: Repeat (Repeat s1) ~ Repeat s1

rstar r = Rstar r
```

Make sure property is available everywhere

---

Equality constraints to the rescue
Submatching using Brzozowski Derivatives

match r w = extract (foldl' deriv r w)

Based on "Martin Sulzmann, Kenny Zhuo Ming Lu. Regular expression sub-matching using partial derivatives."
GHC's take on proofs

- **Compile-time proofs**
  - Type-level function based proof (i.e. `Find`) work best when the argument is concretely known at compile time
  - `Wf` works for properties about a single variable, with simple inductive proof

- **Runtime proofs**
  - Express properties using GADTs, and prove them via functions, but with a runtime cost
  - Creating these proofs is tedious without tactics or IDE support!

- **What's next? More automated theorem proving!**
  - Vilhelm Sjöberg's dissertation [2015] integrates congruence closure algorithm with full-spectrum dependent types
  - Type-checker plugins allow solvers to help [Diatchki, HS 2015]
  - Connection with LiquidHaskell?
Four Features of Dependently Typed Programs

1. Types compute
2. Indices constrain values
3. Double-duty data
4. Equivalence matters
Conclusion: GHC is in a novel & fascinating part of the design space of dependently typed languages.

And more to come!
fin
Awesome Collaborators
Extract the parts of a filepath "dth/popl17/Regexp.hs"

```plaintext
/?((?P<d>[/\-]+)?(?P<b>[/\.-]+)?(?P<e>/.+?)?)?
```

> match path "dth/popl17/Regexp.hs"

Just {b="Regexp", d=["dth","popl17"], e=Just ".hs"}

> d = fromJust it

> get @"b" d

"Regexp"

> get @"a" d

<interactive>:28:1: error:
   • I couldn't find a group named 'a' in 
     {b, d, e}
Demo

Type-level computation of named capture groups
Examples

ghci> r1 = rmark @"a" rany
ghci> :t r1
r1 :: R '["a", 'Once]]
ghci> r2 = rmark @"b" rany
ghci> :t r2
r2 :: R '["b", 'Once]]
ghci> :t r1 `rseq` r1
r1 `rseq` r1 :: R '["a", 'Many]]
ghci> :t r1 `rseq` r2
r1 `rseq` r2 :: R '["a", 'Once), "b", 'Once]]
ghci> :t r1 `ralt` r1
r1 `ralt` r1 :: R '["a", 'Once]]
ghci> :t r1 `ralt` r2
r1 `ralt` r2 :: R '["a", 'Opt), "b", 'Opt]]
ghci> :t rstar r1
rstar r1 :: R '["a", 'Many]]
TemplateHaskell to promote type functions

\[
\begin{align*}
\text{singletons } [d] \\
\text{empty} :: \text{U} \\
\text{empty} = [] \\
\text{one} :: \text{Symbol} \to \text{U} \\
\text{one } s = [(s, \text{Once})] \\
\text{merge} :: \text{U} \to \text{U} \to \text{U} \\
\text{merge } m \ [] = m \\
\text{merge } [] \ m = m \\
\text{merge } (e1@(n1,o1)\ : \ t1) \ (e2@(n2,o2)\ : \ t2) = \\
\quad \text{if } n1 \ == \ n2 \ \text{then } (n1, \text{Many}) : \text{merge } t1 \ t2 \\
\quad \text{else if } n1 \ <= \ n2 \ \text{then } e1 : \text{merge } t1 \ (e2 : t2) \\
\quad \text{else } e2 : \text{merge } (e1 : t1) \ t2 \\
\end{align*}
\]

[Eisenberg and Stolarek, HS 2014]
Regexp Derivatives

ghci> r = [re|....|]  --matches any 4 chars
ghci> deriv r 'P'
...
ghci> deriv it 'O'
...
ghci> deriv it 'P'
.
ghci> deriv it 'L'
ε
ghci> extract it
Just {}
Regexp derivative matching

ghci> r = [re|(?P<b>..)(?P<a>..)|]
ghci> deriv r 'P'
(?P<b:"P">.)(?P<a>..)
ghci> deriv it 'O'
(?P<b:"PO">ε)(?P<a>..)
ghci> deriv it 'P'
(?P<b:"PO">ε)(?P<a:"P">.)
ghci> deriv it 'L'
(?P<b:"PO">ε)(?P<a:"PL">ε)
ghci> extract it
Just {a="PL",b="PO"}
Regular Expression Derivatives w/ matching

match :: R -> String -> Bool
match r w = extract (foldl' deriv r w)

deriv :: R -> Char -> R
deriv (Rchar s) c | c == s = rempty
deriv (Rseq r1 r2) c =
  ralt (rseq (deriv r1 c) r2)
  (rseq (markEmpty r1) (deriv r2 c))
deriv (Rseq r1 r2) c = rseq (deriv r1 c) r2
deriv (Ralt r1 r2) c = ralt (deriv r1 c) (deriv r2 c)
deriv (Rstar r) c = rseq (deriv r c) (rstar r)
deriv (Rmark n w r) c = Rmark n (w ++ [c]) (deriv r c)
deriv _ c = Rvoid

Smart constructors optimize new regexp on the fly, only keeping marked strings
Derivatives with types, almost

defiv :: R s -> Char -> R s
defiv (Rchar s) c | c == s = rempty
defiv (Rseq r1 r2) c =
    ralt (rseq (deriv r1 c) r2) -- needs: s ~ Alt s s
    (rseq (markEmpty r1) (deriv r2 c))
defiv (Rseq r1 r2) c = rseq (deriv r1 c) r2
defiv (Ralt r1 r2) c = ralt (deriv r1 c) (deriv r2 c)
defiv (Rstar r) c = rseq (deriv r c) (rstar r)
    -- needs: Merge s (Repeat s) ~ Repeat s
defiv (Rmark n w r) c = Rmark n (w ++ [c]) (deriv r c)
defiv _ c = Rvoid
Equality constraints to the rescue (again)

class (Repeat (Repeat s) ~ Repeat s, s ~ Alt s s,
       Merge s (Repeat s) ~ Repeat s) => Wf (s :: U)

instance Wf '[] -- base case for all properties
instance (WfOcc o, Wf s) => Wf ('(n,o) : s)

class (o ~ Max o o) => WfOcc (o :: Occ)
instance WfOcc Once
instance WfOcc Opt
instance WfOcc Many
Derivatives with types

\[
\text{deriv} :: \text{Wf } s \Rightarrow \text{R } s \rightarrow \text{Char} \rightarrow \text{R } s
\]
\[
\text{deriv } (\text{Rchar } s) \quad \text{c} \mid \text{c} == \text{s} = \text{reempty}
\]
\[
\text{deriv } (\text{Rseq } r1 \ r2) \quad \text{c} =
\]
\[
\quad \text{ralt } (\text{rseq } (\text{deriv } r1 \ c) \ r2) \quad \text{-- have: } s \sim \text{Alt } s \ s
\]
\[
\quad (\text{rseq } (\text{markEmpty } r1) \ (\text{deriv } r2 \ c))
\]
\[
\text{deriv } (\text{Rseq } r1 \ r2) \quad \text{c} = \text{rseq } (\text{deriv } r1 \ c) \ r2
\]
\[
\text{deriv } (\text{Ralt } r1 \ r2) \quad \text{c} = \text{ralt } (\text{deriv } r1 \ c) \ (\text{deriv } r2 \ c)
\]
\[
\text{deriv } (\text{Rstar } r) \quad \text{c} = \text{rseq } (\text{deriv } r \ c) \ (\text{rstar } r)
\]
\[
\text{-- have: } \text{Merge } s \ (\text{Repeat } s) \sim \text{Repeat } s
\]
\[
\text{deriv } (\text{Rmark } n \ w \ r) \quad \text{c} = \text{Rmark } n \ (w ++ [c]) \ (\text{deriv } r \ c)
\]
\[
\text{deriv } _ \quad \text{c} = \text{Rvoid}
\]
Why Dependent Types?

• **Verification**: Dependent types express application-specific program invariants that are beyond the scope of existing type systems.

• **Expressiveness**: Dependent types enable flexible interfaces, of particular importance to embedded DSLs, generic programming and metaprogramming.

• **Uniformity**: The same syntax and semantics is used for computations, specifications and proofs.

Everything is “just programming”

Ultimate goal: making the type checker more informative

Dependent types can seem mysterious...

... but types dispel mysteries
A type system for static typing of a domain-specific language
Paul E. McKenney, Nathan A. Lindop, Wim A. Vanderbauwhede
February 2008 FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays
Publisher: ACM
Bibliometrics: Citation Count: 0
With the increase in system complexity, designers are increasingly using IP blocks as a means for filling the designer productivity gap. This has given rise to system level languages which connect IP blocks together. However, these languages have in general not been subject to formalisation. They are considered too trivial ...
Keywords: type system, FPGA, static type checking
A type system for format strings
Konstantin Weiht, Gene Kim, Siwakorn Srisakakul, Michael D. Ernst
Publisher: ACM
Bibliometrics: Citation Count: 2
Full text available: PDF
Most programming languages support format strings, but their use is error-prone. Using the wrong format string syntax, or passing the wrong number or type of arguments, leads to unintelligible text output, program crashes, or security vulnerabilities. This paper presents a type system that guarantees that calls to format string APIs ...
Keywords: printf, static analysis, Format string, type system
A type system for regular expressions
Eric Spisshak, Werner Dietl, Michael D. Ernst
ACM International Conference on Design and Analysis of Computer Systems 2017 Workshop on Formal Techniques for Java-like Programs
Publisher: ACM
Bibliometrics: Citation Count: 3