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"

Demo

- A regular expression for file paths

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

Types Compute

We can use dependent types to implement a domain-specific compile-time analysis

Type aware implementation

```
> 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` 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]

Constructors have informative types

```
-- accepts empty string only
rempty :: R '[]
-- accepts single char only
rchar  :: Char -> R '[]

-- alternative r₁|r₂
ralt   :: R s1 -> R s2 -> R (Alt s1 s2)
-- sequence  r₁r₂
rseq   :: R s1 -> R s2 -> R (Merge s1 s2)
-- iteration r*
rstar  :: R s -> R (Repeat s)

-- marked subexpression
rmark  :: ∀n s. R s -> R (Merge '(n,Once) s)
```

TypeFamilies [Schrijvers, Peyton Jones, Chakravarty, Sulzmann, ICFP 2008]

Computing with types

```haskell
data Occ = Once | Opt | Many
type SM = [(Symbol,Occ)]
```

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

```haskell
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

```
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

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

- The dict must be of the form

```
    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

```
get :: ∀n r a. Has n r a => r -> a
```

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

```
instance (Get (Find n s :: Index n o s),
       a ~ OccType o) => Has n (Dict s) a where
    get = …
```

- If Find function fails, custom type error is triggered

Custom Type Errors [Augusstson, 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: Π

```
showEntry :: Π n -> Π o -> Entry '(n,o) -> String
showEntry n o (E x) =
  show n ++ "=" ++ showData o x where
    showData :: Π o -> OccType o -> String
    showData  Once x = show x  -- for String
    showData  Opt  x = show x  -- for Maybe String
    showData  Many x = show x  -- for [String]

show :: Show a => a -> String
instance Show Symbol where show = …
```

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

```
type Sing (a :: k) …
```

- Type class supplies singletons via type inference

```
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 Π 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                    -- ε (accepts empty string)
  Rchar  :: Char -> R            -- accepts single char
  Ralt   :: R -> R -> R          -- alternative r₁|r₂
  Rseq   :: R -> R -> R          -- sequence  r₁r₂
  Rstar  :: R -> R               -- iteration  r*
  Rvoid  :: R                    -- φ (always fails)
  Rmark  :: String -> String -> R -> R


rseq :: R -> R -> 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

Regexps with type indices

```haskell
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
```

Regexps with types indices

```
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!

Equality constraints to the rescue

```
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

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!

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

fin

Awesome Collaborators

Regular Expression Submatching Demo

Extract the parts of a filepath `"dth/popl17/Regexp.hs"`

```
/?((?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

```
$(singletons [d|
   empty :: U
   empty = []
   one :: Symbol -> U
   one s = [(s,Once)]
   merge :: U -> U -> U
   merge m  [] = m
   merge [] m  = m
   merge (e1@(n1,o1):t1) (e2@(n2,o2):t2) =
     if n1 == n2 then (n1, Many) : merge t1 t2
     else if n1 <= n2 then e1 : merge t1 (e2:t2)
     else e2 : merge (e1:t1) t2
  |])
```

[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

```
deriv :: R s -> Char -> R s
deriv (Rchar s)      c | c == s = rempty
deriv (Rseq r1 r2)   c =
  ralt (rseq (deriv r1 c) r2) -- needs: s ~ Alt s s
       (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)
  -- needs: Merge s (Repeat s) ~ Repeat s
deriv (Rmark n w r) c = Rmark n (w ++ [c]) (deriv r c)
deriv _              c = Rvoid
```

Equality constraints to the rescue (again)

```haskell
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

```
deriv :: Wf s => R s -> Char -> R s
deriv (Rchar s)      c | c == s = rempty
deriv (Rseq r1 r2)   c =
  ralt (rseq (deriv r1 c) r2) -- have: s ~ Alt s s
       (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)
  -- have: Merge s (Repeat s) ~ Repeat s
deriv (Rmark n w r) c = Rmark n (w ++ [c]) (deriv r c)
deriv _              c = Rvoid
```

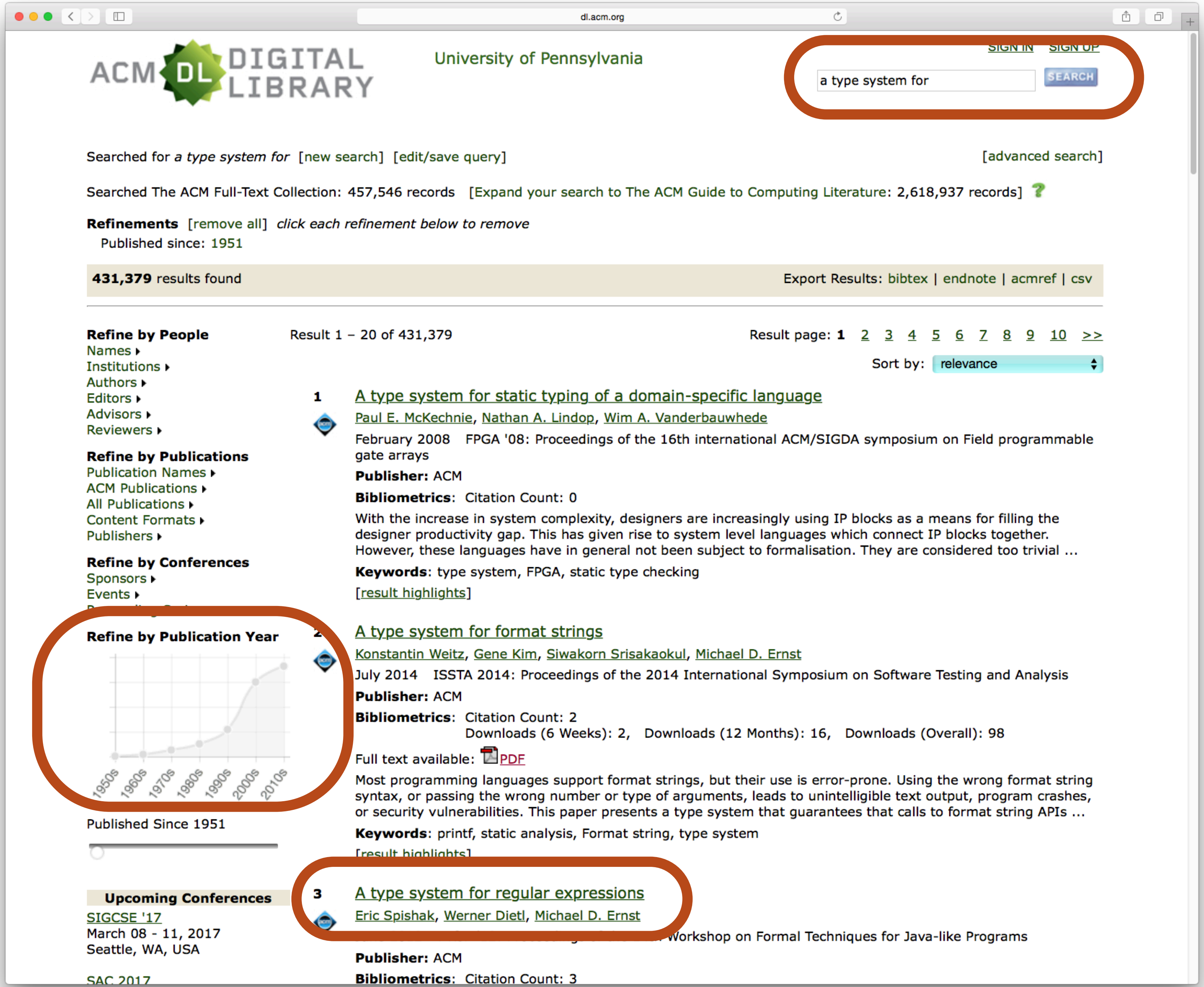
Why Dependent Types?

- *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



University of Pennsylvania

SIGN IN SIGN UP

a type system for SEARCH

Searched for *a type system for* [new search] [edit/save query] [advanced search]

Searched The ACM Full-Text Collection: 457,546 records [Expand your search to The ACM Guide to Computing Literature: 2,618,937 records]

Refinements [remove all] *click each refinement below to remove*
Published since: 1951

431,379 results found Export Results: bibtex | endnote | acmref | csv

Refine by People
Names
Institutions
Authors
Editors
Advisors
Reviewers

Refine by Publications
Publication Names
ACM Publications
All Publications
Content Formats
Publishers

Refine by Conferences
Sponsors
Events

Refine by Publication Year

Published Since 1951

Upcoming Conferences
SIGCSE '17
March 08 - 11, 2017
Seattle, WA, USA

SAC 2017

Result 1 – 20 of 431,379 Result page: **1** 2 3 4 5 6 7 8 9 10 >>

Sort by: relevance

1 A type system for static typing of a domain-specific language
Paul E. McKechnie, 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
[result highlights]

2 A type system for format strings
Konstantin Weitz, Gene Kim, Siwakorn Srisakaokul, Michael D. Ernst
July 2014 ISSTA 2014: Proceedings of the 2014 International Symposium on Software Testing and Analysis
Publisher: ACM
Bibliometrics: Citation Count: 2
Downloads (6 Weeks): 2, Downloads (12 Months): 16, Downloads (Overall): 98
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
[result highlights]

3 A type system for regular expressions
Eric Spishak, Werner Dietl, Michael D. Ernst
Workshop on Formal Techniques for Java-like Programs
Publisher: ACM
Bibliometrics: Citation Count: 3