Dependent Types in Haskell: Theory and Practice

Richard A. Eisenberg
University of Pennsylvania
eir@cis.upenn.edu

Dissertation Proposal Defense
Thursday, June 4, 2015

Off to emacs...

Other applications

- Typed reflection / Cloud Haskell [1]
- Embedded domain-specific languages
- Inferred algebraic effects [2]
- Type-safe database access [3]
- Machine-checked algorithms [4, 5]

[1]: https://ghc.haskell.org/trac/ghc/wiki/DistributedHaskell
[2]: E. Brady. “Programming and reasoning with algebraic effects and dependent types”. ICFP ’13
[3]: N. Oury and W. Swierstra. “The Power of Pi”. ICFP ’08
[4]: T. Altenkirch, C. McBride, and J. McKinna. “Why Dependent Types Matter”. http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf
[5]: A. Dergunov. “Generalized Algebraic Data Types in Haskell”. In The Monad.Reader #22, 2013.

Why Haskell?

Type inference:

```
length :: [a] -> Int
length []       = 0
length (_ : xs) = 1 + length xs
```

Proving soundness and completeness

almost [1]

[1]: D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. “OutsideIn(X): Modular Type Inference with Local Assumptions”. JFP #21, 2011.

Why Haskell?

Embracing partiality:

```
data StepResult :: Expr ‘[] ty -> * where
  Stepped :: pi (e’ :: Expr ‘[] ty)
          -> (‘eval e ~ ‘eval e’)
          => StepResult e
Value   :: pi (v :: Val ty)
          -> (‘eval e ~ v)
          => StepResult e
```

partial (or, hard to prove total)

And: `* :: *`

Why Haskell?

Two dependent quantifiers:

`forall` and `pi`

erased

not erased

Why Haskell?

Haskell is “real world”

- Industrial-strength optimizing compiler
- Some adoption in industry for Getting Work Done
- Broad ecosystem & user base

Dependent Haskell

It's All About the Quantifiers

- A quantifier introduces a function argument type.
- Today's Haskell has 3: `forall`, `->`, and `=>`
- Questions about quantifiers:
  - Is the quantifiee *dependent*?
  - Is the quantifiee *relevant*?
  - Is the quantifiee *visible*?

Dependency

- A quantifiee is *dependent* if it can be used later in a type.
- `forall` is *dependent*. `->` and `=>` are *non-dependent*.

```
foo :: forall a. a -> a
foo = id          -- OK to use a in a type

bar :: Bool -> Proxy b
bar b = Proxy     -- bad to use b in a type
```

Relevance

- A quantifiee is *relevant* if it can be used in a relevant context or matched against.
- Almost, but not quite, the opposite of erasable.
- `forall` is *irrelevant*. `->` and `=>` are *relevant*.

```
foo :: forall (b :: Bool). Proxy b -> Bool

foo _ = not b    -- bad, that's relevant

foo _ = foo (Proxy :: Proxy (Not b))
                 -- OK, that's irrelevant
```

Visibility

- A quantifiee is *visible* if its value must be supplied by the programmer.
- `->` is *visible*. `forall` and `=>` are *invisible*.

```
foo :: forall a. Show a => a -> String
foo x = show x
        -- no pattern for a or Show a
```

Quantifiers, Today

Quantifier	Dep?	Relevant?	Visible?
`forall.`	Yes	No	unification
`->`	No	Yes	Yes
`=>`	No	Yes	solving

Quantifiers, Tomorrow

Quantifier	Dep?	Relevant?	Visible?
`forall.`	Yes	No	unification
`forall->`	Yes	No	Yes
`pi.`	Yes	Yes	unification
`pi->`	Yes	Yes	Yes
`->`	No	Yes	Yes
`=>`	No	Yes	solving

Π

Pi-bound quantifiees are *relevant* and *dependent*.

```
data Vec :: * -> Nat -> * where
  Nil  :: Vec a ‘Zero
  (:>) :: a -> Vec a n -> Vec a (‘Succ n)

replicate ::
  forall a. pi (n :: Nat) -> a -> Vec a n
replicate Zero      _ = Nil
replicate (Succ n’) x
  = x :> replicate n’ x
```

Merging Types and Kinds

* :: *

- *All* types are now kinds, too
- Promoting GADTs (previously unpromotable)
- Requires kind equalities [1]

[1]: S. Weirich, J. Hsu, and R. A. Eisenberg. “System FC with Explicit Kind Equality”. ICFP ’13

Current Status

- Merged type/kind language complete
- Type inference for merged language complete, though not yet analyzed
- Implementation (without Π) working, nearly complete

Remaining Challenges

- Type inference details, including proofs
  - Confident about inference with Π due to experimentation with singletons [1]
- Implementation in GHC
- Expanding subset shared between types and terms
  - Possibly can promote lots of functions [2]
- Integration with roles [3]

[1]: R. A. Eisenberg and S. Weirich. “Dependently typed programming with singletons”. Haskell Symp. ’12.
[2]: R. A. Eisenberg and J. Stolarek. “Promoting functions to type families in Haskell”. Haskell Symp. ’14.
[3]: R. A. Eisenberg. “An overabundance of equality: Implementing kind equalities into Haskell”. Submitted to Haskell Symp. ’15

Beyond Scope

- Promoting class constraints / dictionaries
- Termination checking [1]
- Pattern coverage checking [2]
- Laziness
- Higher-order unification

[1]: N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones. “Refinement Types for Haskell”. ICFP ’14
[2]: G. Karachalias, T. Schrijvers, D. Vytiniotis, and S. Peyton Jones. “GADTs meet their match”. ICFP ’15

Timeline

2015

July Submit POPL paper

Aug

Sep Merge w/ GHC master

Oct Develop new FC

Nov

Dec

2016

Jan Finish Π

Feb Submit ICFP paper

Mar

Apr Prove type inf.

May Finish writing

June Defend thesis

Dependent Types in Haskell: Theory and Practice

Richard A. Eisenberg
University of Pennsylvania
eir@cis.upenn.edu

Dissertation Proposal Defense
Thursday, June 4, 2015