Dependent Types in Haskell

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

Friday, 24 October, 2014
New York, NY, USA

Dependent Types of Today

github.com/goldfirere/nyc-hug-oct2014

off to emacs….

>> break <<

Dependent Types of Tomorrow

This is preliminary.

I want your input.

off to emacs again....

Spoiler Alert!

- All your old Haskell programs will still work.*
- Including non-terminating ones.
- Type inference will, hopefully, remain predictable.

* `let` really should not be generalized. Even over kinds.

Outline, in brief

I. Surface language design of Dependent Haskell

II. Current status

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 *relevant*?
  - Is the quantifiee *dependent*?
  - Is the quantifiee *visible*?
  - Is the quantification *required*?

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
```

Dependence

- 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
```

Visibility

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

```
foo :: forall b. SingI b => Sing (Not b)
foo = sNot (sing :: Sing b)
         -- no argument patterns

bar :: Sing b -> Sing (Not b)
bar sb = sNot sb
         -- sb appears in the code
```

Requirement

- A quantification is *required* if it must be given explicitly by the programmer.
- `->` and `=>` are required. `forall` is optional.

```
foo :: a -> a    -- “forall a.” is omitted
foo = id
```

Quantifiers, Today

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

Quantifiers, Tomorrow

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

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
```

Type = Kind

IT WORKS!

All types can be used as kinds

type synonyms

type families

GADTs

Type = Kind

IT WORKS!

Kind variables can be listed explicitly in declarations.

```
data T k a (b :: k) = MkT (a b)
  T :: pi (k :: *) -> (k -> *) -> k -> *
```

Core Language

IT WORKS!

See

Weirich, Hsu, Eisenberg
System FC With Explicit Kind Equality
ICFP '13

Open Questions

- Promoted type class dictionaries?
- Unsaturated type families? (But see Eisenberg & Stolarek; HS 2014)
- Optional termination checking? (But see Vazos, Seidel, & Jhala; ICFP 2014)
- Optional pattern-match totality checking?
- Other sources of partiality? (Non-strictly-positive datatypes, other recursive datatypes, etc.)
- Promoting infinite terms?

Status Report

Core Language

- Merged type/kind language: Done.
- Eliminated sub-kinding: Done.
- Pi-types: Designed core datatype; still propagating changes.

Type Inference

- Merged type/kind language: Done.
- Accepting explicit kind variables: Done.
- Designed type inference algorithm, based on Gundry's, but to work with OUTSIDEIN: Done?
- Proof of correctness of inference algorithm: Under way.
- Goal: type inference will be sound and infer only principal types. Completeness is not tractable.
- Caveat: No plans for higher-order unification.

Next Steps

- Merge the (type = kind) work into master, including type inference algorithm.
- Finish implementing Π in Core.
- Implement (and prove) type inference for a surface language with Π.
- Parse new language.
- Release.
- Graduate.

Dependent Types in Haskell

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

Friday, 24 October, 2014
New York, NY, USA