Dependent Haskell

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

Saturday, 6 September, 2014
HIW, Göteborg, Sweden

Demo

Disclaimer

This is preliminary.

I want your input.

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

Quantifiers, Today

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

Quantifiers, Today

Can the quantifiee appear later in the type?

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

Quantifiers Today

Must a caller write an argument in this slot?

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

Quantifiers Today

Must this quantifier appear in a type sig?

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

Quantifiers Today

Can the quantifiee appear later in the term?

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

Quantifiers, Tomorrow

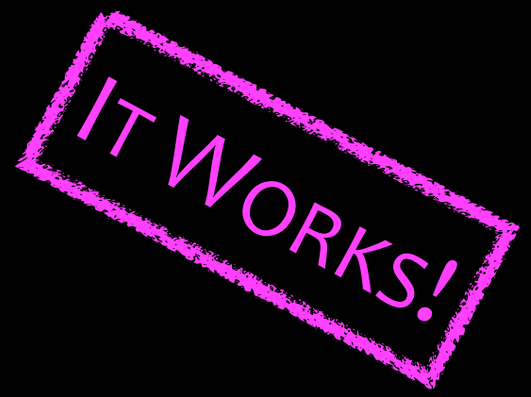
Quantifier	Dep?	Visible?	Required?	Relevant?
`forall.`	Yes	unification	FVs	No
`forall->`	Yes	Yes	Yes	No
`pi.`	Yes	unification	Yes	Yes
`pi->`	Yes	Yes	Yes	Yes
`->`	No	Yes	Yes	Yes
`=>`	No	solving	Yes	Yes

Π

Pi-bound identifiers live in both terms and types:

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

Type = Kind



All types can be used as kinds

type synonyms

type families

GADTs

Type = Kind


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

Core Language

See

Weirich, Hsu, Eisenberg

System FC With Explicit Kind Equality

ICFP '13

Parsing

Below is my best guess. Advice welcome.

- Combine type, kind, and term parsers.
- ‘ injects term in a type; ^ injects type in a term.
- If a name is missing from the default namespace, try the other one.

Parsing *

```
Foo * Int
```

Is it `Foo` applied to the kind `*` and `Int`?

Is it `(*)` applied to `Foo` and `Int`?

Proposal:

- Deprecate `*` in all code
- Disallow `*` with `-XDependentTypes`
- Export `U` (and `Constraint`) from `Data.Kind`
- Perhaps start this transition now

Concrete Syntax Questions

- Can we even merge the type and term parsers?
- How to supply a visible argument when an invisible one is expected?
  Proposal: Prefix with @
- How to avoid supplying a visible argument when one is expected?
  Proposal: Use _. How does this work with holes?
- Is `forall (...) ->` just plain silly?
- What do we think of `U`?

Other 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.

```
data Type = ... | PiTy Binder Ty | ...
data Binder = Binder
   { binder_payload    :: BinderVar
   , binder_dependence :: DependenceFlag
   , binder_visibility :: VisibilityFlag
   , binder_relevance  :: RelevanceFlag   }
data BinderVar = Named Var | Anon Type
```

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 guess-free-complete, like current algorithm.
- 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.

Dependent Haskell

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

Saturday, 6 September, 2014
HIW, Göteborg, Sweden

Π

Arguments to be Pi-bound must be expressible in both terms and types.

Good:
```
replicate (Succ Zero) ‘x’
```

Bad:
```
let n = case Just Zero of
          Nothing -> Zero
          Just m  -> Succ m
in
replicate n ‘x’
```

Parsing: Probable Problems

- `forall` must become a proper keyword, making it not a possible variable name.
- ‘ means “term” in a type, but it means “Template Haskell quote” in a term.
- ! is a strictness annotation in types and patterns, but an operator in terms.
- Non-problems: `->` `=>` `\`