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
<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Dep?</th>
<th>Visible?</th>
<th>Required?</th>
<th>Relevant?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>forall</code> (...) .</td>
<td>Yes</td>
<td>unification</td>
<td>free vars</td>
<td>No</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>=&gt;</code></td>
<td>No</td>
<td>solving</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Quantifiers, Today

Can the quantifiee appear later in the type?

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Dep?</th>
<th>Visible?</th>
<th>Required?</th>
<th>Relevant?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{forall} (...) . )</td>
<td>Yes</td>
<td>unification</td>
<td>free vars</td>
<td>No</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>( \Rightarrow )</td>
<td>No</td>
<td>solving</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Quantifier</td>
<td>Dep?</td>
<td>Visible?</td>
<td>Required?</td>
<td>Relevant?</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>-------------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td><code>forall (...) .</code></td>
<td>Yes</td>
<td>unification</td>
<td>free vars</td>
<td>No</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>=&gt;</code></td>
<td>No</td>
<td>solving</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Quantifier</td>
<td>Dep?</td>
<td>Visible?</td>
<td>Required?</td>
<td>Relevant?</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
<td>--------------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td><code>forall (...) .</code></td>
<td>Yes</td>
<td>unification</td>
<td>free vars</td>
<td>No</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>=&gt;</code></td>
<td>No</td>
<td>solving</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Quantifier</td>
<td>Dep?</td>
<td>Visible?</td>
<td>Required?</td>
<td>Relevant?</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>------------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td><code>forall (...) .</code></td>
<td>Yes</td>
<td>unification</td>
<td>free vars</td>
<td>No</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>=&gt;</code></td>
<td>No</td>
<td>solving</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Quantifier</td>
<td>Dep?</td>
<td>Visible?</td>
<td>Required?</td>
<td>Relevant?</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>--------------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>forall.</td>
<td>Yes</td>
<td>unification</td>
<td>FVs</td>
<td>No</td>
</tr>
<tr>
<td>forall-›</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>pi.</td>
<td>Yes</td>
<td>unification</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>pi-›</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>-›</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>=&gt;</td>
<td>No</td>
<td>solving</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Pi-bound identifiers live in both terms and types:

\[
\text{replicate} :: \\
\quad \text{forall } a. \pi (n :: \text{Nat}) \to a \to \text{Vec } a \text{ } n \\
\text{replicate } \text{Zero } \_ = \text{Nil} \\
\text{replicate } (\text{Succ } n') x \\
\quad = x :: : \text{replicate } n' x
\]
Type = Kind

All types can be used as kinds

- type synonyms
- type families
- GADTs
Type = Kind

\[
\text{data } T \ k \ a \ (b :: k) = \text{MkT} \ (a \ b) \\
\text{-- } T :: \text{pi} \ (k :: U) \to (k \to U) \to k \to U
\]
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.

```haskell
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?
- 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
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: `->` => `\`