Partial Type Constructors

Or, Making ad hoc datatypes less ad hoc

Mark P. Jones
Portland State University

J. Garrett Morris
University of Kansas

Richard A. Eisenberg
Tweag I/O
rae@richarde.dev
@RaeHaskell

Friday, 11 September 2020
MuniHac

```
data List a
  = Nil
  | Cons a (List a)
```

```
List Int            ✅
List Bool           ✅
List (Int -> Int)   ✅
List Person         ✅
```

```
data Set a = ...
```

Sets make sense only for elements with an equality relation.

`Set Int` ✅

`Set Bool` ✅

`Set (Int -> Int)` ❌

`Set Person` ✅

```
data BSTSet a = ...
```

Binary search trees make sense only for elements with a total order.

`BSTSet Int` ✅

`BSTSet Bool` ✅

`BSTSet (Int -> Int)` ❌

`BSTSet Person` ❌

No person is greater than another.

Sets make sense only for elements with an equality relation.

Binary search trees make sense only for elements with a total order.

```
size :: Set (Int -> Int)
size = ...
```

✅

"But nothing can go wrong here!" 🐛

```
idMaybe :: Maybe -> Maybe
idMaybe x = x
```

❌

"But nothing can go wrong here!"

Why reject `idMaybe` but accept `size`?

Because we've assumed all type constructors are total.

There are many partial types:

- `Set a`
- `BST a`
- `UArray a`
- `StateT s m a`
- `Complex n`
- `SharedArray a`
- `Encrypted bits a`
- `...`

Problem is more than static checks

```
instance Functor Set where
  fmap = ...
```

❌

Because `Set`'s functions are constrained, we can't write this instance.

There are workarounds.

But our idea is better.

Related work is in the paper.

Key idea:
Datatype contexts

```
data Ord a => BST a
```

BST a is a type only
when Ord a holds.

Key idea:
Datatype contexts

```
> ghc BST.hs
BST.hs: error:
    Illegal datatype context (use
DatatypeContexts): Ord a =>
```

Key idea: Datatype contexts

```
> ghc BST.hs
BST.hs: warning:
    -XDatatypeContexts is deprecated: It
was widely considered a misfeature, and has
been removed from the Haskell language.
```

GHC's `DataCon` module:

```
  dcStupidTheta :: ThetaType
  -- The context of the data type declaration
  --      data Eq a => T a = ...
  -- "Stupid", because the dictionaries
  -- aren't used for anything.
```

Key idea:
Datatype contexts

But these weren't always stupid…

Key idea:
Datatype contexts

Haskell 1.0 Report [Hudak and Wadler 1990]:

$$\texttt{data c => T u}_1\ \texttt{...}\ \texttt{u}_n$$

"declares that a type $\texttt{T t}_1\ \texttt{...}\ \texttt{t}_n$ is only valid where $\texttt{c[t}_1\texttt{/u}_1\texttt{, ..., t}_n\texttt{/u}_n\texttt{]}$ holds."

This text is missing from the Haskell 1.1 Report [Hudak et al. 1991].

Key idea:
Datatype contexts

Our goal:
Bring back 1990!

(by giving datatype contexts a sensible semantics)

Today's datatype contexts are indeed stupid.

```
data Ord a => BST a = Mk ...

f :: BST Person -> BST Person
f x = x
```

✅

🐛

Today's datatype contexts are indeed stupid.

```
data Ord a => BST a = Mk ...

seqBST :: BST a -> ()
seqBST (Mk {}) = ()
```

❌

```
No instance for (Ord a)
arising from a use of ‘Mk’
```

Key idea:
Datatype contexts

Our interpretation:
An occurrence of `BST a` requires an `Ord a` constraint.

`idBST :: BST a -> BST a` ❌

`idBST :: Ord a => BST a -> BST a` ✅

```
idBST :: BST a -> BST a                  ❌
idBST :: Ord a => BST a -> BST a         ✅
```

But the `Ord a` constraint is redundant and annoying, so we elaborate the former to the latter.

```
f :: BST a -> a -> a -> Bool
f _ x y = x < y                          ✅
```

Ord a is implied.

```
idBST :: BST a -> BST a
idBST :: Ord a => BST a -> BST a
```

✅ ✅

But the `Ord a` constraint is redundant and annoying, so we elaborate the former to the latter.

```
f :: BST a -> a -> a -> Bool
f _ x y = x < y
```

✅

Ord a is implied.

What about abstraction?

For `f a` to be a type,
we must know any constraints are satisfied.

`a` must be in the domain of `f`.

`f @ a` must hold.

For t_1 t_2 to be a type,
t_1 `@` t_2 must hold.

For concrete types `T`,
`T @ a` is `T`'s datatype
context, if any.

`BST @ a` $\Longleftrightarrow$ `Ord a`

$$\frac{\begin{array}{c} P \mid \Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \\ P \mid \Delta \vdash \tau_2 : \kappa_1 \\ P \mid \Delta \models \tau_1 @ \tau_2 \end{array}}{P \mid \Delta \vdash \tau_1\, \tau_2 : \kappa_2}$$

Example

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

elaborates to

```
class Functor f where
  fmap :: (f @ a, f @ b)
       => (a -> b) -> f a -> f b
```

```
instance Functor BST where ...
```
✅

Theory

We can compile our surface language into an internal language without partiality. (but with dependent types)

Internal Language

$$\begin{array}{lrcl}
\text{Kinds} & \kappa & ::= & s \mid (\alpha{:}\kappa_1) \rightarrow \kappa_2 \mid (\delta{:}\pi) \Rightarrow \kappa \\
\text{Types} & \tau, \pi & ::= & C \mid \alpha \mid \tau_1\,\tau_2 \mid \tau\,\upsilon \\
 & & \mid & \forall\alpha{:}\kappa.\tau \mid (\delta{:}\pi) \Rightarrow \tau \\
\text{Evidence} & \upsilon & ::= & \delta \mid \Diamond \mid \ldots \\
\text{Expressions} & E & ::= & x \mid \lambda x{:}\tau.E \mid E_1\,E_2 \mid \lambda\delta{:}\pi.E \\
 & & \mid & E\,\upsilon \mid \Lambda\alpha{:}\kappa.E \mid E\,\tau
\end{array}$$

$$\begin{array}{lrcl}
\text{Type constants} & C, L & ::= & (\rightarrow) \mid \top_\kappa \mid \ldots \\
\text{Type vars} & \alpha, \ell & ::= & \ldots \\
\text{Evidence vars} & \delta & ::= & \ldots \\
\text{Term vars} & x & ::= & \ldots \\
\text{Sorts} & s & ::= & \star \mid \mathsf{o} \\
\text{Kinding env's} & \Delta & ::= & \epsilon \mid \Delta, \alpha{:}\kappa \mid \Delta, \delta{:}\pi \\
\text{Typing env's} & \Gamma & ::= & \epsilon \mid \Gamma, x{:}\tau
\end{array}$$

$$\frac{\Delta \vdash_i \tau_1 : (\alpha{:}\kappa_1) \rightarrow \kappa_2 \qquad \Delta \vdash_i \tau_2 : \kappa_1}{\Delta \vdash_i \tau_1\,\tau_2 : [\tau_2/\alpha]\kappa_2} \qquad \frac{\Delta \vdash_i \tau : (\delta{:}\pi) \Rightarrow \kappa \qquad \Delta \vdash_i \upsilon : \pi}{\Delta \vdash_i \tau\,\upsilon : [\upsilon/\delta]\kappa}$$

$$\frac{\Delta \vdash_i \kappa_1\ \mathsf{kind} \qquad \Delta, \alpha{:}\kappa_1 \vdash_i \kappa_2\ \mathsf{kind}}{\Delta \vdash_i (\alpha{:}\kappa_1) \rightarrow \kappa_2\ \mathsf{kind}} \qquad \frac{\Delta \vdash_i \pi : \mathsf{o} \qquad \Delta, \delta{:}\pi \vdash_i \kappa\ \mathsf{kind}}{\Delta \vdash_i (\delta{:}\pi) \Rightarrow \kappa\ \mathsf{kind}}$$

Compilation

Key idea:
`f a` compiles into `f a d`,
where (`d : f @ a`).

To quantify (`f : * -> *`), we must
quantify over (`c : * -> o`),
where (`(@) f`) = `c`.

Compilation Example

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

elaborates to

```
fmap :: forall (f :: * -> *) (a :: *) (b :: *).
        Functor f => f @ a => f @ b =>
        (a -> b) -> f a -> f b
```

compiles to

```
fmap :  ∀ (c : * -> o) (f : (a:*) -> c a => *)
          (a : *) (b : *).
        Functor c f => (d1 : c a) => (d2 : c b) =>
        (a -> b) -> f a d1 -> f b d2
```

Compilation

$$\boxed{\Delta \rightsquigarrow \Delta'; \mu}$$

$$\frac{}{\epsilon \rightsquigarrow \epsilon; \epsilon} \qquad \frac{\Delta \rightsquigarrow \Delta'; \mu \quad \kappa; \epsilon \rightsquigarrow \kappa'; \psi}{\Delta, \alpha{:}\kappa \rightsquigarrow \Delta', \psi, \alpha{:}\kappa'; \mu, \alpha \mapsto \psi}$$

$$\boxed{\Delta \mid P \rightsquigarrow \Delta'; \mu}$$

$$\frac{\Delta \rightsquigarrow \Delta'; \mu}{\Delta \mid \epsilon \rightsquigarrow \Delta'; \mu} \qquad \frac{\Delta \mid P \rightsquigarrow \Delta'; \mu \quad \Delta \mid P \vdash \pi \text{ pred} \rightsquigarrow_\mu \pi'}{\Delta \mid P, \pi \rightsquigarrow \Delta', \delta{:}\pi'; \mu, \pi \mapsto \delta}$$

$$\boxed{P \Vdash \pi \rightsquigarrow_\mu \upsilon}$$

$$\frac{\pi \mapsto \delta \in \mu}{P \Vdash \pi \rightsquigarrow_\mu \delta} \qquad \frac{\text{solve}(\pi) \rightsquigarrow \upsilon}{P \Vdash \pi \rightsquigarrow_\mu \upsilon}$$

$$\boxed{P \mid \Delta \vdash \pi \text{ pred} \rightsquigarrow_\mu \pi'}$$

$$\frac{P \mid \Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \rightsquigarrow_\mu \tau_1'; \pi, \overline{\tau} \quad P \mid \Delta \vdash \tau_2 : \kappa_1 \rightsquigarrow_\mu \tau_2'; \overline{\tau'}}{P \mid \Delta \vdash \tau_1 \mathbin{@} \tau_2 \text{ pred} \rightsquigarrow_\mu \pi\, \overline{\tau'}\, \tau_2'}$$

$$\frac{L : \overline{\kappa_i} \to \text{pred} \quad P \mid \Delta \vdash \tau_i : \kappa_i \rightsquigarrow_\mu \tau_i'; \overline{\tau_i''}}{P \mid \Delta \vdash L\, \overline{\tau_i} \text{ pred} \rightsquigarrow_\mu L\, \overline{\tau''}\, \overline{\tau'}}$$

$$\boxed{P \mid \Delta \vdash \sigma : \kappa \rightsquigarrow_\mu \tau; \overline{\tau'}}$$

$$\frac{\alpha{:}\kappa \in \Delta \quad \alpha \mapsto \psi \in \mu}{P \mid \Delta \vdash \alpha : \kappa \rightsquigarrow_\mu \alpha; \mathit{dom}(\psi)} \qquad \frac{C : \kappa}{P \mid \Delta \vdash C : \kappa \rightsquigarrow_\mu C; \text{lookup}(C)}$$

$$\frac{\kappa; \epsilon \rightsquigarrow \kappa'; \psi \quad P \mid \Delta, \alpha{:}\kappa \vdash \sigma : \star \rightsquigarrow_{\mu, \alpha \mapsto \psi} \tau; \overline{\tau}}{P \mid \Delta \vdash \forall \alpha{:}\kappa.\sigma : \star \rightsquigarrow_\mu \forall \psi. \forall \alpha{:}\kappa'.\tau; \epsilon}$$

$$\frac{\begin{array}{c} P \mid \Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \rightsquigarrow_\mu \tau_1'; \overline{\tau} \quad P \mid \Delta \vdash \tau_2 : \kappa_1 \rightsquigarrow_\mu \tau_2'; \overline{\tau'} \\ P \Vdash \tau_1 \mathbin{@} \tau_2 \rightsquigarrow_\mu \upsilon \quad \overline{\tau''} = [\tau_0\, \overline{\tau'}\, \tau_2' \mid \tau_0 \leftarrow \text{tail}(\overline{\tau})] \end{array}}{P \mid \Delta \vdash \tau_1\, \tau_2 : \kappa_2 \rightsquigarrow_\mu \tau_1'\, \overline{\tau'}\, \tau_2'\, \upsilon; \overline{\tau''}}$$

$$\frac{P \mid \Delta \vdash \pi \text{ pred} \rightsquigarrow_\mu \pi' \quad P, \pi \mid \Delta \vdash \rho : \star \rightsquigarrow_{\mu, \pi \mapsto \delta} \tau; \overline{\tau}}{P \mid \Delta \vdash \pi \Rightarrow \rho : \star \rightsquigarrow_\mu (\delta : \pi') \Rightarrow \tau; \epsilon}$$

$$\boxed{\kappa; \psi \rightsquigarrow \kappa;' \psi'}$$

$$\frac{}{\star; \psi \rightsquigarrow \star; \epsilon} \qquad \frac{\begin{array}{c} \kappa_1; \epsilon \rightsquigarrow \kappa_1'; \psi_1 \quad \psi' = \psi, \psi_1, \alpha{:}\kappa_1' \\ \kappa_2; \psi' \rightsquigarrow \kappa_2'; \psi_2 \quad \psi_2' = \ell{:}(\forall \psi. \forall \psi_1. \kappa_1' \to \mathrm{o}), \psi_2 \end{array}}{\kappa_1 \to \kappa_2; \psi \rightsquigarrow \forall \psi_1.(\alpha{:}\kappa_1') \to \ell\ \mathit{dom}(\psi)\ \mathit{dom}(\psi_1)\ \alpha \Rightarrow \kappa_2'; \psi_2'}$$

$$\boxed{P \mid \Delta; \Gamma \vdash E : \sigma \rightsquigarrow_\mu E'}$$

$$\frac{x{:}\sigma \in \Gamma}{P \mid \Delta; \Gamma \vdash x : \sigma \rightsquigarrow_\mu x}$$

$$\frac{P \mid \Delta; \Gamma \vdash E_1 : \sigma \rightsquigarrow_\mu E_1' \quad P \mid \Delta \vdash \sigma : \star \rightsquigarrow_\mu \tau'; \overline{\tau''} \quad P \mid \Delta; \Gamma, x{:}\sigma \vdash E_2 : \tau \rightsquigarrow_\mu E_2'}{P \mid \Delta; \Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau \rightsquigarrow_\mu (\lambda x{:}\tau'.E_2')\, E_1'}$$

$$\frac{P \mid \Delta; \Gamma \vdash E_1 : \tau_1 \to \tau_2 \rightsquigarrow_\mu E_1' \quad P \mid \Delta; \Gamma \vdash E_2 : \tau_1 \rightsquigarrow_\mu E_2'}{P \mid \Delta; \Gamma \vdash E_1\, E_2 : \tau_2 \rightsquigarrow_\mu E_1'\, E_2'}$$

$$\frac{P \mid \Delta; \Gamma, x{:}\tau_1 \vdash E : \tau_2 \rightsquigarrow_\mu E' \quad P \mid \Delta \vdash \tau_1 \to \tau_2 : \star \rightsquigarrow_\mu \tau_1' \to \tau_2'; \overline{\tau''}}{P \mid \Delta; \Gamma \vdash \lambda x.E : \tau_1 \to \tau_2 \rightsquigarrow_\mu \lambda x{:}\tau_1'.E'}$$

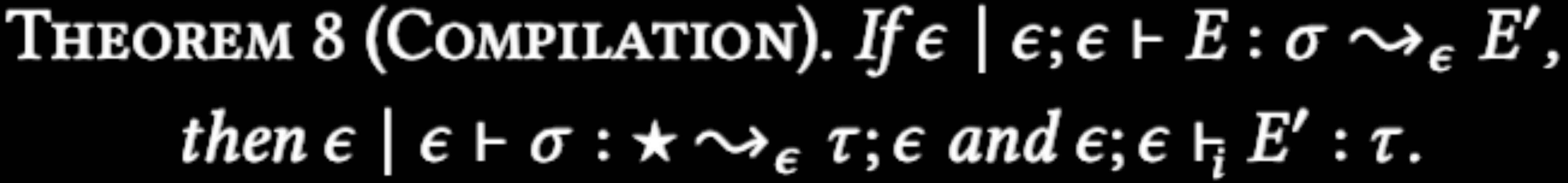
$$\frac{P \mid \Delta; \Gamma \vdash E : \pi \Rightarrow \rho \rightsquigarrow_\mu E' \quad P \Vdash \pi \rightsquigarrow_\mu \upsilon}{P \mid \Delta; \Gamma \vdash E : \rho \rightsquigarrow_\mu E'\, \upsilon}$$

$$\frac{P \mid \Delta \vdash \pi \text{ pred} \rightsquigarrow_\mu \pi' \quad P, \pi \mid \Delta; \Gamma \vdash E : \rho \rightsquigarrow_{\mu, \pi \mapsto \delta} E'}{P \mid \Delta; \Gamma \vdash E : \pi \Rightarrow \rho \rightsquigarrow_\mu \lambda \delta : \pi'.E'}$$

$$\frac{P \mid \Delta; \Gamma \vdash E : \forall \alpha{:}\kappa.\sigma \rightsquigarrow_\mu E' \quad P \mid \Delta \vdash \tau : \kappa \rightsquigarrow_\mu \tau''; \overline{\tau}}{P \mid \Delta; \Gamma \vdash E : [\tau/\alpha]\sigma \rightsquigarrow_\mu E'\, \overline{\tau}\, \tau'}$$

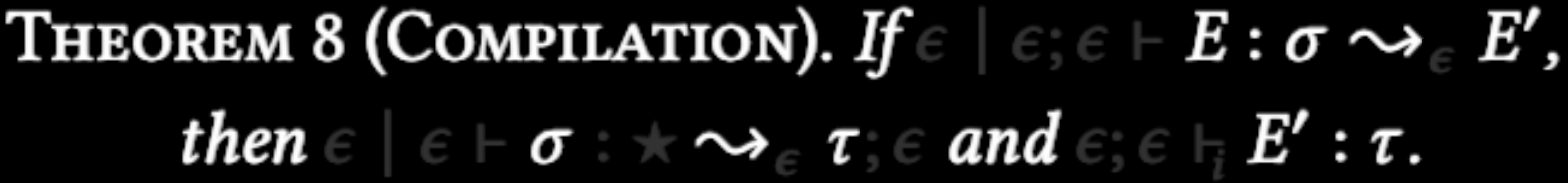
$$\frac{\kappa; \epsilon \rightsquigarrow \kappa'; \psi \quad P \mid \Delta, \alpha{:}\kappa; \Gamma \vdash E : \sigma \rightsquigarrow_{\mu, \alpha \mapsto \psi} E'}{P \mid \Delta; \Gamma \vdash E : \forall \alpha{:}\kappa.\sigma \rightsquigarrow_\mu \Lambda \psi. \Lambda \alpha{:}\kappa'.E'}$$

Compilation



Theorem 8 (Compilation). *If* $\epsilon \mid \epsilon; \epsilon \vdash E : \sigma \rightsquigarrow_{\epsilon} E'$, *then* $\epsilon \mid \epsilon \vdash \sigma : \star \rightsquigarrow_{\epsilon} \tau; \epsilon$ *and* $\epsilon; \epsilon \vdash_{i} E' : \tau$.

Compilation



Theorem 8 (Compilation). *If* $\epsilon \mid \epsilon;\epsilon \vdash E : \sigma \rightsquigarrow_{\epsilon} E'$, *then* $\epsilon \mid \epsilon \vdash \sigma : \star \rightsquigarrow_{\epsilon} \tau;\epsilon$ *and* $\epsilon;\epsilon \vdash_{i} E' : \tau$.

✅

Implementation

- in Hugs
- of "research quality"

Used to test:

- 169 source files
- 38,000 loc

Annotation burden

```
mapAndUnzipM :: Monad m => (a -> m (b, c)) ->
                [a] -> m ([b], [c])
mapAndUnzipM f xs = sequence (map f xs) >>=
                    return . unzip

                                  :: m [(b, c)]
```

We need a (`m @ [(b, c)]`) constraint.

An alternate implementation wouldn't.

Annotation burden

Out of 1,934 type signatures, 20 needed extra annotations.

These were easy.

Modularity

Types constrain implementations.

(A bit like how `Set` operations need an `Ord` or `Hashable` constraint.)

Is this a problem?
Time will tell.

Related Work

- Java/Scala's bounded polymorphism
- ML modules
- Scott's E-logic
- GADTs are an orthogonal feature
- Other approaches to partiality
- Constrained type families

Partial Type Constructors

Or, Making ad hoc datatypes less ad hoc

Mark P. Jones
Portland State University

J. Garrett Morris
University of Kansas

Richard A. Eisenberg
Tweag I/O
rae@richarde.dev
@RaeHaskell

Friday, 11 September 2020
MuniHac