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.

```haskell
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]:

\[
\text{data } c \Rightarrow T \ u_1 \ldots \ u_n
\]

"declares that a type \( T \ t_1 \ldots \ t_n \) is only valid where \( c[t_1/u_1, \ldots, t_n/u_n] \) 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.

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

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

✅
Today's datatype contexts are indeed stupid.

```haskell
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.

\[
\begin{align*}
idBST &:: \text{BST} \ a \rightarrow \text{BST} \ a \quad \text{❌} \\
idBST &:: \text{Ord} \ a \Rightarrow \text{BST} \ a \rightarrow \text{BST} \ a \quad \text{✅}
\end{align*}
\]
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.

\[
f :: BST a \rightarrow a \rightarrow a \rightarrow \text{Bool}
\]

\[
f \_ x y = x < y
\]

`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 $\iff$ Ord a
\[ P \mid \Delta \vdash \tau_1 : \kappa_1 \rightarrow \kappa_2 \]
\[ P \mid \Delta \vdash \tau_2 : \kappa_1 \]
\[ P \mid \Delta \vdash \tau_1 \odot \tau_2 \]
\[ \hline \]
\[ P \mid \Delta \vdash \tau_1 \tau_2 : \kappa_2 \]
Example

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

elaborates to

```haskell
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

Kinds  \( \kappa ::= s \mid (\alpha: \kappa_1) \rightarrow \kappa_2 \mid (\delta: \pi) \Rightarrow \kappa \)

Types  \( \tau, \pi ::= C \mid \alpha \mid \tau_1 \tau_2 \mid \tau \pi \)
  \( \mid \forall \alpha: \kappa. \tau \mid (\delta: \pi) \Rightarrow \tau \)

Evidence  \( \nu ::= \delta \mid \diamond \mid \ldots \)

Expressions  \( E ::= x \mid \lambda x: \tau. E \mid E_1 E_2 \mid \lambda \delta: \pi. E \)
  \( \mid E \nu \mid \Lambda \alpha: \kappa. E \mid E \tau \)

Type constants  \( C, L ::= (\rightarrow) \mid T_\kappa \mid \ldots \)

Type vars  \( \alpha, \ell ::= \ldots \)

Evidence vars  \( \delta ::= \ldots \)

Term vars  \( x ::= \ldots \)

Sorts  \( s ::= \star \mid o \)

Kinding env’s  \( \Delta ::= \epsilon \mid \Delta, \alpha: \kappa \mid \Delta, \delta: \pi \)

Typing env’s  \( \Gamma ::= \epsilon \mid \Gamma, x: \tau \)

\[ \frac{\Delta \vdash \tau_1: (\alpha: \kappa_1) \rightarrow \kappa_2 \quad \Delta \vdash \tau_2: \kappa_1}{\Delta \vdash \tau_1 \tau_2: [\tau_2/\alpha] \kappa_2} \]

\[ \frac{\Delta \vdash \tau: (\delta: \pi) \Rightarrow \kappa \quad \Delta \vdash \nu: \pi}{\Delta \vdash \tau \nu: [\nu/\delta] \kappa} \]

\[ \frac{\Delta \vdash \kappa_1 \text{ kind} \quad \Delta, \alpha: \kappa_1 \vdash \kappa_2 \text{ kind}}{\Delta \vdash (\alpha: \kappa_1) \rightarrow \kappa_2 \text{ kind}} \]

\[ \frac{\Delta \vdash \pi: o \quad \Delta, \delta: \pi \vdash \kappa \text{ kind}}{\Delta \vdash (\delta: \pi) \Rightarrow \kappa \text{ kind}} \]
Compilation

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

To quantify \( (f : * \rightarrow *) \), we must quantify over \( (c : * \rightarrow o) \),
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
```
Theorem 8 (Compilation). If $\epsilon | \epsilon; \epsilon \vdash E : \sigma \leadsto_\epsilon E'$, then $\epsilon | \epsilon \vdash \sigma : * \leadsto_\epsilon \tau ; \epsilon$ and $\epsilon ; \epsilon \vdash_i E' : \tau$. 
Theorem 8 (Compilation). If $\epsilon | \epsilon; \epsilon \vdash E : \sigma \sim_{\epsilon} E'$, then $\epsilon | \epsilon \vdash \sigma : \star \sim_{\epsilon} \tau; \epsilon$ and $\epsilon; \epsilon \vdash_i E' : \tau$. 

Compilation
Implementation

• in Hugs
• of "research quality"

Used to test:
• 169 source files
• 38,000 loc
mapAndUnzipM :: Monad m => (a -> m (b, c)) -> [a] -> m ([b], [c])
mapAndUnzipM f xs = sequence (map f xs) >>=
  return . unzip

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