Partial Type Constructors in Practice

Apoorv Ingle  Alex Hubers  J. Garrett Morris

The University of Iowa, USA
Kind checking rules out nonsensical types

\[(\text{TAPP}) \quad \frac{\Delta \vdash \tau : \kappa \rightarrow \kappa' \quad \Delta \vdash \sigma : \kappa}{\Delta \vdash \tau \sigma : \kappa'}\]
Kind checking rules out nonsensical types

[Int] is well defined

Int [] is nonsensical
Kind checking rules out nonsensical types

[Int] is well kinded

\[ \Delta \vdash [\ ] : \ast \rightarrow \ast \quad \Delta \vdash \text{Int} : \ast \]
\[ \Delta \vdash \text{[Int]} : \ast \]

Int [ ] is ill kinded

\[ \Delta \vdash \text{Int} : \ast \quad \Delta \vdash [ ] : \ast \rightarrow \ast \]
\[ \Delta \vdash \text{Int [ ]} : ??? \]
Does kind checking rule out nonsensical types?

[Int] is well kinded and well defined

Int [] is ill kinded and nonsensical
Defining Partial Types: Motivation

Does kind checking rule out all nonsensical types?
Does kind checking rule out *all* nonsensical types? No :(

\[
\begin{align*}
\Delta \vdash \text{Set} : \star \rightarrow \star & \quad \Delta \vdash \text{Int} \rightarrow \text{Int} : \star \\
\Delta \vdash \text{Set} (\text{Int} \rightarrow \text{Int}) : \star
\end{align*}
\]

Elements of \text{Set} need to be ordered

\text{Int} \rightarrow \text{Int} is not ordered in Haskell
There are more partial types

```haskell
data Ratio a = ... -- a better satisfy Integral a
data UArray i e = ... -- i better satisfy Ix i and e be Unboxed
data StateT s m a = ... -- m better satisfy Monad m
```
Motivation

Problem:
Current Haskell assumes all types are total
Motivation

Problem:
Current Haskell assumes all types are total

Consequences:

1. Library writers need to explicitly write extra constraints

   singleton :: Ord a ⇒ a → Set a
Problem:
Current Haskell assumes all types are total
Consequences:
1. Library writers need to explicitly write extra constraints
   \[ \text{singleton} :: \text{Ord}\ a \Rightarrow a \rightarrow \text{Set}\ a \]
2. Partial datatypes cannot leverage typeclass abstractions
   Constrained Functor Problem

```haskell
instance\Functor\ Set\ where
    fmap \::\ (a \rightarrow\ b) \rightarrow\ \text{Set}\ a \rightarrow\ \text{Set}\ b
-- \text{mapSet} \::\ (\text{Ord}\ a,\ \text{Ord}\ b) \Rightarrow (a \rightarrow\ b) \rightarrow\ \text{Set}\ a \rightarrow\ \text{Set}\ b
    \text{fmap} = \text{mapSet} -- \text{Typechecking fails!}
```
Motivation

How can we make partiality in types explicit?

What impact will this have on existing code?
How can we make partiality in types explicit?
How can we make partiality in types explicit?

Define a predicate on types: $\tau \circlearrowright \sigma$

\[
\tau \circlearrowright \sigma \text{ holds } \iff \tau \sigma \text{ is well-defined}
\]
Defining Partial Types

How can we make partiality in types explicit?

Define a predicate on types: \( \tau @ \sigma \)

\[
\begin{align*}
\tau @ \sigma \text{ holds} & \iff \tau \sigma \text{ is well-defined} \\
\text{Set } @ a \text{ holds} & \iff \text{Ord } a \text{ holds} \\
\text{Ratio } @ a \text{ holds} & \iff \text{Integral } a \text{ holds} \\
\text{UArray } @ i \text{ holds} & \iff \text{Ix } i \text{ holds} \\
\text{UArray } i @ e \text{ holds} & \iff \text{Unboxed } e \text{ holds} \\
[] @ a \text{ holds} & \iff \top \text{ holds}
\end{align*}
\]
New kinding rule rules out all nonsensical types

\[
\frac{
\Delta \vdash \tau : \kappa \rightarrow \kappa' \quad \Delta \vdash \sigma : \kappa
}{
\Delta \vdash \tau \sigma : \kappa'
\}
\]
mapSet :: forall a b. (Ord a, Ord b) ⇒ (a → b) → Set a → Set b
mapSet :: forall a b. (Ord a, Ord b) ⇒ (a → b) → Set a → Set b

With explicit partiality, Set @ a ⇔ Ord a

mapSet :: forall a b. (Set @ a, Set @ b) ⇒ (a → b) → Set a → Set b
What about classes?

```
class Functor f where
    fmap :: (f @ a, f @ b) ⇒ (a → b) → f a → f b
```
What have we managed to do?
What have we managed to do?

\[
\text{fmap} :: (f @ a, f @ b) \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b
\]

\[
\text{mapSet} :: (\text{Set} @ a, \text{Set} @ b) \Rightarrow (a \rightarrow b) \rightarrow \text{Set} a \rightarrow \text{Set} b
\]

[Drum roll]
What have we managed to do?

\[
\text{fmap} :: (f @ a, f @ b) \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b
\]

\[
\text{mapSet} :: (\text{Set} @ a, \text{Set} @ b) \Rightarrow (a \rightarrow b) \rightarrow \text{Set} a \rightarrow \text{Set} b
\]

[Drum roll]

```haskell
instance Functor Set where
  fmap = mapSet -- Typechecks!
```
Partial Types in Action

Also a Monad instance for Set

```haskell
instance Monad Set where -- Typechecks
  return :: (Set @ a) ⇒ a → Set a
  return = ...  
  (>>=) :: (Set @ a, Set @ b)  
    ⇒ Set a → (a → Set b) → Set b  
  (>>=) = ...  
```
Define a predicate on types: \( \tau @ \sigma \)

\( \tau @ \sigma \) holds \( \iff \) \( \tau \sigma \) is well defined

But how do we implement this in GHC?
Define a predicate on types: \( \tau @ \sigma \)

\( \tau @ \sigma \) holds \iff \( \tau \sigma \) is well defined

Take 1: Use a Typeclass

```haskell
class (@) (t :: k \rightarrow k') (u :: k)
```
Define a predicate on types: $\tau @ \sigma$

$\tau @ \sigma$ holds $\iff$ $\tau \sigma$ is well defined

Take 1: Use a Typeclass

```haskell
class (@) (t :: k \rightarrow k') (u :: k)

instance [] @ σ
```
Define a predicate on types: \( \tau \mathrel{@} \sigma \)
\( \tau \mathrel{@} \sigma \) holds \iff \( \tau \sigma \) is well defined

Take 1: Use a Typeclass

\[
\text{class } (@) (t :: k \to k') (u :: k)
\]

\[
\text{instance } [] @ \sigma
\]

\[
\text{instance } \text{Ord } \sigma \Rightarrow \text{Set } @ \sigma
\]
Define a predicate on types: \( \tau \@ \sigma \)
\( \tau \@ \sigma \) holds \( \iff \) \( \tau \sigma \) is well defined

Take 1: Use a Typeclass

class \((\@)\) (t :: k → k') (u :: k)

instance [] @ σ

instance Ord σ ⇒ Set @ σ

\[
\text{Ord } σ \vdash \text{Set } @ σ
\]

but

\[
\text{Set } @ σ \not\vdash \text{Ord } σ
\]

Typeclasses do not allow bidirectional reasoning
Define a predicate on types: \( \tau @ \sigma \)

\( \tau @ \sigma \) holds \iff \( \tau \sigma \) is well defined

Take 2: Use a type family

\texttt{type family } (@) (t :: k' \to k) (u :: k') :: Constraint
Define a predicate on types: $\tau \mathrel{@} \sigma$

$\tau \mathrel{@} \sigma$ holds $\iff$ $\tau \sigma$ is well defined

Take 2: Use a type family

```
type family (@) (t :: k' → k) (u :: k') :: Constraint
```

```
type instance [] @ σ = ()
```
Defining Partial Types

Define a predicate on types: \( \tau @ \sigma \)

\( \tau @ \sigma \) holds \( \iff \) \( \tau \sigma \) is well defined

Take 2: Use a type family

\[
\text{type family } (@) (t :: k' \to k) (u :: k') :: \text{Constraint}
\]

\[
\text{type instance } [] @ \sigma = ()
\]

\[
\text{type instance } \text{Set} @ \sigma = \text{Ord} \sigma
\]

\[
\text{Set} @ \sigma \vdash \text{Ord} \sigma
\]

also

\[
\text{Ord} \sigma \vdash \text{Set} @ \sigma
\]

Exactly what we need ✓
That’s all great but..

- Where do all these $\odot$ constraints come from?

- Are there any programs that are no longer typeable?
Where do these @ constraints come from?
Where do these @ constraints come from?

Elaboration
Type signatures

$(\gg\gg\gg) :: \forall a \ b. \ m\ a \to (a \to m\ b) \to m\ b$

elaborates to

$(\gg\gg\gg) :: \forall a \ b. (m @ a, m @ b) \Rightarrow m\ a \to (a \to m\ b) \to m\ b$
Datatypes

???

elaborates to

data Set a = ...

type instance Set @ a = Ord a
Breaking News: Thetas now considered not stupid

{-# LANGUAGE DatatypeContext #-} to rescue

```haskell
data Ord a ⇒ Set a = ...
```
Datatypes

```haskell
data Ord a ⇒ Set a = ...
```

elaborates to

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

```haskell
type instance Set @ a = Ord a
```
Partial Types in Action

Are there any programs that are no longer typeable?
Are there any programs that are no longer typeable? Yes

```
data Ap f a = MkAp (f a)
-- Ap @ f ∼ ()    Ap f @ a ∼ ()
-- MkAp :: forall f a. f @ a ⇒ f a → Ap f a

instance Functor f ⇒ Functor (Ap f) where

    fmap :: (Ap f @ a, Ap f @ b) ⇒ (a → b) → Ap f a → Ap f b
    fmap g (MkAp k) = MkAp (fmap g k) -- typechecking fails!
```

Cannot prove \( f @ b \) due to the use of MkAp
Partial Types in Action

Need more type annotations

1. Make the data type be well defined only when the type arguments are well defined

\[ \text{data } f \ @ a \Rightarrow \text{Ap } f \ a = \text{MkAp } (f \ a) \]

\[
\begin{align*}
\text{-- Ap } @ f \sim () & \Rightarrow \text{Ap } f \ @ a \sim f \ @ a \\
\text{-- MkAp :: forall } f \ a. f \ @ a \Rightarrow f \ a \rightarrow \text{Ap } f \ a
\end{align*}
\]

\[ \text{instance Functor } f \Rightarrow \text{Functor } (\text{Ap } f) \text{ where} \]

\[ \text{fmap } g \ (\text{MkAp } k) = \text{MkAp } (\text{fmap } g \ k) \rightarrow \text{Okay} \]

\[
\begin{align*}
\text{-- fmap :: (Ap } f @ a, \text{Ap } f @ b) \Rightarrow (a \rightarrow b) \rightarrow \text{Ap } f \ a \rightarrow \text{Ap } f \ b \\
\text{-- fmap :: (f } @ a, f @ b) \Rightarrow (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b
\end{align*}
\]
Need more type annotations

2. Assert that the type is well defined on all types in the instance declaration

```haskell
data Ap f a = MkAp (f a)
-- Ap @ f ~ () Ap f @ a ~ ()
-- MkAp :: forall f a. f @ a ⇒ f a → Ap f a
```
Need more type annotations

2. Assert that the type is well defined on all types in the instance declaration

\[
\text{data } \text{Ap } f \ a = \text{MkAp } (f \ a)
\]

\[\text{-- } \text{Ap } @ f \sim () \quad \text{Ap } f @ a \sim ()\]

\[\text{-- MkAp } :: \text{forall } f \ a. \ f @ a \Rightarrow f a \rightarrow \text{Ap } f a\]

\[
\text{instance } (\text{forall } a. \ f @ a, \text{Functor } f) \Rightarrow \text{Functor } (\text{Ap } f) \text{ where }
\]

\[
\text{fmap } g \ (\text{MkAp } k) = \text{MkAp } (\text{fmap } g \ k) \quad \text{-- Okay}
\]
Need more annotations

2. Assert that the type is well defined on all types in the instance declaration

```haskell
type Total f = forall a. f @ a

instance (Total f, Functor f) ⇒ Functor (Ap f) where
    fmap g (MkAp k) = MkAp (fmap g k) -- Okay
```
\textbf{data} \quad \text{Ap} \ f \ a = \text{MkAp} \ (f \ a) \\
\textbf{data} \ f \ @ \ a \Rightarrow \text{Ap} \ f \ a = \text{MkAp} \ (f \ a) \\
Semantic difference

Should not automate too much
Are there any programs that are no longer typeable? Yes, sometimes

Two ways to fix the problem

1. Make the data type be well defined only when the type arguments are well defined
2. Assert that the type is well defined for all types in the instance declaration
How often is this *sometimes*?
How often is this sometimes?

Case study: Compile GHC and libraries (base, mtl, etc.)

Benchmark changes in types

No term changes
How often is this sometimes?

Case study: Compile GHC and libraries (base, mtl, etc.)

Benchmark changes in types

No term changes

< 10% overall
How often is this \textbf{sometimes}?

Case study: Compile GHC and libraries (base, mtl, etc.)

Benchmark changes in types

No term changes

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
 & Classes and Insts, Modified/Total & Term Sigs, Modified/Total \\
\hline
compiler/GHC & 133/1931 (6.9\%) & 218/16129 (1.3\%) \\
libraries & 495/5442 (9.7\%) & 412/17337 (2.8\%) \\
\hline
\end{tabular}
\end{table}
Who are the biggest culprits in libraries?

<table>
<thead>
<tr>
<th>Category</th>
<th>Classes and Insts, Modified/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraries</td>
<td>495/5442 (9.7%)</td>
</tr>
<tr>
<td>libraries/transformers</td>
<td>167/444 (37.6%)</td>
</tr>
<tr>
<td>libraries/base</td>
<td>78/1108 (7.0%)</td>
</tr>
<tr>
<td>libraries/mtl</td>
<td>69/80 (86.2%)</td>
</tr>
</tbody>
</table>

Top 3 account for > 60%

But why?
The Applicative typeclass
The Applicative typeclass

```haskell
class Functor f ⇒ Applicative f where
  pure :: a → f a

  (<*>) ::
  f (a → b) → f a → f b
  (<*>) = liftA2 id

liftA2 ::
  (a → b → c) → f a → f b → f c
liftA2 f x = (<*>) (fmap f x)
```
The Applicative typeclass, now elaborated

```haskell
class Functor f ⇒ Applicative f where
  pure :: f a ⇒ a → f a

  (<*> :: (f a → b, f a, f b) ⇒ f (a → b) → f a → f b
              = liftA2 id

  liftA2 :: (f a, f b, f c) ⇒ (a → b → c) → f a → f b → f c
  liftA2 f x = (<*> (fmap f x) -- Typechecking fails

  Use of fmap demands f @ (b → c)
```
Partial Types Empirical Evaluation

The Applicative typeclass, elaborated and modified

\[
\text{class } (\text{Total } f, \text{Functor } f) \Rightarrow \text{Applicative } f \ \text{where}
\]

\[
\text{pure} \ : \ f @ a \Rightarrow a \rightarrow f a
\]

\[
(<*>) \ : \ (f @ a \rightarrow b, f @ a, f @ b) \Rightarrow f (a \rightarrow b) \rightarrow f a \rightarrow f b
\]

\[
(<*) = \text{liftA2 id}
\]

\[
\text{liftA2} \ : \ (f @ a, f @ b, f @ c) \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow f a \rightarrow f b \rightarrow f c
\]

\[
\text{liftA2 } f \ x = (<>)(\text{fmap } f \ x) -- \text{Typechecks}
\]
The Applicative typeclass, elaborated and modified

```haskell
class (Total f, Functor f) ⇒ Applicative f where
  pure :: f @ a ⇒ a → f a

  (<>*) :: (f @ a → b, f @ a, f @ b)
  ⇒ f (a → b) → f a → f b
  (<>*) = liftA2 id

liftA2 :: (f @ a, f @ b, f @ c)
  ⇒ (a → b → c) → f a → f b → f c
liftA2 f x = (<>*) (fmap f x) -- Typechecks
```

But now instances of Monads, MonadPlus, etc. all need a Total constraint
Who are the biggest culprits in libraries?

<table>
<thead>
<tr>
<th>Module</th>
<th>Classes and Insts, Modified/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraries</td>
<td>495/5442 (9.7%)</td>
</tr>
<tr>
<td>libraries/transformers</td>
<td>167/444 (37.6%)</td>
</tr>
<tr>
<td>libraries/base</td>
<td>78/1108 (7.0%)</td>
</tr>
<tr>
<td>libraries/mtl</td>
<td>69/80 (86.2%)</td>
</tr>
</tbody>
</table>

But why? Applicative is to blame
The Partial Applicative Problem

```haskell
instance Applicative Set where
    (<*>) :: (Set (a → b), Set a, Set b)
        ⇒ Set (a → b) → Set a → Set b
    (<*>) = ...
```

But Set (a → b) or Ord (a → b) can never be satisfied
Attempt to solve the Partial Applicative Problem
Partial Types and Applicative

Attempt to solve the Partial Applicative Problem

Use Monoidal as Monad’s superclass

```haskell
class Functor f ⇒ Monoidal f where
    pure :: f @ a ⇒ a → f a
    unit :: f @ () ⇒ f ()
    (>*<) :: (f @ a, f @ b, f @ (a, b))
        ⇒ f a → f b → f (a, b)
```

```haskell
instance Monoidal Set where -- ✓
    ...
```

```haskell
class Monoidal m ⇒ Monad m where -- ✓
    ...
```
Was the AMP a good idea?

Functor-Applicative-Monad should have been

Functor-Monoidal-Monad
What's more in the paper?

Partial

- GADTs
- Type Families: Open/Closed/Associated Types
- Data Families
- Newtypes

And more dirty details...
Summary:

- Make partial types first class
  - Generate @ constraints via elaboration
  - Support Functor and Monad instances for partial datatypes

Empirical Study

- Retrofit GHC and core libraries
- Measure code impact (< 10% change overall)

Prototype implementation:

```
GHC + {#- LANGUAGE PartialTypeConstructors -#}
github.com/IaFP/ghc
```