Partial Type Constructors in Practice

Apoorv Ingle Alex Hubers J. Garrett Morris

The University of Iowa, USA

Motivation

Kind checking rules out nonsensical types

$$(\textsc{TApp})\ \frac{\Delta \vdash \tau : \kappa \to \kappa' \quad \Delta \vdash \sigma : \kappa}{\Delta \vdash \tau\,\sigma : \kappa'}$$

Motivation

Kind checking rules out nonsensical types

`[Int]` is well defined

`Int []` is nonsensical

Motivation

Kind checking rules out nonsensical types

`[Int]` is well kinded

$$\frac{\Delta \vdash [\,] : * \to * \quad \Delta \vdash \texttt{Int} : *}{\Delta \vdash [\texttt{Int}] : *}$$

`Int []` is ill kinded

$$\frac{\Delta \vdash \texttt{Int} : * \quad \Delta \vdash \texttt{[]} : * \to *}{\Delta \vdash \texttt{Int []} : {\color{red}???}}$$

Motivation

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?

Defining Partial Types: Motivation

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

$$\frac{\Delta \vdash \texttt{Set} : * \to * \quad \Delta \vdash \texttt{Int} \to \texttt{Int} : *}{\Delta \vdash \texttt{Set}\ (\texttt{Int} \to \texttt{Int}) : *}$$

Elements of `Set` need to be ordered

$\texttt{Int} \to \texttt{Int}$ is not ordered in Haskell

Motivation

There are more partial types

```
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
```

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
   ```
2. Partial datatypes cannot leverage typeclass abstractions
   Constrained Functor Problem

```
instance Functor Set where
    fmap   :: (a → b) → Set a → Set b
 -- mapSet ::  (Ord a, Ord b) ⇒ (a → b) → Set a → Set b
    fmap = mapSet -- Typechecking fails!
```

Motivation

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

Defining Partial Types

How can we make partiality in types explicit?

Defining Partial Types

How can we make partiality in types explicit?

Define a predicate on types: $\tau \mathbin{@} \sigma$

$$\tau \mathbin{@} \sigma \text{ holds} \quad \Longleftrightarrow \quad \tau\,\sigma \text{ is well-defined}$$

How can we make partiality in types explicit?

Define a predicate on types: $\tau \mathbin{@} \sigma$

$$
\begin{aligned}
\tau \mathbin{@} \sigma \text{ holds} &\iff \tau\,\sigma \text{ is well-defined} \\
\texttt{Set} \mathbin{@} \texttt{a} \text{ holds} &\iff \texttt{Ord a} \text{ holds} \\
\texttt{Ratio} \mathbin{@} \texttt{a} \text{ holds} &\iff \texttt{Integral a} \text{ holds} \\
\texttt{UArray} \mathbin{@} \texttt{i} \text{ holds} &\iff \texttt{Ix i} \text{ holds} \\
\texttt{UArray i} \mathbin{@} \texttt{e} \text{ holds} &\iff \texttt{Unboxed e} \text{ holds} \\
\texttt{[]} \mathbin{@} \texttt{a} \text{ holds} &\iff \top \text{ holds}
\end{aligned}
$$

Defining Partial Types: Motivation

New kinding rule rules out all nonsensical types

$$(\textsc{TApp-New})\ \frac{\Delta \vdash \tau : \kappa \to \kappa' \quad \Delta \vdash \sigma : \kappa \quad \boxed{\Delta \vdash \tau \mathbin{@} \sigma}}{\Delta \vdash \tau\, \sigma : \kappa'}$$

Partial Types in Action

```
mapSet :: forall a b. (Ord a, Ord b)
       ⇒ (a → b) → Set a → Set b
```

Partial Types in Action

```
mapSet :: forall a b. (Ord a, Ord b)
       ⇒ (a → b) → Set a → Set b
```

With explicit partiality, `Set @ a` $\iff$ `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
```

Partial Types in Action

What have we managed to do?

Partial Types in Action

What have we managed to do?

```
fmap   :: (  f @ a,    f @ b) ⇒ (a → b) → f a   → f b
mapSet :: (Set @ a, Set @ b) ⇒ (a → b) → Set a → Set b
```

[Drum roll]

Partial Types in Action

What have we managed to do?

```
fmap    :: (   f @ a,    f @ b) ⇒ (a → b) → f a    → f b
mapSet  :: (Set @ a, Set @ b) ⇒ (a → b) → Set a → Set b
```

[Drum roll]

```
instance Functor Set where
    fmap = mapSet -- Typechecks!
```

Partial Types in Action

Also a `Monad` instance for `Set`

```
instance Monad Set where -- Typechecks
    return :: (Set @ a) ⇒ a → Set a
    return = ...
    (>>=) :: (Set @ a, Set @ b)
          ⇒ Set a → (a → Set b) → Set b
    (>>=) = ...
```

Defining Partial Types

Define a predicate on types: $\tau \mathbin{@} \sigma$

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

But how do we implement this in GHC?

Defining Partial Types

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)
```

Defining Partial Types

Define a predicate on types: $\tau \mathbin{@} \sigma$

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

Take 1: Use a Typeclass

```
class (@) (t :: k → k') (u :: k)

instance [] @ σ
```

Defining Partial Types

Define a predicate on types: τ @ σ

τ @ σ 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
```

Defining Partial Types

Define a predicate on types: $\tau \mathbin{@} \sigma$

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

Take 1: Use a Typeclass

$\texttt{class}\ (@)\ (\texttt{t :: k} \to \texttt{k'})\ (\texttt{u :: k})$

$\texttt{instance []}\ @\ \sigma$

$\texttt{instance Ord}\ \sigma \Rightarrow \texttt{Set}\ @\ \sigma$

$$\texttt{Ord}\ \sigma \vdash \texttt{Set}\ @\ \sigma$$

but

$$\texttt{Set}\ @\ \sigma \nvdash \texttt{Ord}\ \sigma$$

Typeclasses do not allow bidirectional reasoning

Defining Partial Types

Define a predicate on types: $\tau \mathbin{@} \sigma$

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

Take 2: Use a type family

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

Defining Partial Types

Define a predicate on types: $\tau \mathbin{@} \sigma$

$\tau \mathbin{@} \sigma$ 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

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

type instance [] @ σ = ()

type instance Set @ σ = Ord σ
```

$$\texttt{Set} \ @ \ \sigma \vdash \texttt{Ord} \ \sigma$$

also

$$\texttt{Ord} \ \sigma \vdash \texttt{Set} \ @ \ \sigma$$

Exactly what we need ✓

Partial Types Empirical Evaluation

That's all great but..

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

Defining Partial Types

Where do these @ constraints come from?

Defining Partial Types

Where do these @ constraints come from?

Elaboration

Defining Partial Types: Elaboration

Type signatures

$$(\gg\!=) \; :: \; \texttt{forall a b.} \qquad\qquad \texttt{m a} \rightarrow (\texttt{a} \rightarrow \texttt{m b}) \rightarrow \texttt{m b}$$

elaborates to

$$(\gg\!=) \; :: \; \texttt{forall a b.} \; (\texttt{m} \,@\, \texttt{a}, \; \texttt{m} \,@\, \texttt{b}) \Rightarrow \texttt{m a} \rightarrow (\texttt{a} \rightarrow \texttt{m b}) \rightarrow \texttt{m b}$$

Datatypes

```
???
```

elaborates to

```
data Set a = ...
type instance Set @ a = Ord a
```

`{-# LANGUAGE DatatypeContext #-}` to rescue

```
data Ord a => Set a = ...
```

Defining Partial Types: Elaboration

Datatypes

```
data Ord a ⇒ Set a = ...
```

elaborates to

```
data Set a = ...
type instance Set @ a = Ord a
```

Are there any programs that are no longer typeable?

Partial Types in Action

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

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

instance Functor f ⇒ Functor (Ap f) where
    fmap g (MkAp k) = MkAp (fmap g k) -- Okay

 -- fmap :: (Ap f @ a, Ap f @ b) ⇒ (a → b) → Ap f a → Ap f b
 -- fmap :: (f @ a    , f @ b)   ⇒ (a → b) → f a    → f b
```

Need more type annotations

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

```
data Ap f a = MkAp (f a)
-- Ap @ f ~ ()        Ap f @ a ~ ()
-- MkAp :: forall f a. f @ a => f a -> Ap f a
```

Partial Types in Action

Need more type annotations

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

```
data Ap f a = MkAp (f a)
-- Ap @ f ~ ()        Ap f @ a ~ ()
-- MkAp :: forall f a. f @ a => f a -> Ap f a

instance (forall a. f @ a, Functor f) => Functor (Ap f) where
    fmap g (MkAp k) = MkAp (fmap g k) -- Okay
```

Partial Types in Action

Need more annotations

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

```
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
```

Partial Types in Action

```
data         Ap f a = MkAp (f a)

data f @ a ⇒ Ap f a = 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

Partial Types Empirical Evaluation

How often is this sometimes?

Partial Types Empirical Evaluation

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

Partial Types Empirical Evaluation

How often is this sometimes?

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

Benchmark changes in types

No term changes

$< 10\%$ overall

	Classes and Insts, Modified/Total	Term Sigs, Modified/Total
compiler/GHC	133/1931 (6.9%)	218/16129 (1.3%)
libraries	495/5442 (9.7%)	412/17337 (2.8%)

Partial Types Empirical Evaluation

Who are the biggest culprits in libraries?

	Classes and Insts, Modified/Total
libraries	495/5442 (9.7%)
libraries/transformers	167/444 (37.6%)
libraries/base	78/1108 (7.0%)
libraries/mtl	69/80 (86.2%)

Top 3 account for $> 60\%$

But why?

The `Applicative` typeclass

The `Applicative` typeclass

```
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)
```

Partial Types Empirical Evaluation

The `Applicative` typeclass, now elaborated

```
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

```
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
```

Partial Types Empirical Evaluation

The `Applicative` typeclass, elaborated and modified

```
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

Partial Types Empirical Evaluation

Who are the biggest culprits in libraries?

Module	Classes and Insts, Modified/Total
libraries	495/5442 (9.7%)
libraries/transformers	167/444 (37.6%)
libraries/base	78/1108 (7.0%)
libraries/mtl	69/80 (86.2%)

But why? `Applicative` is to blame

Partial Types and Applicative

The Partial `Applicative` Problem

```
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

Partial Types and Applicative

Attempt to solve the Partial `Applicative` Problem

Partial Types and Applicative

Attempt to solve the Partial `Applicative` Problem

Use `Monoidal` as `Monad`'s superclass

```
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)

instance Monoidal Set where -- ✓
   ...

class Monoidal m ⇒ Monad m where -- ✓
   ...
```

Partial Types and Applicatives: Hot Take

Was the AMP a good idea?

Functor-Applicative-Monad

should have been

Functor-Monoidal-Monad

Whats more in the paper?

Partial

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

And more dirty details...

That's all Folks

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`