Partial Type Constructors in Practice

Apoorv Ingle
Department of Computer Science
The University of Iowa
Iowa City, Iowa, USA
apoorv-ingle@uiowa.edu

Alex Hubers
Department of Computer Science
The University of Iowa
Iowa City, Iowa, USA
alexander-hubers@uiowa.edu

J. Garrett Morris
Department of Computer Science
The University of Iowa
Iowa City, Iowa, USA
garrett-morris@uiowa.edu

Abstract

Type constructors in functional programming languages are total: a Haskell programmer can equally readily construct lists of any element type. In practice, however, not all applications of type constructors are equally sensible: collections may only make sense for orderable elements, or embedded DSLs might only make sense for serializable return types. Jones et al. proposed a theory of *partial type constructors*, which guarantees that type applications are sensible, and extends higher-order abstractions to apply equally well to partial and total type constructors. This paper evaluates the practicality of partial type constructors, in terms of both language design and implementation. We extend GHC, the most widely used Haskell compiler, with support for partial type constructors, and test our extension on the compiler itself and its libraries. We show that introducing partial type constructors has a minimal impact on most code, but raises important questions in language and library design.

CCS Concepts: • **Theory of computation** → *Type theory*; • **Software and its engineering** → **Functional languages**; **Data types and structures**.

Keywords: Type constructors, Type families, Parametric polymorphism

ACM Reference Format:
Apoorv Ingle, Alex Hubers, and J. Garrett Morris. 2022. Partial Type Constructors in Practice. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium (Haskell '22), September 15–16, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3546189.3549923

1 Introduction

In languages with parameterized types, some type expressions (`[]` or `[Int]`, say) are more meaningful than others (`[Maybe]` or `Maybe Map`). In Haskell, kind checking distinguishes meaningful type expressions. If we know that `Int` has kind ⋆ (that is: a type) and `[]` has kind ⋆ → ⋆ (that is: it constructs types from types), we can conclude that `Int` and `[Int]` are well-kinded and `[[]]` is not. However, kind-checking is not by itself enough to identify all meaningless types. For example:

- The `UArray` type constructor describes arrays of unboxable elements; type `UArray Int (Int → Int)` does not make sense, as functions cannot be unboxed.
- The type `Ratio` describes exact fractions; while `Ratio Char` has inhabitants, none of the expected `Ratio` operations apply to them.

Jones et al. [8] propose a theory of *partial type constructors* to identify such seemingly well-kinded but actually meaningless type expressions. In their approach, type constructors not only have kinds, but also participate in a *definedness relation* (written @). For the `Ratio` type constructor, `Ratio @ a` would be equivalent to `Num a`, ensuring that ratios were of numeric types; or, for unboxed arrays, we would expect `UArray @ a` to be equivalent to a constraint `Unboxable a`, ensuring that elements of unboxed arrays could be unboxed. Jones et al. extend kinding with definedness: a type application `k t` is well-kinded only when the constraint `k @ t` is satisfiable. In their system, a type like `Ratio Char → Ratio Char` is only well-kinded if `Num Char` is satisfiable.

This paper explores whether the theory of partial type constructors could be practical in modern Haskell, as realized by GHC. There are several challenges:

- Haskell's datatypes are more complicated than those considered by Jones et al., including features like kind polymorphism, existential types, and generalized algebraic data types (GADTs).
- Haskell includes type expressions not built from type constructors, such as type families and type synonyms.
- Modern Haskell code depends on highly polymorphic libraries, so partial type constructors might introduce unsustainable annotation burdens.

To evaluate these challenges, we have implemented partial type constructors as a prototype extension of GHC 9.3[1] (the version in development as we wrote this paper). Our implementation elaborates source programs—with all the features of modern Haskell, including GADTs, type families, and generic programs, as well as partial type constructors—to GHC's existing core calculus, System FC. Type constructors

Haskell '22, September 15–16, 2022, Ljubljana, Slovenia

ACM ISBN 978-1-4503-9438-3/22/09.
https://doi.org/10.1145/3546189.3549923

[1] https://github.com/IaFP/ghc

in System FC are total, so our implementation inserts constraints in the elaborated code to capture the meaning of source programs with partial type constructors. We have used our extended version of GHC to compile a number of realistic examples of Haskell programs, including the compiler itself, several motivating examples of partial type constructors, and a number of popular packages from Hackage. We found that, while partial type constructors are not entirely backwards compatible, minimal programmer effort is required to adapt most existing Haskell code to compile with partial type constructors. However, we found several significant examples of library design that (unintentionally) relies on totality of type constructors. Perhaps most prominently, while the `Functor` and `Monad` classes adapt perfectly well to partial type constructors, the `Applicative` class does not.

Contributions. This paper contributes an extension of partial type constructors to modern Haskell, and an evaluation of their impact on practical code. In particular:

- We adapt partial type constructors to GHC's datatypes (§3), including GADTs, kind polymorphism, and newtype declarations, capturing definedness using constraint families.
- We extend partial type constructors to capture partiality in type families (§4), and compare the resulting design with constrained type families [13].
- We evaluate the language design and usability impacts of these changes (§5), based on the compiler itself and its underlying libraries.

We begin by reviewing partial types in Haskell (§2), and conclude by discussing related and future work (§6).

2 Partial Types in Haskell

We begin by reviewing partial type constructors in Haskell: motivating examples of partiality in practical Haskell programming, two sources of partiality in Haskell types, and challenges to making partial types usable in practice.

2.1 Examples of Partial Type Constructors

One immediate question is the prevalence of partial type constructors: if cases like `UArray` or `Ratio` are very rare, then any amount of language change to better support them may be unjustified. Jones et al. catalog a variety of examples of partial type constructors drawn from Haskell and other typed functional languages.

Haskell 1.0. The first version of the Haskell Report [5] allows constraints to appear in data type and type synonym declarations. As an example, the report gives the type synonym declaration

```
type (Num a) ⇒ Point a = (a, a)
```

which would allow a type signature like

```
scale :: (Num a) ⇒ a → Point a → Point a
```

but reject type signatures like

```
scale :: a → Point a → Point a
```

as type variable `a` could be instantiated with non-numeric types. Only one year later, lacking a satisfactory account of the semantics of those constraints, Peyton Jones [16] proposed that constraints on type synonyms be dropped from the language entirely and constraints on datatypes weakened to constraints on the types of individual data constructors.

Monad transformers. The `mtl`[2] package defines a collection of monad transformers, allowing for a modular account of introducing and using side-effecting code. For example, if type `m` is a monad, then type `ExceptT e m` adds to `m` exceptions of type `e`. The latter type is only meaningful if `m` itself is a monad—for example, `ExceptT e Ratio` is well-kinded but not meaningful, as the `Ratio` type is not monadic. However, Haskell cannot exclude this type. As a result, many of the functions in `mtl` need seemingly extraneous `Monad` constraints, simply to exclude such pathological examples.

Collection types. The Haskell type `Set` describes sets of objects, implemented as size-balanced binary trees. To maintain its tree invariants, most operations on values of `Set a` require that elements be ordered:

```
member :: Ord a ⇒ a → Set a → Bool
union  :: Ord a ⇒ Set a → Set a → Set a
```

The most basic constructors of sets, in contrast, do not have such a requirement:

```
empty     :: Set a
singleton :: a → Set a
```

This makes it possible to construct "useless" sets, such as `singleton id`. Further, the `Set` type cannot participate in many of Haskell's higher-order abstractions. For example, while the `Set` type has a mapping operation, it requires that the element types be ordered:

```
setMap :: (Ord a, Ord b)
    ⇒ (a → b) → Set a → Set b
```

Because of the `Ord` constraints in the type of `setMap`, it is not general enough to add `Set` to the `Functor` class, and collection-generic code cannot be applied to `Set`s.

2.2 A Constraint for Definedness

The theory of partial type constructors extends the theory of qualified types [7] with two key ideas:

- A definedness constraint `k @ t`, which holds only when type constructor `k` is applicable to argument `t`; and,
- An extended kinding relation, ensuring that type applications `k t` are allowed only when the constraint `k @ t` is satisfiable.

Their kinding judgment $P \mid \Delta \vdash \tau : \kappa$ denotes that under kinding environment Δ and predicate environment P, type τ has kind κ. The novelty is the incorporation of predicates in

[2] https://hackage.haskell.org/package/mtl

kinding: P will be used to justify that any type applications in τ are well-defined. In their kinding rule for type application:

$$\frac{P \mid \Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad P \mid \Delta \vdash \tau_2 : \kappa_1 \quad P \Vdash \tau_1 \ @ \ \tau_2}{P \mid \Delta \vdash \tau_1\ \tau_2 : \kappa_2}$$

the shaded hypothesis requires that the predicate environment P entail that τ_1 is applicable to τ_2. With their kind system, we could not derive

$$\vdash \forall fab.(a \to b) \to f\ a \to f\ b : \star$$

as we cannot derive either $\Vdash f \ @ \ a$ or $\Vdash f \ @ \ b$. However, we could derive

$$\vdash \forall fab.(f \ @ \ a, f \ @ \ b) \Rightarrow (a \to b) \to f\ a \to f\ b : \star$$

Practically speaking, with partial type constructors, the type of `fmap` would have to be adjusted to assure that its type applications are well-defined:

```
class Functor f where
  fmap :: (f @ a, f @ b) ⇒ (a → b) → f a → f b
```

This change to the typing of `fmap` is essential to the idea of partial type constructors. Because `fmap` is polymorphic in f, we cannot know in advance whether or not f will be instantiated with a partial type constructor. Moreover, by the time that we know—say, when we are attempting to write a `Functor` instance for the `Set` type—it is already too late: the type of `fmap` was determined in the class declaration, not at its instances. With this definition of `fmap`, assuming that the definedness condition `Set @ a` is `Ord a`, the `Functor` instance for `Set` would be accepted and `Functor`-generic code could be applied to `Set`s. For total type cosntructors, like `[]`, the constraints `[] @ a` add no new information in defining the instance, and no additional burden when using it, and so is equivalent to the current treatment of type constructors.

2.3 Elaborating Definedness in Types

The theory of partial type constructors requires a significant number of additional constraints in even mildly polymorphic programs. This would make using partial type constructors onerous to programmers, and not backwards compatible with most existing Haskell code. Moreover, many of these constraints seem obvious from their context. Given that the type of `fmap` explicitly mentions type applications `f a` and `f b`, why should the programmer have to additionally state the assumption that they be well-defined?

To address this verbosity, Jones et al. introduce an elaboration relation on type schemes $\sigma \hookrightarrow \sigma'$. Elaboration extends a type scheme with those constraints needed to assure that it is well-defined. They show that, for all σ that are well-kinded *without* taking partial type constructors into account, σ can be augmented to a σ' that is well-kinded *with* partial type constructors. For example, applying the elaboration relation to the original type signature for `fmap` would give the type signature appropriate for partial type constructors. Applied uniformly, elaboration would seem to allow existing Haskell code to be used, without modification, in Haskell extended with partial type constructors.

To evaluate the effectiveness of elaboration, Jones et al. extended Hugs, a Haskell compiler, with the kinding restrictions and automatic type elaboration of partial type constructors. They tested the extended compiler on the Hugs libraries, an extension of the Haskell 98 standard libraries. They found that the majority of code compiled without modification. They did identify some functions that were rejected with partial type constructors, such as `mapAndUnzipM`:

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

The problem is that `mapAndUnzipM` constructs an `m [(b,c)]` list during the computation, but the constraints produced by elaboration are not sufficient to guarantee that such a type is defined. Jones et al. propose two solutions: either the function can be rewritten to use `foldM`, in which case the original type is again valid. or the type can be changed to reflect the intermediate data structure:

```
mapAndUnzipM :: (Monad m, m @ [(b, c)])
     ⇒ (a → m (b,c)) → [a] → m ([b], [c])
```

In all, they found 16 definitions that required changed type signatures.

2.4 Adopting Partial Type Constructors

The elaboration experiment of Jones et al. leaves us hopeful that partial type constructors might be not just theoretically appealing but also practically viable. However, several significant questions remain in extending partial type constructors to modern Haskell.

Other Haskell features. Jones et al. consider a simple core language. Practical Haskell programs, however, use a variety of features that might interact with partial type constructors, including newtype definitions and derived instances, type synonyms, generalized algebraic data types, and type families. How do partial type constructors interact with each of these features?

Practical applications. Jones et al. did not extend Hugs to allow programs to introduce new partial type constructors, so their experiment could not extend to uses of partial type constructors. Can practical examples of partiality in Haskell be expressed more simply using partial type constructors?

Backward compatibility. The Hugs standard libraries, while they include interesting examples of polymorphic code, may not be representative of modern Haskell. Furthermore, it is unclear how common examples like `mapAndUnzipM` would

be over a larger sample of Haskell code. Is elaboration actually sufficient for backward compatibility in practice?

3 Implementing Partial Type Constructors

In this paper, we evaluate the impacts of partial type constructors on modern Haskell language and library design. To do so, we have built a version of GHC extended with support for partial type constructors. We have modified GHC's type inference algorithm to automatically introduce the constraints that would be required by the kinding relation of partial type constructors, following the elaboration relation of Jones et al. Our implementation supports (and enforces) the use of partial type constructors in the source language, preserving unchanged the existing compilation machinery (and metatheory) of System FC.

3.1 A Constraint Family for Definedness

The crux of the theory of partial type constructors is the definedness relation `k @ t`. When implementing partial type constructors in Haskell, then, we might hope to just introduce `(@)` as a new type class, and reuse all of the compiler's existing support for automatically deriving and using type classes. However, this is not the case. Recall the `Set` example:

- If we have a term `x` of type `T`, such that `Ord T` holds, then we should be able to conclude `Set @ T`, as we might need to build a set `singleton x`.
- If we have assumed `Set @ T`, such as in the body of `fmap`, we should be able to conclude `Ord T`, allowing us to call `setMap`.

In short, we expect that `Set @ a` $\Longleftrightarrow$ `Ord a`. However, this is not how type classes work in Haskell. While we could define a type class `(@)` and populate it with instances like `Ord a` $\Rightarrow$ `Set @ a`, those instances would only allow us to conclude `Set @ a` from `Ord a`, not the other way around.

To capture the intended behavior of the definedness constraint, we instead introduce it as an indexed constraint family [14, 17]:

```
type family (@) (k :: a → b) (t :: a) ::
    Constraint
```

Instances of this family, such as `Set @ a`, are not treated as new predicates with their own instances and superclasses, but instead are equated to existing predicates. For example,

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

introduces a type equation `Set @ a` $\sim$ `Ord a`, which GHC will use symmetrically either to rewrite `Set @ a` assumptions to `Ord a` assumptions (as needed for the `Functor Set` implementation) or to turn proofs of `Ord a` into proof of `Set @ a` (as needed for the typing `singleton x`).

3.2 Extending Elaboration to Modern Haskell

With the `(@)` family defined, our compiler extension applies elaboration to all types that appear in source files, whether in type signatures, annotations within expressions, or class declarations. Figure 1 recasts the elaboration relation of Jones et al. to encompass a representative subset of the features of GHC. Our kind language includes the kinds of types $\star$ and of constraints `Constraint`. Our type language includes type constructors, type synonyms, and type families; we assume that type family applications and type synonym instances appear fully saturated. We include data type declarations (where K stands for term-level data constructors) and open type family instances; other declaration constructs will be handled similarly.

Our elaboration relation $\tau \hookrightarrow P|\tau'$ denotes that type τ elaborates to type τ', and is well-defined given constraints P. By applying elaboration to all sources of types (type signatures, type of data constructors, and so forth), we can guarantee that types are well-defined without having to modify GHC's kind checking or kind inference.

Type applications $\tau_1\ \tau_2$ are well-defined if τ_1 and τ_2 are individually well-defined, and if the application itself is meaningful. We must be careful about that final condition. In a type class constraint `Ord T`, while we want to make sure that `T` is well-defined, we do not want to introduce the additional constraint `Ord @ T` (as this would simply duplicate the existing role of type classes). To capture this distinction, we have two elaboration rules for type applications $\tau_1\ \tau_2$, conditional on the kind of the operator τ_1. If the (eventual) result of τ_1 is of kind `Constraint`, then we do not emit an additional τ_1 `@` τ_2 constraint. We do, of course, still require that τ_1 and τ_2 be individually well-defined. If the (eventual) result of τ_1 is not of kind `Constraint`, then we do emit the definedness constraint.

In elaborating a quantified type (E$\forall$), we distinguish between those constraints (P_1) that limit the bound type variable, and those (P_2) that only refer to free type variables. The result of elaboration is both a new quantified type capturing (the reduction of, see the next section) P_1, and a set of constraints P_2 to propagate farther.

3.3 Reducing Definedness Constraints

Elaboration can (frequently) generate constraints that we know are satisfiable. For example, in elaborating the type of `map`, we would generate constraints `(→) @ a`, `(a →) @ b`, `[] @ a` among many others. What should we do about these constraints?

The first answer is that we should eliminate as many constraints as possible when elaborating, by invoking GHC's existing predicate solver. Unfortunately, it can happen that we elaborate the same signature in different contexts. In particular, to break cyclic module dependency, GHC uses `.hs-boot` files that contain (limited subsets of) the type signatures from each module. We need to elaborate signatures in `.hs-boot` files. However, we may not have access to all the instances when elaborating the `.hs-boot` file that we

Syntax

Type variables	α	Kinds	$\kappa ::= \star \mid \texttt{Constraint} \mid \kappa \to \kappa$
Type constructors	C	Types	$\tau ::= \alpha \mid C \mid \tau_1\ \tau_2 \mid F\ \overline{\tau} \mid S\ \overline{\tau} \mid \tau_1 \Rightarrow \tau_2 \mid \forall \alpha{:}\kappa.\tau$
Type families	F	Declarations	$D ::= \texttt{data}\ P \Rightarrow T\ \overline{\alpha} = \overline{\forall \overline{\beta}.P_i \Rightarrow K_i\ \overline{\tau_{ij}}}$
Type synonyms	S		$\mid\ \texttt{type instance}\ F\ \tau_1 \ldots \tau_n = \tau$

$$\boxed{\tau \hookrightarrow P \mid \tau}$$

$$(\textsc{evar})\ \frac{}{\alpha \hookrightarrow \epsilon \mid \alpha} \qquad (\textsc{econst})\ \frac{}{C \hookrightarrow \epsilon \mid C} \qquad (\textsc{eapp})\ \frac{\tau_i \hookrightarrow P_i \mid \tau_i'}{\tau_1\ \tau_2 \hookrightarrow P_1, P_2, \tau_1\ @\ \tau_2 \mid \tau_1'\ \tau_2'}\ (\tau_1 : \kappa, \mathsf{last}(\kappa) \neq \texttt{Constraint})$$

$$(\textsc{e}\pi)\ \frac{\tau_i \hookrightarrow P_i \mid \tau_i'}{\tau_1\ \tau_2 \hookrightarrow P_1, P_2 \mid \tau_1'\ \tau_2'}\ (\tau_1 : \kappa, \mathsf{last}(\kappa) = \texttt{Constraint}) \qquad (\textsc{efamapp})\ \frac{\tau_i \hookrightarrow P_i \mid \tau_i'}{F\ \overline{\tau_i} \hookrightarrow P_i, WD_F\ \overline{\tau_i'} \mid F\ \overline{\tau_i'}}$$

$$(\textsc{esyninst})\ \frac{\tau_i \hookrightarrow P_i \mid \tau_i' \quad \tau[\tau_i/\alpha_i] \hookrightarrow P \mid \tau'}{S\ \overline{\tau_i} \hookrightarrow P, \overline{P_i} \mid \tau'}\ (S\ \overline{\alpha_i} = \tau) \qquad (\textsc{e}\forall)\ \frac{\tau \hookrightarrow P_1, P_2 \mid \tau' \quad P_1 \rightsquigarrow^\star P_1' \not\rightsquigarrow}{\forall \alpha{:}\kappa.\tau \hookrightarrow P_2 \mid \forall \alpha{:}\kappa.P_1' \Rightarrow \tau'}\ (\alpha \notin ftv(P_2))$$

$$(\textsc{e}\Rightarrow)\ \frac{\tau_i \hookrightarrow P_i \mid \tau_i' \quad \bigcup_i P_i = Q, Q' \quad Q' \rightsquigarrow^\star Q'' \not\rightsquigarrow}{\tau_1 \Rightarrow \tau_2 \hookrightarrow Q \mid Q'' \Rightarrow \tau_1' \Rightarrow \tau_2'}\ (ftv(Q) \cap ftv(\tau_1) = \emptyset)$$

$$\boxed{D \hookrightarrow \overline{D}}$$

$$(\textsc{edata})\ \frac{\tau_{ij} \hookrightarrow Q_{ij} \mid \tau_{ij}' \quad \bigcup_j Q_{ij} \rightsquigarrow^\star Q_i \not\rightsquigarrow}{\texttt{data}\ P \Rightarrow T\ \overline{\alpha} = \overline{\forall \overline{\beta}.P_i \Rightarrow K_i\ \overline{\tau_{ij}}} \hookrightarrow \texttt{data}\ P \Rightarrow T\ \overline{\alpha} = \overline{\forall \overline{\beta}.(P_i, Q_i) \Rightarrow K_i\ \overline{\tau_{ij}'}}}$$

$$(\textsc{efaminst})\ \frac{\tau \hookrightarrow P \mid \tau'}{\texttt{type instance}\ F\ \tau_1 \ldots \tau_n = \tau \hookrightarrow \texttt{type instance}\ F\ \tau_1 \ldots \tau_n = \tau'\ ;\ \texttt{type instance}\ WD_F\ \tau_1 \ldots \tau_n = P}$$

$\boxed{\mathsf{last}(\kappa)}$

$$\mathsf{last}(\kappa' \to \kappa) = \mathsf{last}(\kappa) \qquad \mathsf{last}(\kappa) = \kappa$$

$\boxed{P \rightsquigarrow Q}$

$$C\ @\ \tau, P \rightsquigarrow P \quad \text{if } C \in \mathcal{K} \qquad \pi, P \rightsquigarrow \pi, Q \quad \text{if } P \rightsquigarrow Q$$

where $\mathcal{K} = \{\texttt{IO}, \texttt{Maybe}, [], \texttt{Ratio}, \texttt{StablePtr}, \texttt{Ptr}, \to, \upsilon\ \to\}$

Figure 1. Elaboration for a representative subset of modern Haskell features.

would when elaborating the full module; applying those instances in elaboration would result in a module that does not match its `.hs-boot` file.

The second, less satisfactory answer, is that we should leave all constraints generated in elaboration. While we would expect this to have run-time consequences (we would be adding many unnecessary arguments to even the simplest functions), it would be stable and would surely not reject well-typed code. Again, however, this is not the case. In FFI declarations, type signatures serve both to specify the Haskell side and the expected C side of a function's type. For example:

```
foreign import ccall unsafe "HsBase.h link"
   c_link :: CString → CString → IO CInt
```

Here, if we elaborated the type of `c_link`, we would generate a type that no longer aligned with the number of arguments expected by its C implementation.

Our unhappy middle ground is to maintain a list of type constructors that we know are total—such as `(→)`, `IO`, and so forth—and to eliminate definedness constraints for those, and only those, types. The resulting process is stable (because our list of known-total types does not change) and does not break the run-time system (because we have included the types that appear in its FFI calls). It does, however, still result in generating inefficient code, as we will generate definedness constraints for many total type constructors, including all user-defined total type constructors.

3.4 Datatype Definitions

Contexts in datatype definitions generate equations for the (@) constraint family. For example, the definition

```
data Ord a ⇒ BST a =
    Leaf | Branch a (BST a) (BST a)
```

generates the type family instance

```
type instance BST @ a = Ord a
```

For datatypes with multiple parameters, we introduce constraints as soon as all the relevant type variables are available. For example, a definition

```
data (C a, C b, D a b) ⇒ T a b = ...
```

generates the type family instances

```
type instance T @ a = C a
type instance T a @ b = (C b, D a b)
```

This is very much not the same as augmenting the original data type definition with `Ord` constraints:

```
data BST a where
  Leaf   :: Ord a ⇒ BST a
  Branch :: Ord a ⇒ a → BST a → BST a → BST a
```

With such a definition, `BST a` is defined for all `a`, but its constructors each carry `Ord a` dictionaries. With partial type constructors, any usage of `BST a` must be in a context in which `Ord a` holds; incidentally, this means that we do not need to carry evidence for `Ord a` in the data type itself.

Inferring Datatype Contexts. In the context of the previous examples, consider the definition:

```
data Settish a = One a | Many (BST a)
```

Should the `Settish` datatype itself be partial? For Jones et al., the answer is yes: a datatype is only considered defined in those cases in which all of its constructors are defined. They define an elaboration relation for data types which augments a datatype's declared constraints with the constraints from each constructor's argument types. This simplifies their core calculus: as `Settish @ a` must guarantee the well-definedness of each of its constructors, no constructor needs to carry additional dictionaries to justify the well-definedness of its arguments. However, it requires care for recursive data types, particularly non-regular recursive data types. They account for this declaratively, by requiring that the elaborated constraints are sufficient to entail the required constraints.

That is not the only sensible approach, however. An alternate interpretation of the `Settish` datatype is that it should be total, inhabited by the `One` constructor at all types, and the `One` and `Many` constructors at types in `Ord`. Nor is the `One` constructor necessary for this interpretation: Haskell includes empty types (modulo divergence) and, via GADTs, type constructors that are empty at only some of their instantiations. Under this interpretation, the constraint `Settish @ a` guarantees nothing; in particular, it does not guarantee that `BST @ a` holds. Instead, the `Many` constructor must itself carry the evidence that its argument is well-defined, and so has type `BST @ a ⇒ BST a → Settish a`. Even in this case, we store one `Ord a` dictionary for the entire `BST a` value, not one `Ord a` dictionary at each node. This approach is also more flexible: the programmer is free to define `Settish` as a partial type constructor as well, if that is their intention.

This is a language design trade-off: the first approach can surprise programmers with types that are more partial than they expect, while the second can surprise programmers with types that are less populated than they expect. For our implementation, we have chosen the second approach (see (EDATA) in Figure 1). That is: while our elaboration procedure automatically accounts for *uses* of partial type constructors, we do not automatically *introduce* partial type constructors. The (reduced) set of constraints needed to justify the typing of a particular constructor is captured locally in its type.

This approach extends naturally to GADTs. For example, given the data type

```
data T a where MkT :: Int → T Int
```

we compute the elaboration of `MkT` based on the "Henry Ford encoding" [10] of its type:

```
MkT :: forall a. a ~ Int ⇒ Int → T a
```

Totality. Consider the now-canonical definition of the free monad over a functor `f`:

```
data Free f a = Pure a | Impure (f (Free f a))
```

What constraints should we have on this type? Without further annotation, our implementation would only constrain the `Impure` constructor to guarantee that `f` is applicable to `Free f a`. This constraint is, however, insufficient to prove that `Free f a` is a `Monad`: in the implementation of the bind operator, we need to generate a proof that `f @ Free f b` from the assumption that `f @ Free f a`. If we could instead make `Free` a partial type constructor (such that `Free f @ a` is `f @ (Free f a)`), we would then be able to write the `Monad` instance. But this constraint is self-referential: to show that `Free f a` is a meaningful type, we must make a statement about `Free f a`. (Jones et al. interpret @ constraints coinductively, so this would not be problematic in their approach.) Instead, we might prefer to simply require that `f` be defined everywhere—that is to say, that `f` is *total*; then, its use in `Free f a` would be uncontroversial.

Jones et al. suggest that we could use quantified constructors [1] to capture totality. For example, we might define:

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

and then set `Free @ f` to require `Total f`. In current GHC, this approach *almost* works; unfortunately, while GHC is

happy with quantified constraints, it is less happy with quantified constraint families. Eisenberg (personal communication) showed us a workaround:

```
class f @ a     ⇒ At f a
instance f @ a ⇒ At f a
type Total f   = forall a. At f a
```

The compiler is skeptical about class `At` but will accept it, and so this is sufficient to express totality. We can then add the constraint `Total f` to the definition of `Free`, and define its instances as expected.

Kind Polymorphism and Datatype Promotion. GHC's type system includes two more features that complicate type definition: kind polymorphism [23] and datatype promotion [24]. In our implementation, these features turn out to be essentially orthogonal to partial type constructors, although they each offer interesting potential for future work.

GHC allows datatype declarations to freely mix kind and type parameters; for an artificial example:

```
data T (a :: ⋆) k (b :: k) (c :: k → ⋆ → ⋆)
    where K :: c b a → T a k b c
```

GHC has a *Type* : *Type* type theory, so one might think that the kind parameter (`k`) would behave no differently from (other) type parameters. However, because `k` can appear in the kind signatures of the remaining type arguments, it is actually a limited form of (type) dependency. This is expressed using `forall` in kind signatures

```
T : ⋆ → forall k → k → (k → ⋆ → ⋆) → ⋆
```

We do not extend partial type constructors to such dependent type constructors; we expect that we have lost little expressiveness here, as class constraints over kinds are, while theoretically no different from class constraints over types, uncommon (to say the least) in current practice.

Datatype promotion allows datatypes to automatically be promoted to kinds, with their (value) constructors automatically promoted to type constructors; for example, a datatype of length-indexed lists could be indexed by the promoted datatype of natural numbers. GHC does not promote constructors with constraints—as indeed it must not, as (without partial type constructors) such constraints could not be enforced once promoted; Correspondingly, we extend elaboration of type applications (EAPP) to emit no definedness constraints for applications of promoted type constructors. We could imagine extending datatype promotion to promote partial type constructors to partial kind constructors, and data constructors with constraints to partial type constructors, but as our goal is evaluating how existing code adapts to partial type constructors, we leave this to future work.

3.5 Unexpected Existentials and Inert Constraints

There is, unfortunately, one significant downside to attaching definedness constraints to constructors. Consider a case block that scrutinizes a value of the `Settish` type. In the `Many` branch, the pattern match will introduce not only a value of type `BST a`, but also a constraint `BST @ a`. That constraint is rewritten to `Ord a`, which causes no further problem. The same is not true in more general cases:

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

This code seems simple enough; indeed, without partial type constructors, GHC is happy to conclude that `k` is well-typed at `Ap f a → f a`. With partial type constructors, the story is more complicated. Now, we interpret the constructor `MkAp` as additionally carrying the well-definedness constraint `f @ a`. As before, GHC generates a fresh unification variable β for the result type of `k`, and wants to unify β with `f a` to type the right-hande side of the equation. However, because this is the case for `MkAp`, which has provided `f @ a`, GHC must actually generate an *implication constraint* [17] `f @ a` $\supset \beta \sim$ `f a`. Being unable to further simplify the antecedent, GHC is unable to solve the implication constraint and so cannot infer a type for `k`.

Our first reaction was that GHC was being overly cautious: while we could not (yet) resolve the antecedent, we knew that it was a definedness constraint. Surely such constraints could not threaten principality, and so the implication constraint ought to be solvable. Unfortunately, things are not so simple:

```
data a ~ b ⇒ D a b = MkD a
```

Now constraint `Ap (D a) @ b` equivalent to `a ~ b`, and (returning to the previous example) `k` could equally well be given the types `Ap (D a) b → D a b` and `Ap (D a) b → D a a`. So GHC is right to decline to solve equations guarded by such constraints.

Our solution is to capture our original intuition: definedness constraints *ought not* give rise to type equalities, and then solving implication constraints with antecedent definedness constraints would be no problem. We term constraints that do not give rise to type equalities *inert*:

- A constraint `C t u ...`, for some class `C`, is inert if all of `C`'s superclass constraints are inert
- A type family application `F t u ...`, for some type family `F`, is inert if the right-hand sides of all of `F`'s equations are inert.

We require datatype contexts be inert.

We can check whether classes and their superclasses are inert. Inertness for a type family, however, must be guaranteed from the start: for open type families, we cannot be sure what equations will be added in the future. As we are only interested in a limited number of inert constraints (definedness constraints `f @ a` and those for type families introduced in the next section), building in knowledge of inert type families was sufficient. For more general usage, however, programmers would likely require a way to prescribe that

certain type families be inert, and for the compiler to check this condition for equations of those type families.

Our definition of inertness rejects some conditions that might be useful. For example, given some metric `Size` on types, we could imagine a restricted form of `Either`:

```
data Size t ~ Size u ⇒
    REither t u = RLeft t | RRight u
```

This constraint is clearly not inert. On the other hand, so long as `Size` is not injective, it could not give rise to equalities on either type `t` or `u`. Without having more experience with partial type constructors in practice, it is difficult to guess how much of a limitation this would be.

3.6 Newtype Definitions

Newtype declarations give new names to existing types. In GHC, this is realized using coercions, allowing a value of a newtype to be converted to and from its underlying type without introducing boxing or runtime cost. For partial type constructors, however, this poses a problem. Contrast the following declarations:

```
data Cmp1 f g a    = Mk1 {unCmp1 :: f (g a)}
newtype Cmp2 f g a = Mk2 {unCmp2 :: f (g a)}
```

Each represents the composition of type operators `f` and `g`, and in each case the constructor's type is only well-defined when the constraints `g @ a, f @ g a` hold. However, even when those constraints do not hold, `Cmp1` can still be well-defined and inhabited by ⊥. The same is not true of `Cmp2`: every value of `Cmp2`, even ⊥, corresponds to a value of `f (g a)`, so if the latter is ill-defined, we cannot have any of the former.

We still choose not to try to infer the constraints on newtype declarations, however. First, newtype declarations support arbitrary, including non-regular, recursion, so inferring constraints could introduce non-termination. Second, it would make the programmer's experience of newtypes—which might be more partial than their declarations seemed—inconsistent with that of datatypes. Instead, we introduce additional validation for newtype declarations: such a declaration is only well-formed if the declared constraints entail the well-definedness of the right-hand side. Concretely, we reject the declaration of `Cmp2` above, but accept

```
newtype (g @ a, f @ g a) ⇒
    Cmp2 f g a = Mk2 {unCmp2 :: f (g a)}
```

4 Implementing Partial Type Families

Partial type constructors are not the only source of partiality in Haskell types. Indexed type families [2, 17] are open mappings from types to types; they provide Haskell programs with modular type-level computation. For example, we could define a type family that maps collection types, such as lists or sets, to their element types:

```
type family Elem (a :: ⋆) :: ⋆
type instance Elem [a] = a
type instance Elem (Set a) = a
```

With only these instances in scope, type family applications like `Elem Bool` or `Elem (Tree a)` (given some parametric type `Tree`) do not reduce. However, as type families are open, such instances could be added to the family later. In other cases, type family applications can never reduce to a type:

```
type family Loop :: ⋆
type instance Loop = [Loop]
```

With this definition, the type `Loop` can never rewrite to a type free of type family applications. Nevertheless, avoiding potential unsoundness arising from types like `Loop` introduces significant complexities to other features of Haskell's type system, such as closed type families [4] and injective type families [19].

To account for non-terminating type families applications, Morris and Eisenberg [13] proposed *constrained type families*. In their proposal, type families `F` could only be defined associated to type classes `C`, and applications of `F` would have to be guarded by `C` constraints. Because GHC cannot generate infinite type class dictionaries, any uses of diverging type families would be guarded by unsatisfiable constraints. They show that, as a result, features like closed type families could be made simpler and more intuitive without compromising type soundness.

Partial type constructors introduce another way type family applications may be undefined.

```
type family Underlying (a :: ⋆) :: ⋆
type instance Underlying [a]      = Set a
type instance Underlying (Tree a) = Set a
```

The type family application `Underlying [Int]` sensibly rewrites to `Set Int`. On the other hand, the type family application `Underlying [Int → Int]` is problematic: the equations suggest it should rewrite to `Set (Int → Int)`, but the latter is not well-defined.

4.1 Constraint Families for Definedness

We might hope to apply our existing machinery to type families as well: perhaps we could introduce an equation `Underlying @ t ~ Set @ t`, for example. At the moment, however, type family applications must always be fully applied: we cannot refer to `Underlying` without an argument. Instead of a single definedness constraint for type family applications, we generate a new definedness family mirroring each source type family. For each type family F, with parameters $\alpha_i : \kappa_i$ and result kind κ we define a new type family WD_F[3] with identical parameters $\alpha_i : \kappa_i$ and result kind `Constraint`. To avoid namespace pollution, the names of definedness families are not programmer-visible. We can

[3]…actually named `$wd:F`, but that's a bit of a mouthful.

then elaborate type family applications, generating references to the appropriate *WD* constraint—see (EFAMAPP) in Figure 1.

While the majority of uses of type families are reflected in their types, this is not always the case. For example, GHC includes the following utility function for pattern AST nodes:

```
hsConPatArgs :: (UnXRec p)
    ⇒ HsConPatDetails p → [LPat p]
hsConPatArgs (PrefixCon _ ps) = ps
hsConPatArgs (RecCon fs)      =
    map (hfbRHS ∘ unXRec @p) (rec_flds fs)
hsConPatArgs (InfixCon p1 p2) = [p1,p2]
```

The use of `unXRec` in the second equation is at type

```
XRec p (HsFieldBind (XRec p (FieldOcc p))
                    (XRec p (Pat p))))
```

whose definedness is not guaranteed by elaboration of the type of `hsConPatArgs`.

We introduce a class for well-defined types; as well-definedness is guaranteed by elaboration, it includes all types:

```
class WDT a
instance WDT a
```

We can then use `WDT` constraints to induce elaboration, without requiring access to the underlying definedness families:

```
hsConPatArgs :: (UnXRec p,
   WDT (XRec p (HsFieldBind (XRec p (FieldOcc p))
                            (XRec p (Pat p)))))
   ⇒ HsConPatDetails p → [LPat p]
```

This approach is not just applicable to type families; for example, we could replace the `(@)` constraint in the type of `mapAndUnzipM` (§2.3) with:

```
mapAndUnzipM :: (Monad m, WDT (m [(b, c)]))
    ⇒ (a → m (b,c)) → [a] → m ([b], [c])
```

4.2 Type Family Definitions

We elaborate type family instances to derive instances of their well-definedness families (see (EFAMINST) in Figure 1). We do not need to elaborate the types on the left-hand side of the equation—their definition will be guaranteed at the type family application. If the right-hand side of a type family equation τ elaborates to τ' with constraints P, then we replace the right-hand side of the original equation with the new type τ' and create an equation for WD_F with right-hand side P. (As type family argument and results cannot be quantified, we know that τ and τ' will be the same.)

Our treatment of closed type families is parallel. A closed type family definition of family F gives rise to a closed type definition of family WD_F, in which the i^{th} equation corresponds to the elaboration of the i^{th} equation of the original definition. This can lead to requiring constraints that are not strictly speaking necessary. The following example comes from GHC's internal encoding of extensible data types:

```
data Pass = Parsed | Renamed | Typechecked
type family IdGhcP pass where
  IdGhcP 'Parsed      = RdrName
  IdGhcP 'Renamed     = Name
  IdGhcP 'Typechecked = Id
```

We generate a mirroring definedness family:

```
type family $wd:IdGhcP pass :: Constraint where
  $wd:IdGhcP 'Parsed      = ()
  $wd:IdGhcP 'Renamed     = ()
  $wd:IdGhcP 'Typechecked = ()
```

Unfortunately, even though `$wd:IdGhcP` rewrites to the empty context for every argument, GHC cannot discharge the generic constraint `$wd:IdGhcP p`. If we changed the last equation to `IdGhcP t = Id` (and thus the generated equation as well), the generic constraint would rewrite.

Associated types are treated as if they were stand-alone open type families. The associated type

```
class Collect c where
  type Elem c
  insert :: Elem c → c → c
```

is elaborated to

```
class Collect c where
  type Elem c
  type $wd:Elem c :: Constraint
  insert :: Elem c → c → c
```

and uses of `Elem` are elaborated to be guarded by `$wd:Elem` constraints. It might seem more natural to use the constraint `C t` to guarantee the well-definedness of `Elem c`; unfortunately, associated types are not required to mention all the type variables of their parent classes:

```
class C t u where type F t :: ★
```

Now we could not elaborate type signatures that mention `F t` to have constraints `C t u`, as `u` would be ambiguous.

Data families behave like ordinary type constructors: they can appear partially applied and have → kinds. This means that we can treat data families as other type constructors. Data family applications are elaborated as type constructor applications and data family instances are elaborated as data type declarations.

4.3 Type Synonyms

Type synonyms, despite their long pedigree, have more in common with type families than they do with Haskell's other type constructors. In particular, type synonyms are not first-class entities, but can only appear fully applied. We could almost think about a type synonym as a kind of degenerate type family: a synonym declaration `type S a1 .. an = t` would be interpreted as

```
type family S a1 ... an where S a1 ... an = t
```

This does not work in practice; for example, type synonyms can have quantified types on their right-hand sides, but type family equations cannot. It does suggests one way that we could extend elaboration to type synonyms. Each type synonym S would give rise to another type synonym WD_S, capturing its definedness constraints; each use of S would be guarded by a use of WD_S. For our implementation, however, the generality of this approach is not necessary. At the time that we elaborate type synonym applications, we have access to the right-hand side of the synonym definition. Rather than introducing new definitions, we simply elaborate type synonyms instances to the elaboration of their right-hand sides directly (see (ESYNINST) in Figure 1).

4.4 Constrained Type Families

The key idea that Morris and Eisenberg [13] develop in constrained type families is that many of the current complexities in type families could be simplified if we could guarantee that diverging or otherwise undefined type family applications could never be used. They guarantee this by requiring that every type family application $F\,\overline{\tau_i}$ be guarded by a constraint $X_F\,\overline{\tau_i}$, satisfiable exactly when the type family application reduces to a ground type. In their approach, although they do not provide any guarantee that type family applications appearing in a program terminate, they assure that any code whose typing depends on a non-terminating type family application is unreachable. This allows them to relax restrictions on closed type families (particularly the use of infinitary unification in closed type family matching) without compromising the type safety of the resulting system. Two questions arise in relation to our work. First, in guaranteeing that type family applications are well-defined, have we actually established the same conditions required by Morris and Eisenberg? Second, if so, does that mean that we can gain the same benefits?

We conjecture that the answer to the first question is yes. A type family declaration like

```
type family Loopy :: ⋆
type instance Loopy = [Loopy]
```

gives rise to a well-definedness family

```
type family $wd:Loopy :: Constraint
type instance $wd:Loopy = $wd:Loopy
```

and every use of `Loopy` is guarded by a constraint `$wd:Loopy`. But the constraint `$wd:Loopy` cannot be discharged (because its rewriting can never reach a ground constraint), and so code that types using `Loopy` is unreachable.

We have not, however, relaxed the restrictions on closed or injective type families as proposed by Morris and Eisenberg. Unlike in their work, in which looping type famiy applications are guarded by unsatisfiable constraints, we only guard looping type family applications by other looping type family applications. Showing safety properties comparable to theirs in our context would require a fine characterization of the meaning and role of well-definedness constraints, which we leave to future work.

4.5 Partially-Applied Type Families

Recently, Kiss et al. [9] developed a generalization of type families that permits their partial application. They introduce a new kind constructor ↠, used for "unmatchable" type constructors like type families and type synonyms. In their proposal, the `Elem` type family would have kind `Type ↠ Type`; this would permit its partial application, but would not allow, for example, a `Functor` instance for `Elem`. Kiss et al. further propose matchability polymorphism, allowing definitions to range over both ordinary (in their terms: matchable) constructors and unmatchable constructors. With this feature, we could have defined a single constraint family applicable to both type constructors and type families:

```
type family Ap (m :: Matchability)
    (k :: a →ᵐ b) (t :: a) :: Constraint
```

This would have eliminated the need for introducing (and tracking) a new definedness family for each source type family, but would not, we conjecture, otherwise significantly change our elaboration approach. Support for unsaturated type family applications and matchability polymorphism had not yet landed in GHC when we wrote this paper, but we look forward to exploring this in future work.

5 Impact of Partial Type Constructors

Having an implementation of partial type constructors in GHC allows us to start answering two questions: first, if our implementation realizes practical uses of (intuitively) partial type constructors; second, what impact introducing partial type constructors would have on existing Haskell code. To begin answering these questions, we have rebuilt GHC itself (and the packages upon which it depends, such as Cabal) with the partial type constructors extension enabled for all code, and measured the necessary changes. In our initial measurements, we have not taken advantage of partial type constructors; this is an attempt to measure the worst-case overhead for programmers who would have to adapt to a language with partial type constructors. Our results are shown in Figure 2; we separate out annotations added to instances and superclasses from annotations appearing in ordinary type signatures. For both the compiler and its libraries, we list both the total changes needed and those top-level directories or libraries which contained non-zero numbers of changes. Most of the compiler required very little annotation—we changed about 7% of classes and instances, and about 2% of function signatures.

We have also re-engineered several of these libraries to begin capturing the potential benefits of partial type constructors. Our applications of partial type constructors were

effective at capturing invariants, as measured by reduction in the number of constraints needed to enforce invariants.

5.1 What Makes an Applicative?

The Applicative class [11] characterizes effectful programming with pure flow of control:

```
class Functor f ⇒ Applicative f where
  pure   :: a → f a
  (<*>)  :: f (a → b) → f a → f b
  liftA2 :: (a → b → c) → f a → f b → f c
```

We know that, with partial type constructors, we can make the BST type an instance of Functor. It is not clear whether we ought make it an instance of the Applicative class as well. The challenge is indicated by the elaborated type of (<*>), here specific to BST:

```
(<*>) :: (Ord (a → b), Ord a, Ord b)
      ⇒ BST (a → b) → BST a → BST b
```

While GHC cannot be sure that it will never come across an instance of Ord for functions, we may be fairly certain of difficulty in using this operator. One difficulty appears almost immediately; the default implementation of liftA2

```
liftA2 f fa fb = pure f <*> fa <*> fb
```

requires additional f @ a → b → c, f @ b → c constraints not present in its type signature.

To be clear: the difficulty we are identifying here is not with the foundational account of applicative functors and binary search trees, but with our ability to realize that account. If types a and b are ordered, we can describe an ordering of a to b functions suitable for <*>. However, implementing this ordering in a practical language would introduce not insignificant computational overhead. We also cannot hope to limit ourselves to singleton search trees of functional type: in the implementation of liftA2, for example, while we start with a singleton tree of functions, the result of the first <*> is no longer a singleton.

We can observe a similar situation if we compare the elaborated types for the Monad instance of BST:

```
(>>=) :: (Ord a, Ord b)
      ⇒ BST a → (a → BST b) → BST b
join  :: (Ord a, Ord (BST a))
      ⇒ BST (BST a) → BST a
```

As with the type of <*>, the type of join requires an awkward definedness constraint, which would be inherited by >>= were it defined in terms of join. However, as Monad instances are defined in terms of >>=, the extra definedness constraint only appears in cases where join is used directly.

McBride and Paterson [11] propose just such an alternate account of applicative as well:

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (⋆) :: f a → f b → f (a, b)
```

The type of ⋆ poses no problem for partial type constructors; for example, it is trivial to implement such an operation for binary search trees. And, the operations of Applicative—including the *n*-ary liftA functions—are all definable in terms of ⋆ while introduce no unexpected constraints.

Changing the Applicative class in this way is a more significant change than we intended to make for our evaluation. It requires rewriting all existing instances of Applicative. More significantly, however, to benefit from a version of Applicative friendly to partial type constructors, *uses* of <*> need to be changed to ⋆, or to a liftA function. To complete our evaluation, we have adopted a more light-weight, if less satisfactory, approach: adding Total constraints to many of the Applicative instances in GHC and its libraries. Of course, this burden is felt mostly in libraries that define monadic abstractions; this explains the high change counts in libraries/transformers and libraries/mtl.

5.2 Opportunities for Partial Type Constructors

In adapting GHC, we also discovered several possible applications of partial type constructors within the compiler and its libraries. These examples demonstrate the potential of partial type constructors to simplify and generalize even well-engineered and time-tested Haskell programs.

The containers package contains a number of data structures that rely on ordered elements. By adding Ord constraints to these datatypes, we were able to reduce the number of Ord constraints appearing in instances from 16 to 3, and the number of Ord constraints appearing in top-level signatures from 132 to 11.

The data-partition and PSQueue packages provide data structures with similar invariants. By making their type constructors partial, we were able to eliminate all the Ord constraints from type signatures in data-partition, and all but 2 in PSQueue.

The fingertree package provides a type FingerTree v a which (semantically) requires a Measured v a constraint. By making the FingerTree type constructor partial, we were able to reduce the number of Measured constraints in instances and signatures from 85 to 15. We were also able to add a Functor instance for FingerTree v, which would not previously have been possible.

Monad transformers were one of the original motivating examples for partial type constructors. By modifying the transformers library to make the transformers partial, we eliminated 35 of the 39 Monad constraints in instances, and all of the 117 Monad constraints in type signatures.

6 Conclusion

The theory of partial type constructors wants to have its cake and eat it too: despite entailing a foundational reconception of how we think about types and type constructors, it seems

Source	*Classes/instances*		*Signatures*	
	Changed	Total	Changed	Total
compiler/GHC	133	1931	218	16129
compiler/GHC/CmmToAsm	1	72	3	971
compiler/GHC/Driver	21	58	6	658
compiler/GHC/Types	3	264	3	1616
compiler/GHC/Utils	1	120	10	622
compiler/GHC/Unit	2	67	0	329
compiler/GHC/Data	11	77	29	372
compiler/GHC/Hs	71	106	108	452
compiler/GHC/Iface	19	298	7	517
compiler/GHC/HsToCore	0	52	9	851
compiler/GHC/Parser	1	83	0	251
compiler/GHC/Tc	3	208	6	2735
compiler/GHC/Core	0	279	3	2868
compiler/GHC/Cmm	0	94	4	611
compiler/GHC/ByteCode	0	12	4	54
compiler/GHC/Rename	0	21	2	539
compiler/GHC/StgToCmm	0	17	5	505
compiler/GHC/Runtime	0	7	6	233
compiler/GHC/Stg	0	18	12	220
compiler/GHC/Settings	0	0	1	12
libraries	495	5442	412	17337
libraries/transformers	167	444	33	271
libraries/ghc-prim	4	101	0	32
libraries/Cabal	30	1962	49	5698
libraries/deepseq	12	192	0	15
libraries/haskeline	24	56	92	371
libraries/mtl	69	80	3	6
libraries/array	2	65	8	73
libraries/template-haskell	3	124	5	429
libraries/time	1	203	2	541
libraries/base	78	1108	26	2815
libraries/terminfo	4	16	44	93
libraries/containers	28	419	72	2759
libraries/binary	10	154	0	353
libraries/parsec	12	28	70	153
libraries/exceptions	51	71	5	27
libraries/bytestring	0	85	2	899
libraries/libiserv	0	0	1	21

Figure 2. Impact of partial type constructors on GHC.

(via elaboration) to be applicable to existing functional programs without change. Our exploration suggests that we can only eat about 90% of our cake, and currently must leave some of the best looking parts on the plate. We still think this is a not-discouraging result: while a transition to partial type constructors would hardly be seamless, it need not be more onerous than abandoning inferred polymorphism for local binding or the adoption of applicative functors (which required changing all existing `Monad` instances).

6.1 Related Work

We summarize related work in two categories: partiality in types and practical evaluation of language design.

Partiality in types. The Haskell 1.0 report [5] permits contexts in datatype and type synonym definitions, and gives an informal description of their intended semantics that aligns closely with partial type constructors. However, without a satisfactory formal description, the interpretation of datatype contexts was changed and type synonym contexts were dropped from subsequent versions of the report [16]. Hughes [6] proposed encoding many of the use cases of partial type constructors using multiparameter classes; Orchard and Schrijvers [14] described a language extension that made it possible to realize Hughes' encodings with associated types. Unlike with partial type constructors, these approaches require classes to be adapted to the possibility of partial types, and requires total types to announce their totality in each of their class instances. Several authors [15, 18, 20] have used free monads to separate writing monadic code from checking definedness constraints. This approach trades off expressiveness for efficiency—as Sculthorpe et al. point out, using a deep embedding of a `Set` monad will not result in eliminating duplicates in any intermediate results. With language support for partial type constructors, this trade-off is no longer necessary.

Empirical evaluation of language design. There are several previous examples of empirical evaluation of significant language changes in functional programming languages. Wright [22] studied a selection of prominent ML programs to argue for a limitation of polymorphism in bindings. Coutts [3] described using Hackage to perform regression testing on foundational parts of the Haskell ecosystem. Morris [12] used a survey of Hackage to evaluate the practical uses of overlapping instances. Vytiniotis et al. [21] used a study of Hackage to estimate the impact of a restriction of inferred polymorphism.

6.2 Future Work

We identify two key areas of future work: rethinking current higher-order abstractions, and the performance impacts of elaboration.

Rethinking libraries. The `Applicative` class is only one example (albeit the most significant we have discovered so far) of an existing higher-order abstraction that could be tweaked to better interact with partial type constructors. While we have measured the local benefits of partial type constructors in libraries such as `mtl` and `transformers`, we have not evaluated the wider-ranging possibilities of introducing partial type constructors—such as adding types like `Set` to classes like `Functor` and `Traversable`, and the corresponding reduction in code duplication. Building on our

success with GHC itself, we hope to develop a more comprehensive set of libraries suited to, and taking advantage of, partial type constructors.

Elaboration overhead. Another challenge of our approach is that our elaboration step introduces numerous additional constraints in the types of polymorphic functions. For total type constructors, these additional arguments provide no value, but can still make optimization more difficult. In our current implementation, this causes significant performance loss in some test cases. While eliminating all unnecessary function arguments *in general* is obviously infeasible, we have some hope that the controlled nature of our introduction of definedness constraints will allow us to make significant progress in eliminating empty constraint parameters.

Acknowledgments

We gratefully acknowledge the help and encouragement of the anonymous reviewers in preparing this work. This material is based upon work supported by the National Science Foundation under Grant No. CCF-2044815.

References

[1] Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 148–161.

[2] Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. 2005. Associated type synonyms. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, Tallinn, Estonia, 241–253.

[3] Duncan Coutts. 2009. Regression testing with Hackage. https://well-typed.com/blog/24/. Last accessed June 2, 2022.

[4] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, San Diego, California, USA, 671–683.

[5] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young. 1990. Report on the Programming Language Haskell, A Non-strict Purely Functional Language. https://www.haskell.org/definition/haskell-report-1.0.tar.gz.

[6] John Hughes. 1999. Restricted Data Types in Haskell. In *Proceedings of the 1999 Haskell Workshop*. University of Utrecht, Technical Report UU-CS-1999-28, Paris, France, 83–100.

[7] Mark P. Jones. 1994. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK.

[8] Mark P. Jones, J. Garrett Morris, and Richard A. Eisenberg. 2020. Partial type constructors: or, making ad hoc datatypes less ad hoc. *Proc. ACM Program. Lang.* 4, POPL (2020), 40:1–40:28.

[9] Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. 2019. Higher-order type-level programming in Haskell. *Proc. ACM Program. Lang.* 3, ICFP (2019), 102:1–102:26.

[10] Conor McBride. 1999. *Dependently Typed Programs and their Proofs*. Ph. D. Dissertation. University of Edinburgh.

[11] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13.

[12] J. Garrett Morris. 2010. Experience Report: Using Hackage to Inform Language Design. In *Proceedings of the third ACM symposium on Haskell (Haskell '10)*. ACM, Baltimore, Maryland, USA.

[13] J. Garrett Morris and Richard A. Eisenberg. 2017. Constrained Type Families. *Proc. ACM Program. Lang.* 1, ICFP, Article 42 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110286

[14] Dominic Orchard and Tom Schrijvers. 2010. Haskell Type Constraints Unleashed. In *Proceedings of the 10th International Conference on Functional and Logic Programming* (Sendai, Japan) *(FLOPS'10)*. Springer-Verlag, Berlin, Heidelberg, 56–71. https://doi.org/10.1007/978-3-642-12251-4_6

[15] Anders Persson, Emil Axelsson, and Josef Svenningsson. 2011. Generic Monadic Constructs for Embedded Languages. In *Implementation and Application of Functional Languages - 23rd International Symposium, IFL 2011, Lawrence, KS, USA, October 3-5, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7257)*, Andy Gill and Jurriaan Hage (Eds.). Springer, 85–99.

[16] Simon Peyton Jones. 1991. Contexts in `data` and `type`. http://code.haskell.org/~dons/haskell-1990-2000/msg00072.html.

[17] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming (IFCP '08)*. ACM, Victoria, BC, Canada, 51–62.

[18] Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. 2013. The Constrained-monad Problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. ACM, New York, NY, USA, 287–298. https://doi.org/10.1145/2500365.2500602

[19] Jan Stolarek, Simon L. Peyton Jones, and Richard A. Eisenberg. 2015. Injective type families for Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, Vancouver, BC, Canada, 118–128.

[20] Josef Svenningsson and Bo Joel Svensson. 2013. Simple and compositional reification of monadic embedded languages. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 299–304.

[21] Dimitrios Vytiniotis, Simon L. Peyton Jones, and Tom Schrijvers. 2010. Let should not be generalized. In *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*, Andrew Kennedy and Nick Benton (Eds.). ACM, 39–50.

[22] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.

[23] Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. 2020. Kind inference for datatypes. *Proc. ACM Program. Lang.* 4, POPL (2020), 53:1–53:28.

[24] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon L. Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, Benjamin C. Pierce (Ed.). ACM, Philadelphia, PA, USA, 53–66.