Partial Type Constructors (extended version)

Or, Making ad hoc datatypes less ad hoc

MARK P. JONES, Portland State University, USA
J. GARRETT MORRIS, The University of Kansas, USA
RICHARD A. EISENBERG, Bryn Mawr College, USA
APOORV N. INGLE, The University of Kansas, USA

Functional programming languages assume that type constructors are total. Yet functional programmers know better: counterexamples range from container types that make limiting assumptions about their contents (e.g., requiring computable equality or ordering functions) to type families with defining equations only over certain choices of arguments. We present a language design and formal theory of partial type constructors, capturing the domains of type constructors using qualified types. Our design is both simple and expressive: we support partial datatypes as first-class citizens (including as instances of parametric abstractions, such as the Haskell `Functor` and `Monad` classes), and show a simple type elaboration algorithm that avoids placing undue annotation burden on programmers. We show that our type system rejects ill-defined types and can be compiled to a semantic model based on System F. Finally, we have conducted an experimental analysis of a body of Haskell code, using a proof-of-concept implementation of our system; while there are cases where our system requires additional annotations, these cases are rarely encountered in practical Haskell code.

ACM Reference Format:
Mark P. Jones, J. Garrett Morris, Richard A. Eisenberg, and Apoorv N. Ingle. 2019. Partial Type Constructors (extended version): *Or,* Making ad hoc datatypes less ad hoc. 1, 1 (November 2019), 43 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

When does a type expression—the text that a programmer might use to describe a type or set of values—actually represent a valid type? In languages with simple type systems (e.g., Java before the introduction of generics [Bracha et al. 1998]), parsing and name resolution are enough. More advanced systems, however, require additional tests. Languages that support parameterized types, for example, must also check the *arity* of type constructors: it is not valid to use the type `list` in ML, for instance, without a choice for its parameter. Arity checking is further extended to *kind checking* in languages like Haskell that allow both types and type *constructors* to be used as parameters. The set of all types—including `Int`, `Bool`, and `Char`, for example—is represented by the kind $\star$, while parameterized type constructors like `List` or `Map` are assigned function kinds ($\star \rightarrow \star$ and

Authors' addresses: Mark P. Jones, Department of Computer Science, Portland State University, 1900 SW 4th Avenue, Portland, OR, USA, mpj@pdx.edu; J. Garrett Morris, Department of Electrical Engineering and Computer Science, The University of Kansas, 2335 Irving Hill Road, Lawrence, KS, USA, garrett@ittc.ku.edu; Richard A. Eisenberg, Bryn Mawr College, Department of Computer Science, Bryn Mawr, PA, USA, rae@cs.brynmawr.edu; Apoorv N. Ingle, Department of Electrical Engineering and Computer Science, The University of Kansas, 2335 Irving Hill Road, Lawrence, KS, USA, ani@ku.edu.

XXXX-XXXX/2019/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$\star \to \star \to \star$, respectively). Using a simple analog of type checking, these kinds can be used to show that, for example, `Map Int (List Char)` is valid while `List Int (Map Char)` is not.

In many languages, any type expression that passes basic checks like these is accepted as denoting a valid type.[1] This works well for type constructors like `List`: we truly can construct and use lists of type `List t` for any type `t`. But there are some situations where it is useful to take a more nuanced approach, limiting the ways in which parameterized types are instantiated. We list a collection of examples in Section 2.1, but focus here on a simple running example: unboxed arrays.

Although many functional programmers do not use them as heavily as linked lists, some applications do require the efficiency of proper packed arrays. Haskell's `array` package[2] gives us this capability: An `Array Int` of size 10 is represented in memory by 10 contiguous cells, each storing an `Int`.[3] This provides efficient random access, but does not guarantee locality: each array cell holds a pointer to a machine integer, and these may be stored at almost arbitrary locations.

For applications where locality is important, or where an additional level of indirection is otherwise undesirable, the `array` package also supports *unboxed* arrays in the type `UArray`. Like `Array`, `UArray` is parameterized by the type of its elements, but this type must be one for which the compiler knows an unboxed representation. At the time of writing, this set includes 17 types, including `Int` and `Double`, but not `Int → Bool` or `Integer` (unbounded-size integers). A `UArray Double` of size 10 really stores 10 machine double-precision floating-point numbers in a contiguous space in memory, guaranteeing both efficient access and locality. The current implementation uses a class `IArray` that allows manipulation of `UArray`s built from only the 17 allowed types; all functions that manipulate `UArray`s are class-constrained.

Yet something is dissatisfying here: A `UArray Integer` makes no sense, as `Integer` cannot be represented without indirection. However, nothing prevents us from writing functions that take and return `UArray Integer`s. This goes against the grain of a typed language: we want senseless code to be detected right away and to induce errors.

We call `UArray` a *partial type constructor*: it is not a total function from $\star$ to $\star$, but a partial one. And it is far from unique: Section 2 provides many examples of others. What is striking is the great variety of ways that partiality can arise and the various ad-hoc techniques programmers have invented to work with partial type constructors.

The main innovation in this paper is a new, practical treatment of datatypes that allows us to specify partiality in type constructors uniformly and thus to eagerly reject chimeras like `UArray Integer`. This framework is practical in that it is mostly backward-compatible, requiring extra annotations only rarely. (In some cases, annotations written today become redundant.) We also show that our system allows for the easy definition of instances for such types. Currently, we cannot define, say, a `Functor` instance for `UArray`, because nothing can prevent the user from using `fmap` to create a `UArray Integer` from a `UArray Int`.

While our solution is best suited to languages with qualified types, the problem we identify cuts across languages. Partial types have been lurking behind the scenes since the birth of parameterized datatypes—`'a bst` is only useful if you have an ordering functions for `'a`s—and the problems of abstracting over them since at least the introduction of constructor classes [Jones 1993b]. Approaches to partial types appear in both functional and object-oriented languages (see Section 7).

We offer the following contributions:

---

[1] An important exception is for languages that support bounded polymorphism, which is an alternative approach to our work, suitable for languages that support subtyping. See the end of Section 7.

[2] http://hackage.haskell.org/package/array

[3] The actual implementation of `Array` also parameterizes `Array` with an index type, while the `IArray` class is parameterized over both the array type and the element type. We elide these details.

- A practical design for a type system that uniformly supports partial type constructors (Section 3). Our design forbids types like `UArray Integer` and allows instances like `Functor UArray`. We give datatype contexts, a much-maligned feature of Haskell, a fresh semantics that echoes the described intent in the original Haskell 1.0 Report [Hudak and Wadler 1990] (see Section 7).
- A formalization of our type system (Section 4), showing that it rejects terms with disallowed types and supports a simple elaboration algorithm to introduce many of the required constraints.
- An internal language of evidence, suitable as a compilation target for our surface language (Section 5). The internal language, based on System F, allows for explicit predicate evidence in types and for predicates in kinds. It has a standard semantics; type constructors are total. We prove that compilation of a program accepted in the surface language produces one accepted in the internal language.
- An experimental analysis of a body of Haskell code based on a proof-of-concept implementation of our design (Section 6). We demonstrate that our approach introduces minimal annotation overhead in practical functional programs.

2 MOTIVATING SCENARIOS

Typed programming languages have survived for decades without partial type constructors. This section presents examples of notionally partial type constructors used today and argues that the status quo is lacking.

2.1 Partial Type Constructors in the Wild

`UArray` might seem like a somewhat special case. But there are many other examples of partial type constructors, both in practical Haskell code and in the research literature surrounding functional programming and its extensions.

Collection types. In many languages, we can define a parameterized datatype `BST t` that represents binary search trees storing values of type `t` at each interior node. While it may be possible, in principle, to construct values of this type for any choice of `t`, it is only useful to do so if `t` has an associated ordering that can be used in the implementation of standard search tree operations such as `insert` and `lookup`. There are numerous other examples of collection types, like `BST`, that make sense only in cases where the parameters support some additional operations, such as an equality test, a comparison, or a hash function.

Number types. Haskell's standard libraries include definitions for types `Ratio t` and `Complex t` that are used to represent rational and complex numbers. Although they can technically be used with any parameter type `t`, these types are only intended to be used with types that are instances of the `Integral` and `RealFloat` classes, respectively.

Monad transformers. The monad transformer library, `mtl`[4], introduces a standard set of constructions to build monads out of other monads—if `m` is a monad, then `ExceptT e m` is a similar monad that also provides exceptions of type `e`. Such a construction is only meaningful if `m` is a monad—for example, the type `ExceptT e Ratio` is well-kinded but does not actually capture anything about exceptions because `Ratio` is not an instance of the `Monad` class. As a consequence, the implementation of the monad transformer library is littered with `Monad m` constraints that convey no new information to the programmer, but are necessary to exclude pathological cases.

[4]http://hackage.haskell.org/package/mtl

Representation considerations. Although functional programming encourages the use of high-level abstractions, there are still some settings that require developers to understand (and perhaps make concessions to) details of low-level data representation.

- The `UArray` example falls into this category.
- In distributed computing, we might use typed channels for communication between the components of a system, but we can only use these channels for types whose values can be serialized/marshaled in some appropriate manner.
- In the code that deals with virtual memory management in an operating system we might use a parameterized type to describe the data that is stored in page table entries, with the parameter describing how the bits in an unmapped entry will be used. But this only makes sense for types whose values fit within the limited number of bits that the hardware provides.

Type functions. The type families extension of Haskell [Chakravarty et al. 2005; Schrijvers et al. 2008], as implemented in GHC [GHC Team 2017, Section 10.9], provides a widely used mechanism for type-level programming. In particular, this allows the definition of *open type functions* using a collection of *instance* declarations, potentially spread over multiple modules, each of which describes the result of applying the function to a specific pattern of type arguments. A standard application is to define the function `Elem` that returns the type of elements stored in a given collection type. For example, a programmer might write **`type instance`** `Elem (List a) = a` to identify the element type `a` of a list type `List a`. In general, however, the result of a given type function will only be defined for certain combinations of arguments: Not every type makes sense as a collection type, for example, and so the interpretation of types like `Elem Bool` or `Elem (Int, Bool)` might be left unspecified. This aspect of partial types is explored in prior work [Morris and Eisenberg 2017], but the current paper sets this in a larger framework.

Numeric constraints. Some languages include support for type-level numbers, which can be used as arguments of parameterized types to specify and validate key details such as the size of a vector or array, the depth of a tree, the width of a cryptographic key, or the alignment of a pointer. In practice, however, these types may only be valid in some cases, requiring, for example, that the numbers fall within a given range or set and/or satisfy certain arithmetic constraints, such as being a power of two or a multiple of some constant.

Informative evidence. In a dependently typed setting, we often must return informative evidence from a comparison operation; this evidence can be examined to assert new facts to the type checker. For example, we might compare keys into a heterogeneous map (where `HMap k v` relates keys of type `k i` to values of type `v i`, but where the index `i` varies between entries). The comparison operation cannot return a simple `Ordering`, as the type checker cannot know the difference between `GT` and `EQ` as the program proves the implementation preserves indices. Instead, we must use a datatype like this:

```
data GOrdering a b where
  GLT :: a < b ⇒ GOrdering a b
  GEQ :: GOrdering a a
  GGT :: a > b ⇒ GOrdering a b
```

This would not make sense for any `a` and `b` though: their *kind* must support ordering. Using `GOrdering` at a kind that does not have an ordering constraint would be meaningless, and thus `GOrdering` is a partial type constructor.

Types as sets. It is common to interpret all of the types in a functional language as *domains* (i.e., pointed CPOs), so as to ensure a well-defined semantics for arbitrary recursive definitions. But it is also possible to interpret types as *sets*, or to use combinations of domains and sets within the same environment, as in Isabelle/HOLCF [Huffman 2012] or in the extension of Haskell suggested by Launchbury and Paterson [1996] that uses type classes to distinguish between pointed and unpointed types. When we mix domains and sets like this, it is important to distinguish between the different function spaces that might be used. Continuous functions, for example, correspond to parameterized types that we might write as `a` $\twoheadrightarrow$ `b`, but these only make sense when `b` is a domain type. Similarly, total functions on sets form parameterized types that perhaps are written as `a` $\rightarrowtail$ `b`. But these may only be valid if both `a` and `b` are set types.

Functions of a known arity. Downen et al. [2019] introduce a new function form $\rightsquigarrow$ useful for denoting the final, runtime arity of a function, without any currying. Knowing how many arguments a function takes at runtime is critical for efficient function calls. But we must be careful: a polymorphic higher-order function that takes an argument of type `a` $\rightsquigarrow$ `b` is allowed, but only if `b` *is not a function.* By treating $\rightsquigarrow$ as a partial type constructor, we can easily and compositionally maintain this tricky invariant.

2.2 Impact of Ignoring Partiality

The failure to provide full support for partial type constructors has significant costs that impact the practice of writing code and complicate the underlying language metatheory, in several ways:

Abstraction. Because they are not properly supported by current languages, partial type constructors do not work well in combination with other important features or abstractions. For example, the inability to define certain type constructors, such as `UArray` or `Set`, as instances of standard type classes like `Functor` or `Monad` has been an almost constant source of frustration for Haskell programmers. Evidence for this appears in many forms, from informal requests and queries in online discussions, to encodings and extensions of Haskell to address this specific shortcoming [Hughes 1999; Orchard and Schrijvers 2010; Persson et al. 2011; Sculthorpe et al. 2013; Svenningsson and Svensson 2013].

Error reporting. Skeptics may argue that types like `UArray Integer` are at best a minor annoyance: they pose no immediate threat to type safety. However, developers already rely on types to identify and prevent common forms of programming error—even kind checking itself is unnecessary to assure type safety. Supporting partial type constructors would allow us to report errors earlier, upon, say, spotting `UArray Integer` instead of reporting an error only when some function tries to populate that type.

Technical foundations and limitations. A proper accounting of partiality requires great care. What does it mean, for example, to instantiate a polymorphic type scheme at a type of the form `F t` when the latter only exists for certain choices of `t`? We point to work on *injective type families* [Stolarek et al. 2015] as an example where the designers of a language feature were able to avoid such complications by treating type families as total, but then, to avoid contradictions, were forced to impose syntactic restrictions that prevent it from being used in some practical applications [Morris and Eisenberg 2017, Section 3.3].

3 LANGUAGE DESIGN FOR PARTIAL TYPE CONSTRUCTORS

3.1 Datatype Contexts in Haskell

The syntax for datatype definitions in Haskell includes a feature that, at first glance, seems to have been designed specifically for the purpose of supporting partial type constructors like `UArray`. In particular, Haskell allows definitions of algebraic datatypes to include type class constraints that specify restrictions on how their parameters can be instantiated. The following example illustrates the concrete syntax for this, using an `IArray a` constraint at the start of the definition to suggest that any parameter of the `UArray` type constructor must be an instance of the `IArray` class, and hence must have an unboxed representation:

```
data IArray a ⇒ UArray a = MkU ...
```

However, following the actual definition in the Haskell Report [Marlow 2010, Section 4.2.1], the `IArray a` constraint shown here is interpreted by associating it with *operations* on `UArray` values rather than the `UArray` type itself, and this fails to give the behavior that we want from a partial type constructor. For example, even with the above definition, the type `UArray Integer` is still accepted as valid and can be used in other types, such as `UArray Integer → UArray Integer`, even though `Integer` is not an instance of the `IArray` class.

With the definition in the Haskell Report, the presence of an `IArray a` constraint in the type definition does not itself provide a proof of this constraint in functions that work with unboxed arrays. For example, it is not possible to define a working `map` function of the following type; the type makes the impossible demand for a fully polymorphic implementation that will work with all combinations of `a` and `b`:

```
mapUArray :: (a → b) → UArray a → UArray b
```

Instead, the programmer is forced to insert explicit `IArray` constraints as part of the type:

```
mapUArray :: (IArray a, IArray b) ⇒ (a → b) → UArray a → UArray b
```

Similar difficulties arise in constructing monadic embeddings of domain-specific languages. These embeddings may rely on limiting the range of possible values, such as by requiring that values be serializable, or meet some other domain-specific criteria. However, while deep embeddings can capture such constraints, using GADTs [Cheney and Hinze 2003; Xi et al. 2003], the resulting types do not fit the standard notion of monad (because they require restrictions on the type of `return`). Judging from both recurring postings and complaints and from the research effort surrounding this problem [Orchard and Schrijvers 2010; Persson et al. 2011; Sculthorpe et al. 2013; Svenningsson and Svensson 2013], many Haskell programmers have been frustrated and confused by the inability of the language to support such examples.

3.2 A Constraint for Well-Formed Applications

The partiality we seek to tame arises when one type is applied to another. We thus wish to use a predicate to specify when one type is applicable to another, and we write this as an infix @. Intuitively, `f @ a` holds when its right-hand argument is a valid parameter for the constructor in its left-hand argument. When `f` is a known type constructor (like `List` or `UArray`), we replace `f @ a` with the constraints (if any) in the datatype context of `f`'s declaration. For example, `List @ a` holds for any type `a`, because the `List` type constructor is total (declared without constraints), but the constraint `UArray @ Integer` does not hold because the argument type on the right of the @ symbol is not an instance of the `IArray` class. We need to preserve @ constraints in the types of polymorphic functions where the definedness of type expressions depends on the quantified variables. The `elem` function on lists does not need an @ constraint in its type

```
elem :: Eq a ⇒ a → List a → Bool
```

but we must capture the fact that `UArray` is partial in the types of unboxed array operations:

```
arrayElem :: (UArray @ a, Eq a) ⇒ a → UArray a → Bool
```

While the `UArray @ a` constraint is formally necessary, it is also implied by the structure of the type: occurrences of the type `UArray a` must always be guarded by `UArray @ a` predicates. We can take advantage of this to automatically elaborate such constraints, rather than requiring programmers to write them explicitly; we give our elaboration function in Section 4.2. Interestingly, this process does not remove the need for *all* such explicit constraints: see Section 6.2 for further exploration. Using this elaboration, we would be able to write

```
arrayElem :: Eq a ⇒ a → UArray a → Bool
```

making it fully parallel with the list `elem` operation.

We can see further advantages of fully embracing partial type constructors as part of the type system when we consider higher-order abstractions. One of the biggest advantages of accepting partiality in the type language is that it allows us to accommodate partial type constructors in abstractions that were originally designed with only total type constructors in mind. To see why, recall the mapping function for unboxed arrays

```
mapUArray :: (IArray a, IArray b) ⇒ (a → b) → UArray a → UArray b
```

With our approach, we could rewrite this type to rely on the partiality of `UArray`:

```
mapUArray :: (UArray @ a, UArray @ b) ⇒ (a → b) → UArray a → UArray b
```

These types (and indeed the type that omits the definedness constraints entirely) are all considered equivalent in our system—we neither require programmers to write out definedness constraints, nor penalize them for doing so. We could not use this function to make `UArray` an instance of `Functor` in Haskell today, as the type of `fmap`:

```
fmap :: Functor f ⇒ (a → b) → f a → f b
```

must work on arbitrary `a` and `b`.

However, our system provides a uniform approach for code that abstracts over type constructors to reflect the possibility that those type constructors may be partial. The `Functor` class, for example, would have the following definition:

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

Again, the @ constraints here are required by the structure of the type, and could be omitted by the programmer. With this definition, we can see that `mapUArray` is a candidate for `fmap`, and the following instance would be accepted:

```
instance Functor UArray where
  fmap = mapUArray
```

The `f @ a` and `f @ b` constraints in the type of `fmap` are sufficient to assure that `a` and `b` are unboxed types, and so we can use `mapUArray` to implement `fmap`.

A common weakness in several of the previous solutions to this problem is that they require some modification to the original definition of the `Functor` class, such as adding an extra constraint [Hughes 1999], or an extra class parameter or associated type [Orchard and Schrijvers 2010; Persson et al. 2011; Sculthorpe et al. 2013; Svenningsson and Svensson 2013]. The problem here is that it is always very difficult for any programmer to anticipate fully how the code they write might be extended by later development work. If the original developer does not include appropriate 'hooks' to enable such extensions, then subsequent developers may be forced to modify the additional definitions, and then have to make patches to other parts of the code that had been working properly until the modifications were made. Our approach can also be seen as relying on a modification of the original `Functor` class definition to include the extra constraints seen above. A key difference, however, is that these constraints are included automatically as an inherent part of the structure and that they then function as generic hooks for future extensions, without committing to any specific application or use.

3.3 Consequences of Partial Type Constructors

Although we tend to focus on technical details, a change in the interpretation of type expressions also has some more human implications because it requires programmers to make adjustments in the ways that they think about and write code. As with any new language feature, practicing programmers are unlikely to adopt a new type system if it seems unintuitive, or does not appear to offer benefits over its predecessor. With the type system described in this paper, for example, programmers will need to make subtle distinctions between type expressions like `a → List a` and `a → UArray a`; even though they have essentially the same syntactic structure, the first makes sense for any choice of type `a`, while the second is only valid when `a` has an unboxed representation.

More concretely, a language with partial type constructors changes the way programmers must think about substitution. If we see a type `forall a. a → F a` (for some `F`), it is tempting to think that the existence of this type implies that, say, `Integer → F Integer` is also a type. Yet, in the presence of partial type constructors, this is not true. Our elaboration function would change the original type to `forall a. F @ a ⇒ a → F a`, which is telling: now we see that `Integer → F Integer` should be possible only when `F @ Integer` holds. Programmers, of course, work in the language before elaboration, so they must now be aware that substitution is not as simple as they might naïvely think.[5]

However, we are optimistic that most programmers will adapt to these changes quite easily. One reason is that programmers already rely heavily on documentation and navigation tools to provide quick access to relevant information about the code they are working on; at some level, it is impossible to do useful work involving any type without some means to discover and understand the operations that it supports. Another reason is that several kinds of partial type constructors have found their way into practical use, in the form of language features such as GADTs [Cheney and Hinze 2003; Xi et al. 2003] and type functions [Chakravarty et al. 2005; Schrijvers et al. 2008], so many programmers have already become accustomed to working with them.

4 A THEORY OF PARTIAL TYPE CONSTRUCTORS

Having laid out a high-level, user-facing approach to partial type constructors, we now formalize our work in order to give our design a precise semantics. We begin with Jones's [1994] theory of qualified types to provide an account of predicates in types. We extend his system in two directions. First, we extend the typing of expressions to account for the partiality of type constructors; we use *qualified kinding*, extending the approach of Morris and Eisenberg [2017], accounting for the role of predicates *in* types just as qualified typing accounts for the role of predicates *on* types. Second, we describe the interaction between datatype declarations and well-definedness constraints: when type declarations are themselves well-defined, how definedness axioms are inferred from type declarations, and how they are used in the typing of terms.

4.1 Type System Foundations

The qualified types system from Jones [1994] provides a general framework for describing *predicates* in types, which limit the instantiation of type variables. While the most common application of qualified types is undoubtedly type classes, qualified types have also been used to capture

[5] This subtlety around substitution is only in the surface language before elaboration. Accordingly, it does not imperil any formal results about the language, which are phrased in the elaboration or compilation target languages.

$$
\begin{array}{llcl}
\text{Kinds} & \kappa & ::= & \star \mid \kappa_1 \to \kappa_2 \\
\text{Types} & \tau & ::= & \alpha \mid C \mid \tau_1\,\tau_2 \\
\text{Predicates} & \pi & ::= & L\,\overline{\tau_i} \mid \tau_1 \mathbin{@} \tau_2 \\
\text{Qualified types} & \rho & ::= & \tau \mid \pi \Rightarrow \rho \\
\text{Type schemes} & \sigma & ::= & \rho \mid \forall \alpha{:}\kappa.\sigma \\
\text{Expressions} & E & ::= & x \mid E_1\,E_2 \mid \lambda x.E \\
& & \mid & \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2
\end{array}
\qquad
\begin{array}{llcl}
\text{Predicate constants} & L & ::= & \ldots \\
\text{Type constructors} & C & ::= & (\to) \mid \ldots \\
\text{Type variables} & \alpha & ::= & \ldots \\
\text{Term variables} & x & ::= & \ldots \\
\text{Kinding environments} & \Delta & ::= & \epsilon \mid \Delta, \alpha{:}\kappa \\
\text{Typing environments} & \Gamma & ::= & \epsilon \mid \Gamma, x{:}\sigma \\
\text{Predicate environments} & P, Q & ::= & \epsilon \mid P, \pi
\end{array}
$$

$$\boxed{P \mid \Delta \vdash \sigma : \kappa}$$

$$
(\textsc{kvar})\ \frac{\alpha{:}\kappa \in \Delta}{P \mid \Delta \vdash \alpha : \kappa}
\qquad
(\textsc{kconst})\ \frac{C : \kappa}{P \mid \Delta \vdash C : \kappa}
$$

$$
(\textsc{kapp})\ \frac{\begin{array}{c} P \mid \Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \\ P \mid \Delta \vdash \tau_2 : \kappa_1 \quad P \Vdash \tau_1 \mathbin{@} \tau_2 \end{array}}{P \mid \Delta \vdash \tau_1\,\tau_2 : \kappa_2}
$$

$$
(\textsc{k}\text{-}\Rightarrow)\ \frac{P \mid \Delta \vdash \pi\ \mathsf{pred} \quad P, \pi \mid \Delta \vdash \rho : \star}{P \mid \Delta \vdash \pi \Rightarrow \rho : \star}
$$

$$
(\textsc{k}\text{-}\forall)\ \frac{P \mid \Delta, \alpha{:}\kappa \vdash \sigma : \star}{P \mid \Delta \vdash \forall \alpha{:}\kappa.\sigma : \star}
$$

$$\boxed{P \mid \Delta \vdash \pi\ \mathsf{pred}}$$

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

$$
\frac{L : \overline{\kappa_i} \to \mathsf{pred} \quad P \mid \Delta \vdash \tau_i : \kappa_i}{P \mid \Delta \vdash L\,\overline{\tau_i}\ \mathsf{pred}}
$$

$$\boxed{\Delta \vdash P}$$

$$
\frac{}{\Delta \vdash \epsilon}
\qquad
\frac{\Delta \vdash P \quad P \mid \Delta \vdash \pi\ \mathsf{pred}}{\Delta \vdash P, \pi}
$$

$$\boxed{P \mid \Delta \vdash \Gamma}$$

$$
\frac{}{P \mid \Delta \vdash \epsilon}
\qquad
\frac{P \mid \Delta \vdash \Gamma \quad P \mid \Delta \vdash \sigma : \star}{P \mid \Delta \vdash \Gamma, x{:}\sigma}
$$

$$\boxed{P \mid \Delta\,;\Gamma \vdash E : \sigma}$$

$$
(\textsc{var})\ \frac{(x{:}\sigma) \in \Gamma}{P \mid \Delta\,;\Gamma \vdash x : \sigma}
\qquad
(\textsc{let})\ \frac{P \mid \Delta\,;\Gamma \vdash E_1 : \sigma \quad P \mid \Delta\,;\Gamma, x{:}\sigma \vdash E_2 : \tau}{P \mid \Delta\,;\Gamma \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau}
$$

$$
(\to\mathrm{I})\ \frac{P \mid \Delta\,;\Gamma, x{:}\tau_1 \vdash E : \tau_2 \quad P \mid \Delta \vdash \tau_1 \to \tau_2 : \star}{P \mid \Delta\,;\Gamma \vdash \lambda x.E : \tau_1 \to \tau_2}
\qquad
(\Rightarrow\mathrm{I})\ \frac{P, \pi \mid \Delta\,;\Gamma \vdash E : \rho \quad P \mid \Delta \vdash \pi\ \mathsf{pred}}{P \mid \Delta\,;\Gamma \vdash E : \pi \Rightarrow \rho}
$$

$$
(\to\mathrm{E})\ \frac{P \mid \Delta\,;\Gamma \vdash E_1 : \tau_1 \to \tau_2 \quad P \mid \Delta\,;\Gamma \vdash E_2 : \tau_1}{P \mid \Delta\,;\Gamma \vdash E_1\,E_2 : \tau_2}
\qquad
(\Rightarrow\mathrm{E})\ \frac{P \mid \Delta\,;\Gamma \vdash E : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid \Delta\,;\Gamma \vdash E : \rho}
$$

$$
(\forall\mathrm{E})\ \frac{P \mid \Delta\,;\Gamma \vdash E : \forall \alpha{:}\kappa.\sigma \quad P \mid \Delta \vdash \tau : \kappa}{P \mid \Delta\,;\Gamma \vdash E : [\tau/\alpha]\sigma}
\qquad
(\forall\mathrm{I})\ \frac{P \mid \Delta, \alpha{:}\kappa\,;\Gamma \vdash E : \sigma}{P \mid \Delta\,;\Gamma \vdash E : \forall \alpha{:}\kappa.\sigma}
$$

Fig. 1. Syntax, kinding, and typing for partial type constructors

applications from subtyping [Jones 1994] to various record systems [Gaster and Jones 1996; Morris and McKinna 2019]. These applications, however, have considered the use of predicates only to constrain the polymorphism of terms, not to constrain the construction of types. Our approach will need to make more fundamental extensions to the base system of qualified types.

The syntax of types and terms is given at the top of Figure 1; it is standard for qualified type systems. We add applications of (@) to the set of predicates, which otherwise contains applied predicate symbols L. The set of type constructors C contains at least the function type constructor $(\rightarrow)$. We will assume the Barendregt convention in the construction of terms and types, so that all variables appearing in environments Δ, Γ are distinct.

The center of Figure 1 gives our *qualified* kinding relation $P \mid \Delta \vdash \sigma : \kappa$. Unlike standard kinding relations, which need only track the kinds of type variables Δ, but like the formation rules of Morris and Eisenberg [2017], we also track a predicate context P. The predicate context comes into play in two kinding rules.

- Rule (κ-⇒), for qualified types $\pi \Rightarrow \rho$, adds π to the context. We require both that π itself is well-formed, and that with π in context ρ is well-formed. As a consequence, the order of predicates in qualified types matters: the judgment

$$\epsilon \mid \epsilon \vdash \forall \alpha{:}\star \rightarrow \star . \alpha \ @ \ \texttt{Int} \Rightarrow \texttt{Eq}\,(\alpha\ \texttt{Int}) \Rightarrow \alpha\ \texttt{Int} : \star$$

  is derivable, whereas the judgment with interposed predicates

$$\epsilon \mid \epsilon \vdash \forall \alpha{:}\star \rightarrow \star . \texttt{Eq}\,(\alpha\ \texttt{Int}) \Rightarrow \alpha \ @ \ \texttt{Int} \Rightarrow \alpha\ \texttt{Int} : \star$$

  is not. While this is somewhat unusual for qualified types system, it is perfectly consistent both with the existing theory and with several of its applications [Jones 1993a].
- Rule (KAPP), for type applications $\tau_1\ \tau_2$, uses the predicate context in showing that type constructor τ_1 is defined at τ_2. To do so we rely on the entailment relation $\cdot \Vdash \cdot$, which we will describe later in this section.

The other rules are all standard.

Figure 1 includes judgments to check formation of predicates ($P \mid \Delta \vdash \pi$ pred), and to predicate ($\Delta \vdash P$) and typing ($P \mid \Delta \vdash \Gamma$) environments. As for type constructors, we assume an assignment of predicate constants L to kinds of the form $\overline{\kappa} \rightarrow$ pred; for example, we would expect that $\texttt{Ord} : \star \rightarrow \text{pred}$ or $\texttt{MonadState} : \star \rightarrow (\star \rightarrow \star) \rightarrow \text{pred}$. As predicates are checked via a judgment separate from that of types, we do not incorporate predicates into the partiality mechanism. Doing so would not pose significant technical difficulty. However, as predicates may already be unsatisfiable, adding partiality to predicate construction seems to add little expressiveness.

The typing relation is given at the bottom of Figure 1. There are two significant differences from existing systems. In (∀E), as usual, we confirm that the instantiating type is well-kinded; given our extension of kinding, however, this also ensures that type applications in the instantiating type are well-defined. In (→I), we confirm that the resulting function type is well-kinded, and so also that type applications in the domain and codomain are defined. The remaining rules are standard for qualified types.

The key formal guarantee provided by our type system is that the typing of terms respects partial type constructors. As the latter is built into the qualified kinding relation, we have the following:

THEOREM 1 (REGULARITY). *If $\Delta \vdash P$, $P \mid \Delta \vdash \Gamma$, and $P \mid \Delta\,;\Gamma \vdash E : \sigma$, then $P \mid \Delta \vdash \sigma : \star$.*

The proof is by induction on the derivation of $P \mid \Delta\,;\Gamma \vdash E : \sigma$; details are given in the appendix, along with proofs of other theorems we present in the text.

4.2 Elaborating Types

Our type system may seem to require that polymorphic functions be annotated with an unwieldy and unintuitive set of constraints. For example, for `fmap`'s type to be well-kinded it must mention several definedness predicates:

$$\forall f{:}\star \rightarrow \star . \forall a{:}\star . \forall b{:}\star . (\texttt{Functor}\,f, f \ @ \ a, f \ @ \ b) \Rightarrow (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

$$\boxed{\Delta \vdash_{\mathsf{u}} \sigma : \kappa} \qquad \boxed{\Delta \vdash_{\mathsf{u}} \pi \ \mathsf{pred}}\ \boxed{\Delta \vdash_{\mathsf{u}} P}$$

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash_{\mathsf{u}} \alpha : \kappa} \qquad \frac{C : \kappa}{\Delta \vdash_{\mathsf{u}} C : \kappa} \qquad \frac{\Delta \vdash_{\mathsf{u}} \tau : \kappa' \to \kappa \quad \Delta \vdash_{\mathsf{u}} \tau' : \kappa'}{\Delta \vdash_{\mathsf{u}} \tau\,\tau' : \kappa} \qquad \frac{L : \overline{\kappa_i} \to \mathsf{pred} \quad \Delta \vdash_{\mathsf{u}} \tau_i : \kappa_i}{\Delta \vdash_{\mathsf{u}} L\,\overline{\tau_i}\ \mathsf{pred}}$$

$$\frac{\Delta \vdash_{\mathsf{u}} \pi\ \mathsf{pred} \quad \Delta \vdash_{\mathsf{u}} \rho : \star}{\Delta \vdash_{\mathsf{u}} \pi \Rightarrow \rho : \star} \qquad \frac{\Delta, \alpha{:}\kappa \vdash_{\mathsf{u}} \sigma : \star}{\Delta \vdash_{\mathsf{u}} \forall \alpha{:}\kappa.\sigma : \star} \qquad \frac{}{\Delta \vdash_{\mathsf{u}} \epsilon} \qquad \frac{\Delta \vdash_{\mathsf{u}} P \quad \Delta \vdash_{\mathsf{u}} \pi\ \mathsf{pred}}{\Delta \vdash_{\mathsf{u}} P, \pi}$$

$$\boxed{\rho \hookrightarrow P}$$

$$\frac{}{\alpha \hookrightarrow \epsilon} \qquad \frac{}{C \hookrightarrow \epsilon} \qquad \frac{\tau_1 \hookrightarrow P_1 \quad \tau_2 \hookrightarrow P_2}{\tau_1\,\tau_2 \hookrightarrow P_1, P_2, \tau_1 \mathbin{@} \tau_2} \qquad \frac{\pi \hookrightarrow P_1 \quad \rho \hookrightarrow P_2}{\pi \Rightarrow \rho \hookrightarrow P_1, P_2}$$

$$\boxed{\pi \hookrightarrow P} \qquad \boxed{\sigma \hookrightarrow \sigma'}$$

$$\frac{\tau_i \hookrightarrow P_i}{L\,\overline{\tau_i} \hookrightarrow \overline{P_i}} \qquad \frac{\rho \hookrightarrow P}{\forall \alpha{:}\kappa.\rho \hookrightarrow \forall \alpha{:}\kappa.P \Rightarrow \rho} \qquad \frac{\sigma \hookrightarrow \sigma'}{\forall \alpha{:}\kappa.\sigma \hookrightarrow \forall \alpha{:}\kappa.\sigma'}$$

Fig. 2. Elaborating definedness constraints

These definedness predicates may seem obvious: as the type application $f\ a$ appears in the type, is it also necessary to mention the predicate $f\ @\ a$? We suggested in the previous section that such predicates might be inferred in an implementation of partial type constructors. We will now characterize formally how such inference could be done.

We begin by defining a version of the kinding relation, written $\vdash_{\mathsf{u}}$, which is unaware of definedness constraints. We do so by eliminating the use of the definedness constraint in (KAPP), and, as they no longer play any role, eliminating the predicate contexts P. The resulting kinding rules are shown at the top of Figure 2. This new relation reflects the expectation of current functional languages: all type constructors are assumed to be total, and so the kinding relation need only check the kinds of type constructors. We can relate derivations in the unaware and full kinding relations.

We now define an elaboration relation $\sigma \hookrightarrow \sigma'$ on type schemes, shown at the bottom of Figure 2. The elaboration relation on base types and qualified types $\rho \hookrightarrow P$ collects the definedness predicates implied by the type structure of ρ, which are then added to the existing qualifiers for type schemes. (We write P_1, P_2 to denote the concatenation of predicate sequences P_1 and P_2.)

We can use elaboration to connect the unaware and full kinding relations. Intuitively, if the kinds in a type match, then we can invent the definedness constraints necessary to make the type well-kinded.

Theorem 2. *If* $\Delta \vdash_{\mathsf{u}} \sigma : \kappa$ *and* $\sigma \hookrightarrow \sigma'$ *then* $\epsilon \mid \Delta \vdash \sigma' : \kappa$.

This theorem does not guarantee that the constraints in the elaborated type will be satisfiable. For example, our intuition is that the type `UArray Integer` is undefined. The elaboration of this type,

```
UArray @ Integer ⇒ UArray Integer
```

does not make this type any better defined; it simply makes explicit the unsatisfiable constraint implied by the original type expression. Nor does the elaboration relation mean that programmers will never need to write definedness constraints explicitly. Terms may include type instantiations that are not reflected directly in their types, but whose definedness must still be ensured. However,

$$\begin{array}{lrcl} \text{Data constructors} & K & ::= & \ldots \\ \text{Datatype declarations} & D & ::= & \texttt{data}\, P \Rightarrow C\,\overline{\alpha{:}\kappa} = \overline{K\,\overline{\tau}} \end{array}$$

$$\boxed{\vdash D}$$

$$\frac{\overline{\alpha_i{:}\kappa_i} \vdash \mathit{wft}(C\,\overline{\alpha_i}), P \quad \mathit{wft}(C\,\overline{\alpha_i}), P \mid \overline{\alpha_i{:}\kappa_i} \vdash \tau_{jk} : \star}{\vdash \texttt{data}\, P \Rightarrow C\,\overline{\alpha_i{:}\kappa_i} = \overline{K_j\,\overline{\tau_{jk}}}}$$

$$\boxed{\vdash_{\mathsf{u}} D}$$

$$\frac{\overline{\alpha_i{:}\kappa_i} \vdash_{\mathsf{u}} P \quad \overline{\alpha_i{:}\kappa_i} \vdash_{\mathsf{u}} \tau_{jk} : \star}{\vdash_{\mathsf{u}} \texttt{data}\, P \Rightarrow C\,\overline{\alpha_i{:}\kappa_i} = \overline{K_j\,\overline{\tau_{jk}}}}$$

$$\boxed{D \hookrightarrow P}$$

$$\frac{P' \hookrightarrow P \quad \tau_{jk} \hookrightarrow P_{jk} \quad P'' = \{\pi \mid \pi \in P, \overline{P_{jk}} \wedge \pi \notin \mathit{wft}(C\,\overline{\alpha})\}}{\texttt{data}\, P' \Rightarrow C\,\overline{\alpha{:}\kappa} = \overline{K_j\,\overline{\tau_{jk}}} \hookrightarrow P''}$$

Fig. 3. User-defined datatype validation and elaboration

it does suggest that, in the majority of cases, the requisite definedness conditions can be computed automatically. We evaluate the effectiveness of this approach empirically in Section 6.

4.3 User-Defined Datatypes

We have described how definedness constraints are propagated through the types of expressions, and how they can be elaborated from type expressions. Next, we turn to the initial source of definedness constraints: user-defined datatypes.

Our first problem is validating datatype declarations themselves: a datatype cannot be more defined than the data it stores. We give the syntax of partial datatype declarations in Figure 3: a datatype declaration combines a predicate context P with the usual type arguments and constructor types. We characterize valid datatype declarations with the judgment $\vdash D$. This requires that the context P be well-formed; that the P be sufficient to justify that each constructor argument has kind $\star$; and that any type application in constructor arguments is well-defined. This definition encompasses both regular and non-regular data types. For example, the definition

```
data Eq a ⇒ T a = MkT (T [a])
```

is accepted, so long as there is an instance `Eq a ⇒ Eq [a]` available.

Recursive datatypes pose a small challenge. Consider the classic datatype fixed point declaration:

```
data Fix f = In (f (Fix f))
```

Under what constraints should we consider $\texttt{Fix}\,f$ to be well-defined? The application $f\,(\texttt{Fix}\,f)$ must be defined, but this seems to presuppose that $\texttt{Fix}\,f$ is already well-defined. Our approach is to assume that new datatypes are well-defined in their own definitions.[6] In checking $\vdash D$, we extend the declared predicates P with an additional set of constraints, abbreviated $\mathit{wft}(\tau)$ for "well-formed type", asserting that $C\,\overline{\alpha}$ is well-defined. The abbreviation $\mathit{wft}(\tau)$ is defined by ordered pattern-matching on these equations:

$$\mathit{wft}(\tau\,\tau') = \mathit{wft}(\tau), \mathit{wft}(\tau'), \tau \mathbin{@} \tau' \qquad\qquad \mathit{wft}(\tau) = \epsilon$$

[6]That is, we use a greatest fixed point model of the definedness relation (@). This choice affects only our type language and is independent of Haskell's choice to have lazy runtime semantics.

$$\frac{\pi \in P}{P \Vdash \pi} \qquad \frac{P \Vdash Q_i}{P \Vdash \overline{Q}} \qquad \frac{P_1 \Vdash P_2 \quad P_2 \Vdash P_3}{P_1 \Vdash P_3}$$

$$\frac{P \Rightarrow C\,\alpha_1\,\ldots\,\alpha_n\,\beta_1\,\ldots\,\beta_m \in \mathcal{D} \quad P' = \{\pi \in P \mid fv(\pi) \subseteq \overline{\alpha_i}\}}{\mathcal{D} \triangleright C\,\tau_1\,\ldots\,\tau_{n-1} \mathbin{@} \tau_n \dashv\vdash [\tau_i/\alpha_i]P'}$$

Fig. 4. Entailment

With this definition, the only predicate needed for the definition of `Fix` to be well defined is $f \mathbin{@} \texttt{Fix}\, f$.

We can elaborate datatype declarations to include routine definedness constraints. Datatype declaration elaboration $D \hookrightarrow P$ proceeds by elaborating the declared context P and the types appearing in the constructors. The final elaborated predicate excludes any predicates arising from recursive instances of the datatype being defined. As in elaborating types, the elaborated constraints are sufficient to ensure that datatype declarations are well-formed.

Theorem 3. *If* $\vdash_u \texttt{data}\, P \Rightarrow C\,\overline{\alpha{:}\kappa} = \overline{K\,\overline{\tau}}$ *and* $\texttt{data}\, P \Rightarrow C\,\overline{\alpha{:}\kappa} = \overline{K\,\overline{\tau}} \hookrightarrow P'$, *then* $\vdash \texttt{data}\, P', P \Rightarrow C\,\overline{\alpha{:}\kappa} = \overline{K\,\overline{\tau}}$.

The proof follows from Theorem 2. However, the consequences of this theorem are stronger. As there is no source of definedness constraints in datatype declarations other than those discovered by elaboration, the user must only ever add constraints if they are not implied by the components of the datatype being defined.

4.4 Entailment

The entailment relation captures relationships between predicates, and plays a central role in any qualified type system. In our system, we have seen that entailment plays a central role in both kinding (kapp) and typing ($\Rightarrow$E). We now describe the entailment rules for definedness constraints. As with other applications of qualified types, we do not assume that this is the only entailment rule; others could be included to support type classes, extensible records, and so on.

Our entailment relation for definedness constraints is given in Figure 4. Entailment is defined in the context of a set of datatype declarations $\mathcal{D}$; however, as this context is constant in the course of any derivation, we generally write $P \Vdash Q$ for $\mathcal{D} \triangleright P \Vdash Q$. Intuitively, given the following three definitions:

```
data                    Either a b  = Left a | Right b
data Ord a           ⇒ BST a       = Empty | Fork a (BST a) (BST a)
data (Ord a, Ord b) ⇒ OrdPair a b = ...
```

we would generate the following collection of entailment rules:

$$\begin{array}{rcl} & & P \Vdash \texttt{Either} \mathbin{@} a \\ & & P \Vdash \texttt{Either}\ a \mathbin{@} b \\ P \Vdash \texttt{Ord}\ a & \Longleftrightarrow & P \Vdash \texttt{BST} \mathbin{@} a \\ P \Vdash \texttt{Ord}\ a & \Longleftrightarrow & P \Vdash \texttt{OrdPair} \mathbin{@} a \\ P \Vdash \texttt{Ord}\ a \wedge P \Vdash \texttt{Ord}\ b & \Longleftrightarrow & P \Vdash \texttt{OrdPair}\ a \mathbin{@} b \end{array}$$

for any choices of types a and b. This is captured by the final rule in Figure 4. Suppose that we have a predicate $C\,\tau_1\,\tau_2\,\ldots\,\tau_{n-1} \mathbin{@} \tau_n$, and that the corresponding datatype declaration is of the form

$$\texttt{data}\, P \Rightarrow C\,\alpha_1\,\ldots\,\alpha_n\,\beta_1\,\ldots = \overline{K\,\overline{\tau}}$$

$$
\begin{array}{llcl}
\text{Kinds} & \kappa & ::= & s \mid (\alpha{:}\kappa_1) \to \kappa_2 \mid (\delta{:}\pi) \Rightarrow \kappa \\
\text{Types} & \tau, \pi & ::= & C \mid \alpha \mid \tau_1\,\tau_2 \mid \tau\,\upsilon \\
 & & \mid & \forall \alpha{:}\kappa.\tau \mid (\delta{:}\pi) \Rightarrow \tau \\
\text{Evidence} & \upsilon & ::= & \delta \mid \Diamond \mid \ldots \\
\text{Expressions} & E & ::= & x \mid \lambda x{:}\tau.E \mid E_1\,E_2 \mid \lambda \delta{:}\pi.E \\
 & & \mid & E\,\upsilon \mid \Lambda \alpha{:}\kappa.E \mid E\,\tau
\end{array}
\qquad
\begin{array}{llcl}
\text{Type constants} & C, L & ::= & (\to) \mid \top_\kappa \mid \ldots \\
\text{Type vars} & \alpha, \ell & ::= & \ldots \\
\text{Evidence vars} & \delta & ::= & \ldots \\
\text{Term vars} & x & ::= & \ldots \\
\text{Sorts} & s & ::= & \star \mid \mathsf{o} \\
\text{Kinding env's} & \Delta & ::= & \epsilon \mid \Delta, \alpha{:}\kappa \mid \Delta, \delta{:}\pi \\
\text{Typing env's} & \Gamma & ::= & \epsilon \mid \Gamma, x{:}\tau
\end{array}
$$

$$\boxed{\Delta \vdash_i \kappa\ \mathsf{kind}}$$

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

$$\boxed{\Delta \vdash_i \tau : \kappa}$$

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

$$
\frac{\Delta \vdash_i \kappa\ \mathsf{kind} \quad \Delta, \alpha{:}\kappa \vdash_i \tau : s}{\Delta \vdash_i \forall \alpha{:}\kappa.\tau : s}
\qquad
\frac{\Delta \vdash_i \pi : \mathsf{o} \quad \Delta, \delta{:}\pi \vdash_i \tau : s}{\Delta \vdash_i (\delta{:}\pi) \Rightarrow \tau : s}
$$

$$\boxed{\Delta \vdash_i \upsilon : \pi} \qquad \boxed{\Delta; \Gamma \vdash_i E : \tau} \qquad \boxed{E_1 \longrightarrow E_2}$$

$$
\frac{\delta{:}\pi \in \Delta}{\Delta \vdash_i \delta : \pi}
\qquad
\frac{\Delta \vdash_i \top_\kappa\,\overline{\tau_i} : \mathsf{o}}{\Delta \vdash_i \Diamond : \top_\kappa\,\overline{\tau_i}}
\qquad \cdots \qquad \text{(elided)} \qquad \text{(elided)}
$$

Fig. 5. The internal language

Let P' be those predicates in P that restrict only the α_i. The predicate holds (that is, the type application $C\,\tau_1\ \ldots\ \tau_n$ is defined) exactly when (the substitution instances of) P' hold. This includes the treatment of total parameterized datatypes as a special case with $P = \emptyset$: hypotheses are vacuous, and so the definedness predicate always holds.

The theory of qualified types places several requirements on the entailment relation [Jones 1994].

Lemma 4 (Properties of entailment).

(1) Monotonicity: If $P \Vdash \pi$ then $P, P' \Vdash \pi$

(2) Cut: If $P \Vdash \pi_1$ and $P, \pi_1 \Vdash \pi_2$ then $P \Vdash \pi_2$.

(3) Closure under substitution: If S is some well-kinded substitution, and $P \Vdash \pi$, then $S\,P \Vdash S\,\pi$.

The proofs of these properties for our entailment relation are unsurprising.

5 MAKING PARTIALITY EXPLICIT

We have presented the details of a surface language supporting partial type constructors, including an elaboration process for inserting routine definedness constraints. But we must still be cautious: this system is a departure from our usual understanding of type constructors, a fundamental concept in typed functional programming languages. We wish to be sure it has reasonable runtime behavior and is compilable using standard techniques. This section presents an internal language, inspired by System F, into which our surface language compiles. We prove that this internal language is type-safe (by the usual progress and preservation theorems [Wright and Felleisen 1994]) and that compilation from our surface language preserves typability.[7]

5.1 Internal Language Syntax and Semantics

This internal language is laid out in Figure 5. The key difference between the surface language and this internal language is that the internal language uses explicit *evidence* to prove predicates. This follows from the long-standing dictionary translation of qualified types [Jones 1994]. Evidence terms v prove both standard predicates $L\,\overline{\tau_i}$ and also definedness constraints $\tau_1 \ @\ \tau_2$. Evidence can be abstracted over; δ is the metavariable for evidence variables. The trivial evidence $\Diamond$ proves the trivial predicate $\top_\kappa$ of kind κ, and we allow for the possibility of further evidence forms, echoing the possibility of expanding the entailment relation of Section 4.4.

This internal language also merges the grammars of types and predicates and includes two sorts s: $\star$ and o.[8] As we will see, the internal language needs to abstract over predicates, and thus we promote o to be a kind, alongside $\star$. Kinds also include dependent functions over both types and predicates. Types are System F types, extended with quantification over predicates and evidence application. Terms are standard for an evidence-bearing translation of a qualified type system. The typing rules for this language are unsurprising; note, in particular, that the type application rule is entirely standard—type applications are total in our internal language. Term typing rules and runtime operational semantics are also standard; they appear in our appendix.

The key to understanding the connection between our surface language and this internal language is that we represent surface language partiality by constraints in kinds in the internal language. For example, recall `UArray :: ★ → ★`, but with the partiality condition `UArray @ a` defined as `IArray a`. In the internal language, we get `UArray` $: (\alpha{:}\star) \to$ `IArray` $\alpha \Rightarrow \star$. That is, `UArray` τ has kind $\star$ only when we can supply evidence that `IArray` τ holds. This encoding of partiality via constraints in kinds is why we need dependent functions in our language.

Our internal language is type safe:

Definition 5 (Values)**.** The three abstraction forms of expressions E are considered *values*. Other expression forms are not values.

Theorem 6 (Preservation). *If* $\Delta; \Gamma \vdash_{\mathrm{i}} E : \tau$ *and* $E \longrightarrow E'$, *then* $\Delta; \Gamma \vdash_{\mathrm{i}} E' : \tau$.

Theorem 7 (Progress). *If* $\Delta; \epsilon \vdash_{\mathrm{i}} E : \tau$, *then either* E *is a value or there exists* E' *such that* $E \longrightarrow E'$.

This internal language is *not* a final compilation target. It is meant as an intermediate language, where a complete compiler might perform optimizations with the extra security of being able to check that these optimizations respect types. In this way, our language plays a very similar role to System FC [Sulzmann et al. 2007], used as an intermediate language within GHC. In particular, our design here has no bearing on *type erasure*: further compilation steps may indeed erase types and trivial evidence.

5.2 Examples

We can compile surface-language expressions and types into our internal language, converting partiality constraints as appropriate. This compilation function, presented in full in our appendix, is intricate. It is best explained by example.

The simplest example is

[7] *Elaboration* is distinct from *compilation*. Elaboration adds necessary definedness constraints in the source language; the process takes a source program and returns another source program. Compilation, on the other hand, translates from a source language with partiality and definedness constraints into an internal language with neither.

[8] Following Church [1940] we use the symbol o to classify predicates.

```
length :: UArray a → Int
```

Elaboration of elided definedness constraints converts this to

```
length :: UArray @ a ⇒ UArray a → Int
```

equivalent to

```
length :: IArray a ⇒ UArray a → Int
```

In the internal language, this becomes[9]

```
length : forall (a : ★). (d : IArray a) ⇒ UArray a d → Int
```

Note that we explicitly apply `UArray a` to the definedness evidence `d`.

The interesting aspects of compilation arise when we consider abstracting over type variables of a higher kind. So, we proceed to examine `fmap`:

```
fmap :: Functor f ⇒ (a → b) → f a → f b
```

Elaborating and making quantification explicit, this becomes

```
fmap :: forall (f :: Type → Type) (a :: Type) (b :: Type).
        Functor f ⇒ f @ a ⇒ f @ b ⇒ (a → b) → f a → f b
```

Compiling yields

```
fmap : forall (c : ★ → o) (f : (a:★) → c a ⇒ ★) (a : ★) (b : ★).
       Functor c f ⇒ (d1 : c a) ⇒ (d2 : c b) ⇒ (a → b) → f a d1 → f b d2
```

Here, we see that it is necessary to quantify over a constraint variable `c`, denoting the definedness constraint of applying `f` to a variable. Because `f`'s kind mentions `c`, we also must alter the kind of `Functor` appropriately. To wit, we have

```
Functor : (c : ★ → o) → ((a:★) → c a ⇒ ★) → o
```

We thus apply `Functor` to `c` before we can apply it to `f`.

This translation becomes more intricate as we build more abstraction. Our final example will be

```
lift :: (MonadTrans t, Monad m) ⇒ m a → t m a
```

from the monad transformers [Jones 1995a] library. Elaboration and explicit quantification yield

```
lift :: forall (t :: (★ → ★) → ★ → ★) (m :: ★ → ★) (a :: ★).
        MonadTrans t ⇒ Monad m ⇒ m @ a ⇒ t @ m ⇒ t m @ a ⇒ m a → t m a
```

Compiling yields this monster:

```
lift : forall (ct1 : (cm:★ → o) → ((a:★) → cm a ⇒ ★) → o)
              (ct2 : (cm:★ → o) → ((a:★) → cm a ⇒ ★) → ★ → o)
              (t : (cm:★ → o) → (b1:(a:★) → cm a → ★) → ct1 cm b1 ⇒
                   (b2:★) → ct2 cm b1 b2 ⇒ ★)
              (mt : ★ → o) (m : (a:★) → mt a ⇒ ★) (a : ★).
       MonadTrans ct1 ct2 t ⇒ Monad mt m ⇒
       (d1 : mt a) ⇒ (d2 : ct1 mt m) ⇒ (d3 : ct2 mt m a) ⇒
       m a d1 → t mt m d2 a d3
```

Because there is no way to know what the definedness constraints are on `t` and `m`, we must quantify over them. Call sites will instantiate these appropriately, using the trivial predicate $\top$ if instantiating with a total type constructor.

[9]We will use Haskell-like syntax in these examples, with the exception that we use only one colon to remind the reader that we are in an internal language.

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

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

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

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

Fig. 6. Compiling type applications and entailment

5.3 Compiling Types

The fragment of the compilation algorithm concerning types appears in Figure 6. The rest is included in the appendix; the details of this algorithm are not important for understanding our main theorem (that deterministic compilation to a total language is possible) or partial type constructors more generally.

For each compilation judgment, there is a corresponding judgment in the source language with the same structure. Accordingly, these judgments can be viewed as a function on source typing derivations. Values to the left of $\rightsquigarrow$ (and any decorations on a $\rightsquigarrow$) are considered inputs, while values to the right are considered outputs. This function is syntax-directed and deterministic (as proved by a straightforward induction).

The judgments presented here, along with most others, are parameterized by a *compilation context* μ. This contains auxiliary information needed by compilation. Relevant to entailment compilation are mappings from predicates π to evidence variables δ: a mapping $\pi \mapsto \delta \in \mu$ means that δ is evidence for π.

Type applications. The type compilation judgment has two outputs: the compiled type τ and a list of other types $\overline{\tau'}$. This list of types should be passed to any type function that will then be passed the main output type. The intuition here is that the list of types includes the instantiations for any quantified constraints in the type being compiled. For example, note the extra `c` argument in `Functor c f` in the `fmap` example: this would be the second return value when compiling `f`.

Compiling a type application $\tau_1\, \tau_2$ naturally begins by compiling τ_1 and τ_2 separately, yielding τ_1' (with $\overline{\tau}$) and τ_2' (with $\overline{\tau'}$). Since a precondition of compilation is that the source type is well-formed, we know $P \Vdash \tau_1 \mathbin{@} \tau_2$ must hold. We thus compile the derivation of that judgment into evidence υ.

We must now assemble the outputs. We cannot simply compile the application $\tau_1\, \tau_2$ into $\tau_1'\, \tau_2'$: if τ_2 is higher-kinded (i.e., has a function kind), then its compiled kind will depend on a definedness predicate, like the `f` in the type of `fmap`. The choice of this definedness predicate must be passed to τ_1' before passing τ_2'. In the general case, there may be many such definedness predicates: see how `t` in the type of `lift` depends on both `ct1` and `ct2`. The choices for these predicates are $\overline{\tau'}$, the second return value from compiling τ_2. There is still one more wrinkle in assembling the primary result: since τ_1 might be partial, we need to pass evidence that $\tau_1\, \tau_2$ is well-defined. This evidence is the output of compiling the entailment of $P \Vdash \tau_1 \mathbin{@} \tau_2$, which is called υ in the rule. We thus see that the correct primary result from compiling $\tau_1\, \tau_2$ is $\tau_1'\, \overline{\tau'}\, \tau_2'\, \upsilon$.

Lastly, we must determine what predicates the kind of $\tau_1'\, \overline{\tau'}\, \tau_2'\, \upsilon$ depends on. These types, the $\overline{\tau''}$, are built by a list comprehension. The $\overline{\tau}$ are the predicates free in the kind of τ_1'. The first of these predicates is evidenced by υ; it is no longer free in the kind of $\tau_1'\, \overline{\tau'}\, \tau_2'\, \upsilon$. Each of the rest—the tail of the list $\overline{\tau}$—must be accounted for in $\overline{\tau''}$. Since we have applied τ_1' to τ_2', though, we must do the same for the predicates: we thus apply τ_0 (one element of $\mathsf{tail}(\overline{\tau})$) to τ_2', but we cannot forget to insert the $\overline{\tau'}$ first. We thus get the definition of $\overline{\tau''}$ as stated in the rule.

Entailment. Compiling an entailment is straightforward in our presentation, as we can invoke a solver process to produce evidence. The idea here is that our compiler is equipped with a solver than can produce evidence for all entailed predicates. The two rules can be tried in order: if we have a predicate assumption (witnessed by the variable δ) in our compilation context μ, use that. Otherwise, the solver must be able to produce the evidence. The compilation context is extended with evidence when assuming a constraint in the form $\pi \Rightarrow \rho$ (this corresponds to a λ over evidence in the internal language), which then can be retrieved here.

Other judgments. Compiling type applications is one of the two tricky points in compilation; the other is in compiling kinds. This should be unsurprising, because the compilation of a function kind must be intricate enough to support compiling type applications as we have done above. In the end, we prove that a well-kinded source type compiles into a well-kinded internal language type, but even a full statement of the lemma would take us too far afield from our primary goal of discussing partial type constructors.

5.4 Correctness

We have proved compilation correct, stated here for top-level (closed) expressions:

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

The proof generalizes this considerably, allowing compilation of open terms and non-empty contexts; stating that more general theorem would require introducing more technical judgments.

A consequence of this theorem is that we have grounded the theory behind our surface language: it is merely a decoration over a language with fairly standard static and dynamic semantics. Notably, this language has total type constructors; the special treatment of partiality is compiled away.

6 EVALUATION: HOW WILL IT WORK IN PRACTICE?

We began this project with a healthy skepticism about its ultimate feasibility. We were concerned in particular that, while a constraint system seemed reasonable in theory, it might not be usable in practice if it required large numbers of constraints. In the context of extending an existing language to support partial type constructors, there will also be concerns about backward compatibility and about the possibility that substantial portions of existing code will need to be modified or rewritten to account for the new features.

In an attempt to preempt such problems, we have already described how we can allow constraints to be omitted from a program when they are implied directly by other parts of the code. However, as Hughes [1999] noted in his proposal for restricted datatypes, it is not always possible to derive the full list of required constraints for a given function just by looking at its type. For example, a function that sorts an input list by building and then flattening a binary search tree will require a type `(BST @ a) ⇒ List a → List a`, or, equivalently in our system, `(Ord a) ⇒ List a → List a`. Clearly, there is nothing in the type `List a → List a` to hint at a need for either the `BST @ a` or the `Ord a` constraints suggested here.

The observations described above raise an important question about the practical feasibility of the system that we propose in this paper: How often will programmers be required to use extra annotations, either in new code, or when updating existing code? Ideally, we would hope for a *zero-cost abstraction*, meaning, as Stroustrup [1994] put it: "What you don't use, you don't pay for." In our specific case, this means that we would hope not to incur any annotation overhead in code that does not make use of partial type constructors.

To address this question, we built a simple, proof-of-concept implementation of our design, based on the Hugs interpreter. We used it to process a collection of 169 Haskell source files that were taken from the Hugs distribution. This includes the full Hugs `Prelude` as well as standard libraries from the from the `System`, `Data`, `Text`, `Control`, `Test`, and `Language` packages, and combines code from multiple, independent developers who have contributed to the development of Hugs and its libraries. We focus primarily on library code, because it is highly polymorphic. Because we cannot discharge a constraint `f @ a` until we know `f`, we expect polymorphic code to incur a higher annotation overhead than monomorphic code. For exactly the same reason, we did not seek out more application-level code. In total, our sample comprises more than 38,000 lines of code.

Though it supports the partial `BST a` as a built-in type, our prototype does not include the ability to define user-defined partial type constructors. Given that our goal is around backward compatibility of polymorphic code, adding more partial datatypes would not shed more light on this goal: it is partial type *variables* we are after, not partial type *constants*.

Our primary goal was to determine what changes we would need to make in order for this code to be accepted by our prototype. Naturally, some infelicities in our implementation required that we made small edits to these files to allow compilation; these changes are not indicative of our approach and are simply an artifact of the fact that implementation is only a proof-of-concept. We include details of these changes—and further description of our prototype—in the appendix.

6.1 Implementation Details

As described previously, our prototype implementation was developed as an extension of the Hugs interpreter, which already includes a type checker for an extended language based on qualified types. The key changes that we made to add support for well-formedness constraints, as described in this paper, were as follows.

Definedness predicates. We defined a new, three parameter built-in type class `f @ a = r`, with a functional dependency `f a → r` [Jones 1995b, 2000], corresponding to the definedness constraint `f @ a`. The constraint `f @ a = r` requires that the application `f a` is well-defined, just as the two-parameter version `f @ a`. Additionally, it names the type `f a` as `r`. Using this version of the constraint avoided the need to worry about predicate order, as we can simply use the result `r` instead of the type application `f a`, but does not fundamentally change the meaning of definedness constraints. As for the two-parameter version, we have instances of @ for any parameters of kinds $\kappa_1 \to \kappa_2$, κ_1, and κ_2, respectively, for any kinds κ_1 and κ_2.

Elaborating type signatures. We modified the type checker to rewrite every type signature in the input program to include extra constraints, as necessary, to ensure that the type is well-formed. No constraints are generated for applications of known, total type constructors such as `List`, `Maybe`, and `(→)`, but applications of type variables result in a new constraint.

Entailment. The implementation of type classes in Hugs also relies on a definition of *entailment*, as described in Section 4.4. We extended this mechanism in several ways:

- Any predicate of the form `f @ a = f a`, where `f` is an application of a known, total type constructor, can be discharged immediately with no further work. In practice, this often occurs as the second step of a process where a constraint `f @ a = r` has previously been improved by unifying `r` with `f a`. In general, however, it is important to treat this process as two separate steps, either of which may be used independently of the other.
- The corresponding rule for a partial type constructor like `BST` is to allow a constraint `BST @ a = BST a` to be discharged if the constraint `Ord a` can be established from the

assumed constraints. Because these constraints are equivalent, we also have the reverse entailment, allowing an `Ord a` constraint to be discharged if a constraint of the form `BST @ a = r` can be established from the assumed constraints. (There is no need to check that `r` has been improved to `BST a` here; that is already the only possible option.)

6.2 Annotation Overhead

Our first finding was that almost all of the Haskell source files in our test set—164 files, to be precise—are accepted as is by our prototype without the need for any annotations. This provides good initial evidence that the annotation burden for our system is likely to be low. Annotations were required, however, in the five remaining files. Unsurprisingly, these all have to do with abstractions involving higher-kinded type variables: applicative functors, arrows, and monads. For example, the original version of the `Control.Monad` library included the following definition:

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

A detail that can be seen in the function body, but not in its type, is that `sequence (map f xs)` constructs a value of type `m [(b,c)]` that is then used as the left argument of the ≫= operator. To document this fact, the type signature for `mapAndUnzipM` must be modified to include an additional constraint, as follows:

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

With this edit, the entire `Control.Monad` library—which includes numerous definitions of (much more widely used) general monad operators, such as `sequence` and `mapM`—type checks without any further annotations. Finding and making this edit was also easy: the extra constraint was identified in the type error message that was generated in response to the original definition of `mapAndUnzipM`; after that, it was also easy to understand why an extra constraint was needed. Alternative implementations of `mapAndUnzipM` would introduce different constraints. Had we implemented `mapAndUnzipM` using `foldM` instead, the `m @ [(b,c)]` constraint would be unnecessary—indeed, this fact might suggest to use a `foldM`-based implementation instead of the one above.

We found similar examples in four other library files: `Control.Monad.Reader` (1 example); `Data.Foldable` (3 examples); `Control.Applicative` (8 examples); and `Control.Arrow` (4 examples). As before, we easily identified the needed constraints in each case.

The examples in `Data.Foldable` (for the functions `traverse_`, `for_`, and `sequenceA_`) were notable because they each require a constraint of the form `f @ (() → ())` for some applicative functor, `f`. This is interesting because the type `() → ()` is not generally useful in practical work. As such, the presence of this constraint may provide useful feedback, perhaps leading to a new implementation with less plumbing overhead, or to a review of whether these functions are useful enough in practice to be included in the library.

These experiences were encouraging because they suggest that that the need for annotations will be relatively low, even in code that abstracts over parameterized type constructors, and must therefore allow for partiality.

We did, however, find one additional example of a function in the `Control.Arrow` library that requires additional constraints. The original definition of this function is as follows:

```
leftApp :: ArrowApply a ⇒ a b c → a (Either b d) (Either c d)
leftApp f = arr ((λb → (arr (λ() → b) ⋙ f ⋙ arr Left, ())) |||
                 (λd → (arr (λ() → d) ⋙ arr Right, ()))) ⋙ app
```

Although the body of this function is quite compact, it makes heavy use of arrow combinators, including five uses of `arr` (which constructs an arrow from a pure function), and three uses of the arrow composition operator, `(>>>)`. As we dig deeper in to the details of how it works, we also start to see that it involves quite a few different arrow types. For example, `arr (λ() → b)` creates an arrow of type `a () b`; `arr (λ() → d)` creates an arrow of type `a () d`; and so on, with each of these different arrow types requiring a pair of definedness constraints. When we put all of these together, the resulting list of constraints on `liftApp`'s types is intimidatingly long:

```
ArrowApply a, a () @ b, a () @ c, a () @ d, a c @ Either c d, a d @ Either c d,
a b @ Either c d, a (Either b d) @ a () (Either c d),
a (a () (Either c d)) @ Either c d
```

Considered individually, however, each of the constraints is reasonable: in each case, it is easy to find a subexpression in the definition of `leftApp` that produces an arrow of the corresponding type and, hence to explain the need for each constraint.

Overall, while the `leftApp` example demonstrates that it is *possible* for a function definition to require large numbers of constraints, it is also an outlier, and, we believe, not representative of what can be expected in practical code. In this case, a comment in the `Control.Arrow` library explains that `leftApp` can be used to make an instance of the `ArrowChoice` type class for any arrow that is already included in the `ArrowApply` type class. But `leftApp` is not actually used anywhere in the code: there are only two instances of `ArrowApply` in our code sample, both of which already have more direct implementations of `ArrowChoice`. In short, `leftApp` corresponds to a formal proof that every instance of one class can be made an instance of another—with some additional hypotheses reflected by our @ constraints—but inspection of the code shows that it achieves this in a roundabout way that does not appear to be useful in practice.

Summary. By instrumenting our prototype, we were able to count a total of 1,934 type signatures, across our full collection of 169 test files. Each of these signatures was checked automatically by the implementation and 142 of them required additional constraints to ensure well-formedness (with a total of 345 additional constraints). As described above, there were only 20 type signatures (i.e., approximately 1% of the total) that required an additional, programmer-supplied annotation, and all of these occurred in a small number of library files, all dealing with abstractions over higher-kinded parameters. Moreover, in each of these examples, the need for extra constraints was identified automatically and was easy to understand in the context of the associated function definition. From our perspective, these results provide strong evidence that our proposed type system will not create an undue burden on programmers. Indeed, two of the language features that are most likely to stress our type system are polymorphism and parameterization. While these are still useful in the construction of practical applications, we suspect that they will often not be used as heavily as in library code—which has been the focus of our evaluation—that is specifically written to encourage reuse. As such, we conjecture that an extension of our evaluation to include code from practical applications will likely show an even smaller annotation overhead than we have reported here.

6.3 Modularity

One concern that can easily arise in reading this discussion is that of *modularity*: when a library author designs an interface to a function, implementation details should not affect the type. And when it is time to update for a later release, it should not be necessary to change the type in order to streamline an implementation. Yet as we see here, implementation choices in `mapAndUnzipM` and `leftApp` "leak" into the type.

The appearance of these "extra" constraints is a *feature*, not a bug. If we think about a world where types are partial, it is vitally important that the implementation of a function does not use a type in a way that brings it outside of its domain. Imagine we are writing a function parametric in a functor, with a type something like `Functor f ⇒ f Int → f Bool`. To satisfy this type, our function must be applicable to any functor, including functors like `BST` or those representing embedded domain specific languages. If our function builds pairs, or lists, or other arbitrary data structures, this is not an implementation detail: those data structures might be unordered, or not be representable in particular EDSLs. This is much like the fact that Haskell's standard `Set` type requires an `Ord` constraint: even if, conceptually, sets might only require an `Eq` constraint, clients must know that elements are actually stored in order.

How will this modularity issue affect users in practice? It is hard to assess this without a wide release of our design, as the issue arises most sharply in the context of long-term maintenance of a polymorphic library with a distributed set of clients. In this context, the library author may choose to, say, optimize a polymorphic function `fun`. This optimization may, in turn, apply an abstract type constructor `f` to a new type argument internally, and thus the optimization may require a change in `fun`'s exported type. We can further imagine a client of `fun` can now no longer instantiate `f` to their desired concrete type constructor `T`, because the new type argument falls outside of `T`'s domain. This is a disappointing scenario: should the library author refrain from making the optimization? Or should downstream clients expect such disruptions occasionally? The ecosystem of such a language would have to work out these expectations.

A potential aid here is the use of *quantified constraints* [Bottu et al. 2017]. Using a quantified constraint, a type can require a total type constructor or one that accepts arguments satisfying a certain constraint. For example, the functions

```
tot :: forall (f :: ⋆ → ⋆) (b :: ⋆). (forall a. f @ a) ⇒ f b → f b
ord :: forall (f :: ⋆ → ⋆) (b :: ⋆). (forall a. Ord a ⇒ f @ a) ⇒ f b → f b
```

can be instantiated only with a total type constructor or one that requires its argument to be ordered, respectively. We can even make a type synonym **type** `Total f = (forall a. f @ a)` to capture this constraint, but we would hope that such a constraint would be used sparingly.

We should note that the modularity problem as described here *already* exists in the context of languages with qualified types. For example, as hinted above, we can imagine an early, unoptimized implementation of a `Set` type to export functions that require only an `Eq a` constraint. Later, the library author discovers binary search trees and wishes to optimize their `Set`—but doing so would require changing all the exported types to require an `Ord a` constraint instead of an `Eq a` one. The library author has the same conundrum as the author of our `fun`, above. Interestingly, our approach eases this change somewhat: both before and after the optimizations, instantiations `a` need only satisfy `Set @ a`; it's just that the expansion of `Set @ a` changes after the optimization.

7 RELATED WORK

Restricted data types. Hughes [1999] observed that many collection types in Haskell were naturally partial; he focused on sets represented as lists rather than binary search trees, but the issues are the same. He proposed two approaches to this problem. The first required explicitly capturing partiality, reifying constraints as dictionaries in class methods. For example, he proposes a class of collections defined by:

```
class Collection c ctxt where
  empty     :: Sat (ctxt a) ⇒ c a
  singleton :: Sat (ctxt a) ⇒ a → c a
  union     :: Sat (ctxt a) ⇒ c a → c a → c a
  member    :: Sat (ctxt a) ⇒ a → c a → Bool
```

Here, `ctxt` would be instantiated by a variable reifying the constraint on type `c`, capturing an `Eq` dictionary for sets or an `Ord` dictionary for search trees. However, he suggested that these extra constraints would soon become overwhelming, and that the need for parameterizing classes (such as `Collection`) over `ctxt` would limit their applicability. As an alternative, he introduced `wft t` constraints, expressing that type `t` was well-formed. Following his approach, the `Functor` class, for example, would be defined as:

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

Hughes argued that `wft` constraints like those in the definition of `fmap` above should be written explicitly, so as to avoid surprising programmers with unexpected restrictions or behavior. However, Hughes also observed that, in many (but not all) cases, the `wft` constraints that a given example requires may be fully determined by the shape of the type to which they are attached. The type for `fmap` given above is a good example of this: the need for the two `wft` constraints to the left of the ⇒ symbol follows immediately from the use of the type applications `f a` and `f b` on the right.

The choice between requiring `wft` constraints to be stated explicitly, or allowing them to be omitted when they are already implied by context is a language design decision. Twenty years on, Hughes' arguments for avoiding programmer surprise—a vote for requiring explicit constraints—may be tempered by concerns about the burden on programmers for dealing with `wft` constraints and about the impact on backward compatibility.

To the best of our knowledge, the approach that Hughes proposed has not been implemented and experimented with in any practical system. In addition, there are some missing details in the implementation sketch that he provided—having to do, for example, with partial applications of type constructors. Nevertheless, we know of no fundamental reason that Hughes' approach could not also be made workable.

E-logic. Hughes' approach has a surprising antecedent: Scott's [1979] work on undefined terms in intuitionistic logic. Scott was concerned about the meaning of logical propositions such as $\forall a.(1/a) \times a = 1$. While this may seem intuitively correct, and is derivable in many presentations of intuitionistic logic, it is unclear what it means if a is instantiated to 0. It would seem to suggest that the equality $1/0 \times 0 = 1$ should be derivable, but $1/0$ is not defined (and the corresponding derivation is not included in models of intuitionistic logic). Scott's solution is to introduce an existence predicate $E(-)$, and require its use at instantiation of quantifiers. Concretely, in his approach, the above formula is not derivable, but $\forall a.E(1/a) \Rightarrow (1/a \times a = 1)$ is. The instantiation of a with 0 is no longer a problem because the term $1/0$ does not satisfy the existence predicate. A crucial difference between Scott's setting and ours is that he considers arbitrary terms, and so cannot give a more refined characterization of existence. We are working in the more constrained domain of type applications, and so can further refine the conditions under which type expressions denote types.

There has been significant further interest in characterizing partial functions in the type theory and theorem proving communities; for a summary, see Bove et al. [2016]. This work has generally focused on partiality arising from recursive definitions, however, whereas we focus on functions undefined on parts of their domain. Our discussion of compilation (Section 5) demonstrates that, while encoding partial functions in terms of total functions may be intuitively direct, managing complexity in the resulting types and terms is still challenging.

Datatype contexts in Haskell. This particular feature has an interesting history. In the original Haskell 1.0 report [Hudak and Wadler 1990, Section 4.1.3], contexts were allowed in both datatype and type synonym definitions and the intended semantics, explained only informally, was very much in line with what we propose in this paper: a declaration of the form **type** $c \Rightarrow T\ u_1\ \ldots\ u_n = \ldots$, for example, "declares that a type ($T\ t_1\ \ldots\ t_n$) is only valid where $c[t_1/u_1,\ \ldots,\ t_n/u_n]$ holds." The report also includes a concrete example, `type (Num a) ⇒ Point a = (a,a)`, and explains that types like `Point a` are only valid when they appear in the scope of a context asserting `Num a`. This text, however, was removed in the Haskell 1.1 report [Hudak et al. 1991], completely disallowing the use of contexts for type synonyms, and introducing the interpretation for contexts in data definitions that remains in the current report (i.e., the only effect is to add constraints to constructor function types). This change appears to have been made in response to a proposal by Peyton Jones [1991] after an online discussion in which "nobody [was] able to give a satisfactory account of what contexts in data and type declarations actually mean" (the latter presumably referring to the lack of either a formal system or a concrete implementation). In 2010, this feature was deprecated as part of the GHC 7.0.1 release: any programs that use it now require an additional command line flag to compile. The associated documentation [GHC Team 2017, Section 10.4.2], explains that "this is widely considered a misfeature, and is going to be removed from the language." Rather than eliminate it, however, the approach that we describe in this paper would allow us to reinstate the feature and at last, with the benefit of nearly three decades of subsequent experimentation and development, provide a semantics and an implementation for it that matches the vision of the original Haskell committee.

Generalized algebraic data types. Interestingly, our approach does not really interact meaningfully with GADTs [Cheney and Hinze 2003; Schrijvers et al. 2009; Xi et al. 2003]. Suppose we have

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

then `T` is actually a total type constructor: `T Bool` is well-formed but uninhabited. If we have an inhabitant of `T a`, we can then prove `a ~ Int`. On the other hand,

```
data (a ~ Int) ⇒ S a = MkS Int
```

means that `S` is partial: `S Bool` is nonsense. In order to even talk about whether `S a` is inhabited, we must first ascertain that `a` is `Int`. These two notions compose: nothing prevents a partial GADT. The features are thus essentially orthogonal.

Encoding partial type constructors in Haskell. Hughes was far from the last author to propose an encoding of partial type constructors in Haskell. Orchard and Schrijvers [2010] suggest extending Haskell with constraint kinds, giving a built-in realization of Hughes' reification of constraints. They give an encoding of their approach in terms of Kiselyov's [2007] reduction of Haskell to one master type class. Persson et al. [2011]; Sculthorpe et al. [2013]; Svenningsson and Svensson [2013] tackle `Monad` instances for types such as `BST`. Their approach is to represent computations generically, using a free monad or continuation monad, realized using a GADT, and validate the constraints on the underlying type when interpreting the resulting term.

Constrained type families. Our use of qualified kinding is similar to constrained type families [Morris and Eisenberg 2017], which support partiality in type families. In that work, each type family `F` is associated with a unique type class `CF`, identifying its domain; uses of type family $F\ t_i$ are then only allowed in contexts where $CF\ t_i$ is provable. The @ constraints in this work play a parallel role, restricting the use of type constructors to cases where they are defined. However, the implementation strategies differ. Constrained type families simplify their intermediate language, while partial type constructors require an extension of their intermediate language.

There are synergizing effects in a language with both constrained type families and partial type constructors. Most immediately, adding constraints to type families makes explicit an implicit partiality in datatype declarations. Suppose that `F` is a type family; how should the following datatype declaration be interpreted?

```
data T a = MkT a (F a)
```

Clearly, there are only instances of this datatype for parameters `a` for which family `F` is defined. Partial type constructors give the natural realization of this constraint. (Indeed, the same observation was made during discussion of implementing constrained type families in GHC.[10]) The previous work addressed this problem by requiring that the datatype `T` store a dictionary for `CF`; with support for partial data types, that dictionary no longer needs to be stored.

The previous work also did not discuss *data families* [Chakravarty et al. 2005], which should be constrained as type families are. In a language with partial type constructors, these constrained data families would behave much like partial type constructors.

Finally, unconstrained type families would be unable to return applied partial types, as there would be no way to know that the application is valid. However, constrained type families could guarantee that a partial type constructor's argument lies within its domain.

We could also view constrained type families themselves as instances of partial type constructors, in which the constraint `CF` on a type family `F` is required by the corresponding axioms for @. One interesting aspect of this direction is that, unlike type constructors, current formulations of type families do not allow them to be partially applied. (This restriction may soon be lifted [Kiss et al. 2019].) Thus, using @ with type families may require a special case, allowing for the appearance of an unsaturated type family.

ML modules. Partial type constructors may seem to address problems similar to some of those addressed by the ML module system. To work with binary search trees in ML, for example, we might define a signature `Ord` that pairs an element type `t` with an ordering relation, and a signature `BST` that pairs a `tree` type with operations like `insert` and `lookup`. The general construction can then be described by a functor `instBST` that maps any `Ord` structure to a corresponding `BST` structure.

This might seem to accomplish something similar to our goals, defining binary search trees restricted to ordered types. However, it does not actually address the combination of parameterized types and partiality that is our primary focus. For example, if `instBST` is applied to distinct `Ord` structures for the `int` and `real` types, then the resulting `tree` types will be treated as entirely distinct types and not as two different instances of a common parameterized type—even though the underlying construction is the same. A practical consequence is that we cannot add new operations that will work on any binary search tree without going back to modify the `BST` and `instBST` definitions: a decidedly non-modular change to the code. If, instead, the two types were defined as instances of some common parameterized type, then we should be able to add definitions of operations like the following that would work on either kind of binary search tree:

```
val size : 'a tree → int
val toList : 'a tree → 'a list
```

But how then do we document and enforce the requirement that the `tree` type constructor used here should be partial? (In other words, that only certain instantiations of `'a` are valid?) One of the goals of our approach is to be able to detect and report such issues to the programmer as compile-time errors.

The framework of modular type classes that was proposed by Dreyer et al. [2007] would likely be a good setting for exploring the introduction of partial type constructors in a language with ML-style modules. Although that proposal does not specifically provide support for standard,

[10] https://github.com/ghc-proposals/ghc-proposals/pull/177#issuecomment-431507862, and following

user-defined, parameterized datatype definitions, we see no reason why a more fully developed version would not be broadly compatible with the ideas described in this paper.

Subtyping and partial types in object-oriented languages. Bounded polymorphism [Cardelli and Wegner 1985] allows for the instantiation of a type variable only by types that are subtypes of some other type τ. Modern object-oriented programming languages adopt this feature to good effect. Notably, Java, C#, and Scala all support datatypes with bounded type parameters. For example, we can define a Java class

```
public class BST<A extends Comparable<? super A>> { ... }
```

such that an instantiating type of `BST` must be a subtype of the `Comparable` interface—that is, it must have an ordering. This example also demonstrates Java's support for a limited amount of contravariance in setting type parameter bounds. The type `BST<T>` is malformed if `T` is not a subtype of `Comparable`, just like we model in this paper. Though not based on bounded polymorphism, C++'s *concepts* [Dos Reis and Stroustrup 2006] similarly restrict the choice of an instantiating type.

There is a key difference, however, between the systems in Java and C# and what we propose here: our type system allows *quantification* over partial type constructors. By contrast, the languages mentioned here are first-order in types: it is impossible to quantify over a parameterized type. Naturally, it is in dealing with higher-kinded type variables (such as when dealing with functors, arrows, and monad transformers) that our system's power becomes clear.

Scala's implementation of bounded polymorphism *does* allow quantification over partial type constructors [Moors et al. 2008] and is a suitable alternative to what we propose here. Naturally, Scala's approach fits its object-oriented setting and its reliance on subtyping requires more type annotations. As usual, subtyping and qualified polymorphism achieve similar goals in different ways. Recent work on a formal foundation for Scala [Stucki 2017] is based on System $F^{\omega}_{<:}$ bears resemblance to our evidence-carrying internal language, but a detailed comparison of the two approaches is beyond the scope of this work.

8 CONCLUSION

The most immediate direction for future work is to implement partial type constructors and explore their practical utility. To that end, we are adding support for partial type constructors to an experimental functional language focused on low-level programming; our motivations here concern expressive abstractions for representing low-level and hardware-defined formats.

We started with a seemingly paradoxical question: when is a type not a type? Surprisingly often, it turns out: whether it is an unboxed array of boxed values, a binary search tree of incomparable values, or type family application unmatched by its defining equations. In this paper, we set out to explore the possibility of using a constraint-based type system as a framework for describing and working with partial type constructors. We have developed such a language design, characterized its formal properties and semantics, and experimentally evaluated its consequences for existing functional programs. Our approach rules out ill-defined types (such as `UArray Integer`), allows abstraction over partial type constructors (such as `Functor UArray`), and does so with minimal disruption to programmers.

REFERENCES

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. https://doi.org/10.1145/3122955.3122967

Ana Bove, Alexander Krauss, and Matthieu Sozeau. 2016. Partiality and recursion in interactive theorem provers - an overview. *Mathematical Structures in Computer Science* 26, 1 (2016), 38–88.

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.* ACM, Vancouver, British Columbia, Canada, 183–200. https://doi.org/10.1145/286936.286957

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.* 17, 4 (Dec 1985), 471–523.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '05).* ACM, Long Beach, California, USA, 1–13.

James Cheney and Ralf Hinze. 2003. *First-class Phantom Types.* Technical Report TR1901. Cornell University.

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. Symb. Log.* 5, 2 (1940), 56–68. https://doi.org/10.2307/2266170

Gabriel Dos Reis and Bjarne Stroustrup. 2006. Specifying C++ Concepts. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06).* ACM, New York, NY, USA, 295–308.

Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Making a Faster Curry with Extensional Types. In *Proceedings of the 12th ACM SIGPLAN Haskell Symposium (Haskell '19).* ACM, Berlin, Germany.

Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. 2007. Modular type classes. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 63–70.

Benedict R. Gaster and Mark P. Jones. 1996. *A Polymorphic Type System for Extensible Records and Variants.* Technical Report NOTTCS-TR-96-3. University of Nottingham.

GHC Team. 2017. GHC User's Guide Documentation. http://www.haskell.org/ghc/docs/latest/users_guide.pdf.

Paul Hudak, Simon Peyton Jones, and Philip Wadler (Eds.). 1991. *Report on the Programming Language Haskell, Version 1.1.* Available from http://haskell.org/definition/haskell-report-1.1.tar.gz.

Paul Hudak and Philip Wadler (Eds.). 1990. *Report on the Programming Language Haskell, Version 1.0.* Available from http://haskell.org/definition/haskell-report-1.0.ps.gz.

Brian Charles Huffman. 2012. *HOLCF'11: A Definitional Domain Theory for Verifying Functional Programs.* Ph.D. Dissertation. Portland State University, Portland, OR, USA. Advisor(s) Hook, James G. and Matthews, John.

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.

Mark P. Jones. 1993a. *Coherence for qualified types.* Technical Report YALEU/DCS/RR-989. Yale University.

Mark P. Jones. 1993b. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Conference on Functional Programming and Computer Architecture (FPCA '93).* ACM, Copenhagen, Denmark, 52–61.

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

Mark P. Jones. 1995a. Functional Programming with Overloading and Higher-Order Polymorphism. In *First International Spring School on Advanced Functional Programming Techniques (Lecture Notes in Computer Science)*, Vol. 925. Springer, Berlin Heidelberg, 97–136.

Mark P. Jones. 1995b. Simplifying and improving qualified types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture (FPCA '95).* ACM, La Jolla, California, USA, 160–169.

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00).* Springer-Verlag, Berlin, Germany, 230–244.

Oleg Kiselyov. 2007. Haskell with only one type class. http://okmij.org/ftp/Haskell/Haskell1/Haskell1.txt.

Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. 2019. Higher-order Type-level Programming in Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '19).* ACM, Berlin, Germany.

John Launchbury and Ross Paterson. 1996. Parametricity and Unboxing with Unpointed Types. In *Proceedings of the 6th European Symposium on Programming Languages and Systems (ESOP '96).* Springer-Verlag, London, UK, UK, 204–218. http://dl.acm.org/citation.cfm?id=645391.651452

Simon Marlow (Ed.). 2010. *Haskell 2010 Language Report.* Available online in HTML and pdf formats from https://www.haskell.org/documentation.

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Generics of a Higher Kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08).* ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/1449764.1449798

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

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *PACMPL* 3, POPL (2019), 12:1–12:28. https://dl.acm.org/citation.cfm?id=3290325

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

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)*, Andy Gill and Jurriaan Hage (Eds.), Vol. 7257. Springer, 85–99.

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

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.

Tom Schrijvers, Simon L. Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and decidable type inference for GADTs. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 341–352.

Dana Scott. 1979. Identity and existence in intuitionistic logic. In *Applications of Sheaves: Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra, and Analysis, Durham, July 9–21, 1977*, Michael Fourman, Christopher Mulvey, and Dana Scott (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 660–696.

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 (ICFP '13)*. ACM, New York, NY, USA, 287–298. https://doi.org/10.1145/2500365.2500602

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.

Bjarne Stroustrup. 1994. *The Design and Evolution of C++*. Addison-Wesley, Boston, MA.

Sandro Stucki. 2017. *Higher-Order Subtyping with Type Intervals*. Ph.D. Dissertation. School of Computer and Communication Sciences, École polytechnique fédérale de Lausanne, Lausanne, Switzerland. https://doi.org/10.5075/epfl-thesis-8014 EPFL thesis no. 8014.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, François Pottier and George C. Necula (Eds.). ACM, 53–66.

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.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 224–235.

A FORMAL PROPERTIES OF THE SURFACE LANGUAGE

See Figure 1, Figure 4.

Assumptions:

(1) There exists a relation $C : \kappa$ assigning kinds to type constants C.
(2) We have $(\to) : \star \to \star \to \star$.
(3) There exists a relation $L : \overline{\kappa_i} \to \mathsf{pred}$ assigning lists of argument kinds to class constants L.
(4) We have $\epsilon \Vdash (\to) \mathbin{@} \tau$ and $\epsilon \Vdash (\to)\, \tau_1 \mathbin{@} \tau_2$.

Definition 9 ($dom(\Delta)$). The domain is the set of all the type variables present in the kinding environment.

$$dom(\Delta) = \{\alpha \mid (\alpha{:}\kappa) \in \Delta\}$$

Lemma 10 (Weakening). *If $P \mid \Delta \vdash \sigma : \kappa$, then $P \mid \Delta, \Delta' \vdash \sigma : \kappa$.*

Proof. By induction on the structure of $P \mid \Delta \vdash \sigma : \kappa$. As we require that environments not repeat variables, the bindings in Δ' cannot interfere with the remainder of the derivation. □

Lemma 11 (Strengthening). *If $P \mid \Delta, \Delta' \vdash \sigma : \kappa$ such that $dom(\Delta') \cap (fv(\sigma) \cup fv(P)) = \emptyset$, then $P \mid \Delta \vdash \sigma : \kappa$*

Proof. By induction on the structure of $P \mid \Delta, \Delta' \vdash \sigma : \kappa$. We have six cases, each of them straightforward to prove by inspecting the derivation. □

Lemma 12 (Cut). *If $P, \pi \mid \Delta \vdash \sigma : \kappa$ and $P \Vdash \pi$, then $P \mid \Delta \vdash \sigma : \kappa$*

Proof. Proof by induction on the structure of $P, \pi \mid \Delta \vdash \sigma : \kappa$, using Lemma 4 for ($\kappa$-$\Rightarrow$). □

Lemma 13 (Source kinding is deterministic). *If $P \mid \Delta \vdash \sigma : \kappa$ and $P \mid \Delta \vdash \sigma : \kappa'$, then $\kappa = \kappa'$.*

Proof. Straightforward induction. □

Lemma 14 (Source type substitution). *Assume $\Delta, \alpha{:}\kappa_2, \Delta' \vdash P$ and $P \mid \Delta \vdash \tau_2 : \kappa_2$.*
(1) If $P \mid \Delta, \alpha{:}\kappa_2, \Delta' \vdash \sigma : \kappa$, then $[\tau_2/\alpha]P \mid \Delta, \Delta' \vdash [\tau_2/\alpha]\sigma : \kappa$.
(2) If $P \mid \Delta, \alpha{:}\kappa_2, \Delta' \vdash \pi\ \mathsf{pred}$, then $[\tau_2/\alpha]P \mid \Delta, \Delta' \vdash [\tau_2/\alpha]\pi\ \mathsf{pred}$.

Proof. By induction on the structure of the input typing derivation.
Case $\sigma = \alpha$: Immediate by the definition of substitution and Lemma 11.
Case $\sigma = \alpha', \alpha' \neq \alpha$: σ is unchanged by substitution, and so this, too, is immediate by Lemma 10.
Case $\sigma = \tau_1\, \tau_2$: By the induction hypothesis and the substitution property of entailment (Lemma 4.3.)
Other cases: By the induction hypothesis or the fact that constant kinds are closed. □

Theorem 1 (Regularity). *If $\Delta \vdash P$, $P \mid \Delta \vdash \Gamma$, and $P \mid \Delta\,; \Gamma \vdash E : \sigma$, then $P \mid \Delta \vdash \sigma : \star$.*

Proof. The proof goes by induction on the structure of $P \mid \Delta\,; \Gamma \vdash E : \sigma$
Case $x : \sigma$: We have a derivation of the form

$$(\textsc{var})\,\frac{(x{:}\sigma) \in \Gamma}{P \mid \Delta\,; \Gamma \vdash x : \sigma}$$

Given $(x{:}\sigma) \in \Gamma$ and hypothesis $P \mid \Delta \vdash \Gamma$ immediately follows that $P \mid \Delta \vdash \sigma : \star$
Case $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau$: We have a derivation of the form

$$(\textsc{let})\,\frac{P \mid \Delta\,; \Gamma \vdash E_1 : \sigma \quad P \mid \Delta\,; \Gamma, x{:}\sigma \vdash E_2 : \tau}{P \mid \Delta\,; \Gamma \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau}$$

By induction on the first hypothesis we have $P \mid \Delta \vdash \sigma : \star$; combining this with the assumption $P \mid \Delta \vdash \Gamma$ it follows that $P \mid \Delta \vdash \Gamma, x{:}\sigma$. Now, using induction on the second hypothesis we obtain the required result $P \mid \Delta \vdash \tau : \star$.

Case $E_1\, E_2 : \tau$**:** We have a derivation of the form

$$(\rightarrow\mathrm{E})\,\frac{P \mid \Delta\,;\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \quad P \mid \Delta\,;\Gamma \vdash E_2 : \tau_1}{P \mid \Delta\,;\Gamma \vdash E_1\, E_2 : \tau_2}$$

By induction on the first hypothesis we have $P \mid \Delta \vdash \tau_1 \rightarrow \tau_2 : \star$. Also by the kinding relation we have:

$$\frac{P \mid \Delta \vdash (\rightarrow)\ @\ \tau_1 : \star \rightarrow \star \quad P \mid \Delta \vdash \tau_2 : \star \quad P \Vdash \tau_1\ @\ \tau'}{P \mid \Delta \vdash \tau_1 \rightarrow \tau_2 : \star}$$

This also includes the derivation of $P \mid \Delta \vdash \tau_2 : \star$ hence giving us the required conclusion.

Case $\lambda x.E : \tau_1 \rightarrow \tau_2$**:** We have a derivation of the form

$$(\rightarrow\mathrm{I})\,\frac{P \mid \Delta\,;\Gamma, x{:}\tau_1 \vdash E : \tau_2 \quad P \mid \Delta \vdash \tau_1 \rightarrow \tau_2 : \star}{P \mid \Delta\,;\Gamma \vdash \lambda x.E : \tau_1 \rightarrow \tau_2}$$

The required conclusion $P \mid \Delta \vdash \tau_1 \rightarrow \tau_2 : \star$ appears in the hypothesis of the derivation, thus giving us the needed result.

Case $E{:}\rho$**:** We have a derivation of the form

$$(\Rightarrow\mathrm{E})\,\frac{P \mid \Delta\,;\Gamma \vdash E : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid \Delta\,;\Gamma \vdash E : \rho}$$

By induction on the first hypothesis we have $P \mid \Delta \vdash \pi \Rightarrow \rho : \star$ and hence, by kinding relation we have $P, \pi \mid \Delta \vdash \rho : \star$. Now by Lemma 12, and the second hypothesis we have the required conclusion $P \mid \Delta \vdash \rho : \star$.

Case $E : \pi \Rightarrow \rho$**:** We have a derivation of the form

$$(\Rightarrow\mathrm{I})\,\frac{P, \pi \mid \Delta\,;\Gamma \vdash E : \rho \quad P \mid \Delta \vdash \pi\ \mathsf{pred}}{P \mid \Delta\,;\Gamma \vdash E : \pi \Rightarrow \rho}$$

By induction on the first hypothesis we have $P, \pi \mid \Delta \vdash \rho : \star$. By the second hypothesis we have $P \mid \Delta \vdash \pi$ pred. Now by the kinding relation we can build a derivation of the required conclusion

$$\frac{P \mid \Delta \vdash \pi\ \mathsf{pred} \quad P, \pi \mid \Delta \vdash \rho : \star}{P \mid \Delta \vdash \pi \Rightarrow \rho : \star}$$

Case $E : [\tau/\alpha]\sigma$**:** We have a derivation of the form

$$(\forall\mathrm{E})\,\frac{P \mid \Delta\,;\Gamma \vdash E : \forall\alpha{:}\kappa.\sigma \quad P \mid \Delta \vdash \tau : \kappa}{P \mid \Delta\,;\Gamma \vdash E : [\tau/\alpha]\sigma}$$

By induction on the first hypothesis we have $P \mid \Delta \vdash \forall\alpha{:}\kappa.\sigma : \star$. Now due to the substitution lemma (Lemma 14) and the second hypothesis we can build the required conclusion.

Case $E : \forall\alpha{:}\kappa.\sigma$**:** We have a derivation of the form

$$(\forall\mathrm{I})\,\frac{P \mid \Delta, \alpha{:}\kappa\,;\Gamma \vdash E : \sigma}{P \mid \Delta\,;\Gamma \vdash E : \forall\alpha{:}\kappa.\sigma}$$

By induction on the hypothesis we know that $P \mid \Delta \vdash \sigma : \star$. By virtue of $\alpha \notin TV(P)$ and the kinding relation, the required conclusion can be derived. □

Lemma 15 (Elaborating Mono Types, Predicates and Qualified Types).

(1) If $\Delta \vdash_{\mathsf{u}} \tau : \kappa$ and $\tau \hookrightarrow P$ then $P \mid \Delta \vdash \tau : \kappa$.
(2) If $\Delta \vdash_{\mathsf{u}} \pi : \kappa$ and $\pi \hookrightarrow P$ then $P \mid \Delta \vdash \pi : \kappa$.
(3) If $\Delta \vdash_{\mathsf{u}} \pi \Rightarrow \rho : \kappa$ and $\pi \hookrightarrow P_1$ and $\rho \hookrightarrow P_2$ then $P_1, P_2 \mid \Delta \vdash \pi \Rightarrow \rho : \kappa$.

Proof.

(1) By induction on the structure of τ we get three cases:
Case α: We have a derivation

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash_{\mathsf{u}} \alpha : \kappa}$$

Using the hypothesis we can build the required conclusion $\epsilon \mid \Delta \vdash \alpha : \kappa$.
Case C: This case is similar to previous case.
Case $\tau_1\,\tau_2$: We have the following derivation

$$\frac{\Delta \vdash_{\mathsf{u}} \tau_1 : \kappa' \rightarrow \kappa \quad \Delta \vdash_{\mathsf{u}} \tau_2 : \kappa'}{\Delta \vdash_{\mathsf{u}} \tau\,{}_1\tau_2 : \kappa}$$

We also have the following elaboration hypothesis $\tau_1 \hookrightarrow P_1$, $\tau_2 \hookrightarrow P_2$ from $\tau_1\,\tau_2 \hookrightarrow P_1, P_2, \tau_1 \mathbin{@} \tau_2$.
By induction on the first hypothesis we get $P_1 \mid \Delta \vdash \tau_1 : \kappa' \rightarrow \kappa$ and induction on second hypothesis we have $P_2 \mid \Delta \vdash \tau_2 : \kappa'$ Thus we can build a derivation for the required conclusion $P_1, P_2, \tau_1 \mathbin{@} \tau_2 \mid \Delta \vdash \tau_1\,\tau_2 : \kappa$

(2) By examining the structure of π we get:
Case L: We have a derivation

$$\frac{L : \overline{\kappa_i} \quad \Delta \vdash_{\mathsf{u}} \tau_i : \kappa_i}{\Delta \vdash_{\mathsf{u}} L\,\overline{\tau_i}\ \mathsf{pred}}$$

By elaboration we have the hypothesis $\tau_i \hookrightarrow P_i$. By applying Lemma 15.1 to the ith hypothesis we get $P_i \mid \Delta \vdash \tau_i : \kappa_i$. Now using the first hypothesis we can build a derivation of required conclusion $\overline{P_i} \mid \Delta \vdash L\,\overline{\tau_i}$ pred

(3) By induction on the structure of ρ:
Case $\pi \Rightarrow \rho$: We have the following derivation:

$$\frac{\Delta \vdash_{\mathsf{u}} \pi\ \mathsf{pred} \quad \Delta \vdash_{\mathsf{u}} \rho : \star}{\Delta \vdash_{\mathsf{u}} \pi \Rightarrow \rho : \star}$$

By elaboration we have two hypothesis $\pi \hookrightarrow P_1$ and $\rho \hookrightarrow P_2$. Applying Lemma 15.2 to the first hypothesis, gives $P_1 \mid \Delta \vdash \pi$ pred and by induction to the second hypothesis we get $P_2 \mid \Delta \vdash \rho : \star$, so we can build the derivation of the required conclusion $P_1, P_2 \mid \Delta \vdash \pi \Rightarrow \rho : \star$
Case τ: by Lemma 15.1. □

Lemma 16. *If $\Delta \vdash_{\mathsf{u}} \rho : \star$ and $\rho \hookrightarrow P$ then $\Delta \vdash P$.*

Proof. By induction. □

Theorem 2 (Elaborating Type Schemes). *If $\Delta \vdash_{\mathsf{u}} \sigma : \kappa$ and $\sigma \hookrightarrow \sigma'$ then $\epsilon \mid \Delta \vdash \sigma' : \kappa$.*

Proof. By structural induction on σ we get two cases:

Case $\forall\alpha{:}\kappa.\rho$: By elaboration assumption $\rho \hookrightarrow P$ we get $\forall\alpha{:}\kappa.\rho \hookrightarrow \forall\alpha{:}\kappa.P \Rightarrow \rho$. By Lemma 15, we have $P \mid \Delta, \alpha{:}\kappa \vdash \rho : \star$ and so by a simple induction on P we have $\epsilon \mid \Delta, \alpha{:}\kappa \vdash P \Rightarrow \rho : \star$. Finally, we can construct $\epsilon \mid \Delta \vdash \forall\alpha{:}\kappa.P \Rightarrow \rho : \star$, as we wanted.

Case $\forall\alpha{:}\kappa.\sigma$: By elaboration assumption $\sigma \hookrightarrow \sigma'$ $\forall\alpha{:}\kappa.\sigma \hookrightarrow \forall\alpha{:}\kappa.\sigma'$. The rest follows by induction. □

Theorem 3 (Elaborating User-defined Datatypes).
If $\epsilon \vdash_{\mathsf{u}} \mathtt{data}\, P \Rightarrow C\,\overline{\alpha{:}\kappa} = \overline{K\,\overline{\tau}}$ *and* $\mathtt{data}\, P \Rightarrow C\,\overline{\alpha{:}\kappa} = \overline{K\,\overline{\tau}} \hookrightarrow P'$,
then $\epsilon \vdash \mathtt{data}\, P', P \Rightarrow C\,\overline{\alpha{:}\kappa} = \overline{K\,\overline{\tau}}$.

Proof. We have the following derivation

$$\Pi_1 = \frac{\overline{\alpha_i{:}\kappa_i} \vdash_{\mathsf{u}} P \quad \overline{\alpha_i{:}\kappa_i} \vdash_{\mathsf{u}} \tau_{jk} : \star}{\epsilon \vdash_{\mathsf{u}} \mathtt{data}\, P \Rightarrow C\,\overline{\alpha_i{:}\kappa_i} = \overline{K_j\,\overline{\tau_{jk}}}}$$

We also have the following elaboration derivation

$$\Pi_2 = \frac{P \hookrightarrow P'' \quad \tau_{jk} \hookrightarrow P_{jk} \quad P' = \{\pi \mid \pi \in P'', \overline{P_{jk}} \wedge \pi \notin \mathit{wft}(C\,\overline{\alpha})\}}{\mathtt{data}\, P \Rightarrow C\,\overline{\alpha{:}\kappa} = \overline{K_j\,\overline{\tau_{jk}}} \hookrightarrow P'}$$

We have $\tau_{jk} \hookrightarrow P_{jk}$ and $P_{jk} \subseteq (P, P', \mathit{wft}(C\,\overline{\alpha}))$
Now using them and applying Lemma 15 on the second hypothesis of Π_1 we get $\mathit{wft}(C\,\overline{\alpha}), P, P' \mid \overline{\alpha_i{:}\kappa_i} \vdash \tau_{jk} : \star$.
From Lemma 16 we have that $\overline{\alpha_i{:}\kappa_i} \vdash \mathit{wft}(C\,\overline{\alpha_i}), P, P'$
We can now build the required conclusion. □

Lemma 4 (Properties of entailment).

(1) Monotonicity: If $P \Vdash \pi$ *then* $P, P' \Vdash \pi$.
(2) Cut: If $P \Vdash \pi_1$ *and* $P, \pi_1 \Vdash \pi_2$ *then* $P \Vdash \pi_2$.
(3) Closure under substitution: If S *is some well-kinded substitution, and* $P \Vdash \pi$, *then* $S\,P \Vdash S\,\pi$.

Proof.

(1) By induction.
(2) By monotonicity and transitivity.
(3) By induction. □

B INTERNAL LANGUAGE

C, L	$::=$	$(\rightarrow) \mid \top_\kappa \mid \ldots$	type constants
x	$::=$	$\ldots$	term-level variables
α, ℓ	$::=$	$\ldots$	type-level variables
δ	$::=$	$\ldots$	evidence variables
s	$::=$	$\star \mid \mathsf{o}$	
κ	$::=$	$s \mid (\alpha{:}\kappa_1) \rightarrow \kappa_2 \mid (\delta{:}\pi) \Rightarrow \kappa$	
τ, π	$::=$	$C \mid \alpha \mid \tau_1\,\tau_2 \mid \tau\,v \mid \forall\alpha{:}\kappa.\tau \mid (\delta{:}\pi) \Rightarrow \tau$	
v	$::=$	$\delta \mid \Diamond \mid \ldots$	evidence terms
E	$::=$	$x \mid \lambda x{:}\tau.E \mid E_1\,E_2 \mid \lambda\delta{:}\pi.E \mid E\,v \mid \Lambda\alpha{:}\kappa.E \mid E\,\tau$	
Δ	$::=$	$\epsilon \mid \Delta, \alpha{:}\kappa \mid \Delta, \delta{:}\pi$	
Γ	$::=$	$\epsilon \mid \Gamma, x{:}\tau$	

Assumptions:

(1) There exists a relation $C : \kappa$ assigning kinds to type constants C. These kinds κ are closed—that is, they have no free variables.
(2) We write $\kappa_1 \longrightarrow \kappa_2$ as an abbreviation for $(\alpha{:}\kappa_1) \longrightarrow \kappa_2$ when α is not free in κ_2.
(3) We have $(\longrightarrow) : (\alpha_1{:}\star) \longrightarrow \top_{\star \to \mathsf{o}}\, \alpha_1 \Rightarrow (\alpha_2{:}\star) \longrightarrow \top_{\star \to \star \to \mathsf{o}}\, \alpha_1\, \alpha_2 \Rightarrow \star$ and $\top_\kappa : \kappa$. We write $\tau_1 \longrightarrow \tau_2$ for $(\longrightarrow)\, \tau_1 \diamondsuit\, \tau_2 \diamondsuit$.
(4) The typing judgment for evidence $\Delta \vdash_i v : \pi$ (defined below) contains rules for the unspecified evidence forms. These rules have the substitution property. That is, for any judgment $\mathcal{J}$ in the rules' premises, if we assume that $\Delta, \alpha{:}\kappa_2, \Delta' \vdash \mathcal{J}$ and $\Delta \vdash_i \tau_2 : \kappa_2$ implies $\Delta, [\tau_2/\alpha]\Delta' \vdash [\tau_2/\alpha]\mathcal{J}$, then the conclusion $\Delta, [\tau_2/\alpha]\Delta' \vdash_i [\tau_2/\alpha]v : [\tau_2/\alpha]\pi$ also holds.

$$\boxed{\Delta \vdash_i \kappa \ \mathsf{kind}}$$

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

$$\boxed{\Delta \vdash_i \tau : \kappa}$$

$$\frac{C : \kappa}{\Delta \vdash_i C : \kappa} \qquad \frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash_i \alpha : \kappa} \qquad \frac{\Delta \vdash_i \tau_1 : (\alpha{:}\kappa_1) \longrightarrow \kappa_2 \quad \Delta \vdash_i \tau_2 : \kappa_1}{\Delta \vdash_i \tau_1\, \tau_2 : [\tau_2/\alpha]\kappa_2} \qquad \frac{\Delta \vdash_i \tau : (\delta{:}\pi) \Rightarrow \kappa \quad \Delta \vdash_i v : \pi}{\Delta \vdash_i \tau\, v : [v/\delta]\kappa}$$

$$\frac{\Delta \vdash_i \kappa \ \mathsf{kind} \quad \Delta, \alpha{:}\kappa \vdash_i \tau : s}{\Delta \vdash_i \forall \alpha{:}\kappa.\tau : s} \qquad \frac{\Delta \vdash_i \pi : \mathsf{o} \quad \Delta, \delta{:}\pi \vdash_i \tau : s}{\Delta \vdash_i (\delta{:}\pi) \Rightarrow \tau : s}$$

$$\boxed{\Delta \vdash_i v : \pi}$$

$$\frac{\delta{:}\pi \in \Delta}{\Delta \vdash_i \delta : \pi} \qquad \frac{\Delta \vdash_i \top_\kappa\, \overline{\tau_i} : \mathsf{o}}{\Delta \vdash_i \diamondsuit : \top_\kappa\, \overline{\tau_i}} \qquad \ldots$$

$$\boxed{\Delta; \Gamma \vdash_i E : \tau}$$

$$\frac{x{:}\tau \in \Gamma}{\Delta; \Gamma \vdash_i x : \tau}(\textsc{Var}) \qquad \frac{\Delta; \Gamma \vdash_i E_1 : \tau_1 \longrightarrow \tau_2 \quad \Delta; \Gamma \vdash_i E_2 : \tau_1}{\Delta; \Gamma \vdash_i E_1\, E_2 : \tau_2}(\longrightarrow\mathrm{E}) \qquad \frac{\Delta \vdash_i \tau_1 : \star \quad \Delta; \Gamma, x{:}\tau_1 \vdash_i E : \tau_2}{\Delta; \Gamma \vdash_i \lambda x{:}\tau_1.E : \tau_1 \longrightarrow \tau_2}(\longrightarrow\mathrm{I})$$

$$\frac{\Delta; \Gamma \vdash_i E : \forall \alpha{:}\kappa.\tau_2 \quad \Delta \vdash_i \tau_1 : \kappa}{\Delta; \Gamma \vdash_i E\, \tau_1 : [\tau_1/\alpha]\tau_2}(\forall\mathrm{E}) \qquad \frac{\Delta \vdash_i \kappa \ \mathsf{kind} \quad \Delta, \alpha{:}\kappa; \Gamma \vdash_i E : \tau}{\Delta; \Gamma \vdash_i \Lambda \alpha{:}\kappa.E : \forall \alpha{:}\kappa.\tau}(\forall\mathrm{I})$$

$$\frac{\Delta; \Gamma \vdash_i E : (\delta{:}\pi) \Rightarrow \tau \quad \Delta \vdash_i v : \pi}{\Delta; \Gamma \vdash_i E\, v : [v/\delta]\tau}(\Rightarrow\mathrm{E}) \qquad \frac{\Delta \vdash_i \pi : \mathsf{o} \quad \Delta, \delta{:}\pi; \Gamma \vdash_i E : \tau}{\Delta; \Gamma \vdash_i \lambda \delta{:}\pi.E : (\delta : \pi) \Rightarrow \tau}(\Rightarrow\mathrm{I})$$

$$\boxed{E_1 \longrightarrow E_2}$$

$$\frac{E_1 \longrightarrow E_1'}{E_1\,E_2 \longrightarrow E_1'\,E_2}(@\approx) \qquad \frac{}{(\lambda x{:}\tau.E_1)\,E_2 \longrightarrow [E_2/x]E_1}(\beta)$$

$$\frac{E \longrightarrow E'}{E\,\tau \longrightarrow E'\,\tau}(\tau@\approx) \qquad \frac{}{(\Lambda\alpha{:}\kappa.E)\,\tau \longrightarrow [\tau/\alpha]E}(\tau\beta)$$

$$\frac{E \longrightarrow E'}{E\,\upsilon \longrightarrow E'\,\upsilon}(\upsilon@\approx) \qquad \frac{}{(\lambda\delta{:}\pi.E)\,\upsilon \longrightarrow [\upsilon/\delta]E}(\upsilon\beta)$$

Lemma 17 (Weakening in types). *Assume $\Delta \subseteq \Delta'$ and, as usual, there are no repeated bindings in Δ'.*

(1) If $\Delta \vdash_i \kappa$ kind, then $\Delta' \vdash_i \kappa$ kind.
(2) If $\Delta \vdash_i \tau : \kappa$, then $\Delta' \vdash_i \tau : \kappa$.
(3) If $\Delta \vdash_i \upsilon : \pi$, then $\Delta' \vdash_i \upsilon : \pi$.

Proof. Straightforward mutual induction. □

Lemma 18 (Strengthening in types). *Assume $dom(\Delta) \cap dom(\Delta') = \emptyset$*

(1) If $\Delta, \Delta' \vdash_i \kappa$ kind, then $\Delta \vdash_i \kappa$ kind.
(2) If $\Delta, \Delta' \vdash_i \tau : \kappa$, then $\Delta \vdash_i \tau : \kappa$.
(3) If $\Delta, \Delta' \vdash_i \upsilon : \pi$, then $\Delta \vdash_i \upsilon : \pi$.

Proof. Straightforward mutual induction. □

Lemma 19 (Weakening). *If $\Delta; \Gamma \vdash_i E : \tau$, then $\Delta; \Gamma, \Gamma' \vdash_i E : \tau$. As usual, we assume $dom(\Gamma) \cap dom(\Gamma') = \emptyset$.*

Proof. Straightforward induction on $\Delta; \Gamma \vdash_i E : \tau$. □

Lemma 20 (Strengthening). *If $\Delta; \Gamma, x : \tau_2, \Gamma' \vdash_i E : \tau_1$ and x is not free in E, then $\Delta; \Gamma, \Gamma' \vdash_i E : \tau_1$.*

Proof. Straightforward induction on $\Delta; \Gamma, x : \tau_2, \Gamma' \vdash_i E : \tau_1$. □

Lemma 21 (Substitution). *If $\Delta; \Gamma, x : \tau_2, \Gamma' \vdash_i E_1 : \tau_1$ and $\Delta; \Gamma \vdash_i E_2 : \tau_2$, then $\Delta; \Gamma, \Gamma' \vdash_i [E_2/x]E_1 : \tau_1$.*

Proof. By induction on $\Delta; \Gamma, x : \tau_2, \Gamma' \vdash_i E_1 : \tau_1$.

Case Var: We know $E = x'$. We have three cases:
Case $x' \in \Gamma$: Then $[E_2/x]E_1 = E_1$ and we are done by strengthening (Lemma 20).
Case $x' = x$: We are done by assumption and weakening (Lemma 19).
Case $x' \in \Gamma'$: Similar to first sub-case.
Other cases: By the induction hypothesis. □

Lemma 22 (Type substitution in types). *If $\Delta \vdash_i \tau_2 : \kappa_2$:*

(1) If $\Delta, \alpha{:}\kappa_2, \Delta' \vdash_i \kappa_1$ kind, then $\Delta, [\tau_2/\alpha]\Delta' \vdash_i [\tau_2/\alpha]\kappa_1$ kind.
(2) If $\Delta, \alpha{:}\kappa_2, \Delta' \vdash_i \tau_1 : \kappa_1$, then $\Delta, [\tau_2/\alpha]\Delta' \vdash_i [\tau_2/\alpha]\tau_1 : [\tau_2/\alpha]\kappa_1$.
(3) If $\Delta, \alpha{:}\kappa_2, \Delta' \vdash_i \upsilon : \pi$, then $\Delta, [\tau_2/\alpha]\Delta' \vdash_i [\tau_2/\alpha]\upsilon : [\tau_2/\alpha]\pi$.

Proof. Proof is by mutual induction and structural analysis on each sub parts.

(1) We have three cases:
Case s: This case is trivial as s can be $\star$ or o. Both cases are idempotent to substitution.
Case $(\alpha{:}\kappa_1) \to \kappa_2$: By induction on hypothesis of the derivation and Lemma 17
Case $\pi \Rightarrow \kappa$: By induction on hypothesis of the derivation and Lemma 17
(2) We have six cases:
Case C: by induction on the hypothesis of derivation and weakening.
Case α': This has three sub-cases:
Case $\alpha' \in \Delta$: The substitution is idempotent and Lemma 18
Case $\alpha' = \alpha$: This the proved by assumption and Lemma 17
Case $\alpha' \in \Delta'$: this is similar to sub-case 1
Case $\tau_1\, \tau_2$: by induction hypothesis of derivation and Lemma 17.
Case $\tau\, \upsilon$: by induction hypothesis of derivation and Lemma 17.
Case $\forall\alpha{:}\kappa.\tau$: by induction hypothesis of derivation and Lemma 17.
Case $(\delta{:}\pi) \Rightarrow \tau$: by induction hypothesis of derivation and Lemma 17.
(3) we have two cases:
Case δ: similar to case α'
Case $\Diamond$: by induction hypothesis and Lemma 17 □

Lemma 23 (Type substitution). *If* $\Delta, \alpha{:}\kappa, \Delta'; \Gamma \vdash_i E : \tau$ *and* $\Delta \vdash_i \tau' : \kappa$, *then* $\Delta, [\tau'/\alpha]\Delta'; [\tau'/\alpha]\Gamma \vdash_i [\tau'/\alpha]E : [\tau'/\alpha]\tau$.

Proof. By induction and appeal to Lemma 22. □

Lemma 24 (Evidence substitution in types). *If* $\Delta \vdash_i \upsilon_2 : \pi_2$*:*
(1) If $\Delta, \delta{:}\pi_2, \Delta' \vdash_i \kappa_1$ *kind, then* $\Delta, [\upsilon_2/\delta]\Delta' \vdash_i [\upsilon_2/\delta]\kappa_1$ *kind.*
(2) If $\Delta, \delta{:}\pi_2, \Delta' \vdash_i \tau_1 : \kappa_1$*, then* $\Delta, [\upsilon_2/\delta]\Delta' \vdash_i [\upsilon_2/\delta]\tau_1 : [\upsilon_2/\delta]\kappa_1$.
(3) If $\Delta, \delta{:}\pi_2, \Delta' \vdash_i \upsilon_1 : \pi_1$*, then* $\Delta, [\upsilon_2/\delta]\Delta' \vdash_i [\upsilon_2/\delta]\upsilon_1 : [\upsilon_2/\delta]\pi_1$.

Proof. The proof is a standard proof of a substitution property, relying on our assumption that the $\Delta \vdash_i \upsilon : \pi$ relation supports substitution. □

Lemma 25 (Evidence substitution). *If* $\Delta, \delta{:}\pi, \Delta'; \Gamma \vdash_i E : \tau$ *and* $\Delta \vdash_i \upsilon : \pi$, *then* $\Delta, [\upsilon/\delta]\Delta'; [\upsilon/\delta]\Gamma \vdash_i [\upsilon/\delta]E : [\upsilon/\delta]\tau$.

Proof. By induction and appeal to Lemma 24. □

Theorem 6 (Preservation). *If* $\Delta; \Gamma \vdash_i E : \tau$ *and* $E \longrightarrow E'$, *then* $\Delta; \Gamma \vdash_i E' : \tau$.

Proof. By induction on $E \longrightarrow E'$.
Congruence rules ($\approx$): By the induction hypothesis.
Case (β): By the substitution lemma (21).
Case ($\tau\beta$): By the type substitution lemma (23).
Case ($\upsilon\beta$): By the evidence substitution lemma (25).

□

Definition 26 (Values)**.** The three abstraction forms of expressions E are considered *values*. Other expression forms are not values.

Lemma 27 (Canonical forms). *Assume E is a value.*
(1) If $\Delta; \Gamma \vdash_i E : \tau_1 \to \tau_2$*, then* $E = \lambda x{:}\tau_1.E'$ *for some* E'.
(2) If $\Delta; \Gamma \vdash_i E : \forall\alpha{:}\kappa.\tau$*, then* $E = \Lambda\alpha{:}\kappa.E'$ *for some* E'.
(3) If $\Delta; \Gamma \vdash_i E : (\delta{:}\pi) \Rightarrow \tau$*, then* $E = \lambda\delta{:}\pi.E'$ *for some* E'.

Proof. Straightforward case analysis. □

Theorem 7 (Progress). *If* $\Delta; \epsilon \vdash_i E : \tau$*, then either* E *is a value or there exists* E' *such that* $E \longrightarrow E'$.

Proof. By induction on the typing derivation.

Case Var: Impossible.
Case →E: We use the induction hypothesis on the first premise, learning that E_1 is either a value or steps. If it steps, we are done by @≈. If it is a value, we invoke Lemma 27 and step by β.
Case →I: E is a value.
Other cases: Similar to the cases above.

□

C COMPILATION

We define *compilation* by the set of judgments below. Compilation relies on two auxiliary structures and one auxiliary judgment:

$$\begin{array}{ll} \psi ::= \epsilon \mid \psi, \alpha{:}\kappa & \text{telescopes} \\ \mu ::= \epsilon \mid \mu, \alpha \mapsto \psi \mid \mu, \pi \mapsto \delta & \text{compilation contexts} \end{array}$$

$$\boxed{\Delta \vdash_i \psi \ \mathsf{tele}}$$

$$\frac{}{\Delta \vdash_i \epsilon \ \mathsf{tele}} \qquad \frac{\Delta \vdash_i \kappa \ \mathsf{kind} \quad \Delta, \alpha{:}\kappa \vdash_i \psi \ \mathsf{tele}}{\Delta \vdash_i \alpha{:}\kappa, \psi \ \mathsf{tele}}$$

Lemma 28 (Weakening in telescopes). *If* $\Delta \subseteq \Delta'$ *and* $\Delta \vdash_i \psi \ \mathsf{tele}$*, then* $\Delta' \vdash_i \psi \ \mathsf{tele}$.

Proof. Straightforward induction. □

The judgments defining compilation follow:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Assumptions regarding compilation:

(1) For every C, lookup(C) is a list of types $\overline{\pi_i}$ such that π_i is the constraint induced when C is applied to its ith argument.
(2) For every $C : \kappa$ in the source program, we have $C : [\text{lookup}(C)/dom(\psi)]\kappa'$ in the internal language, where $\kappa; \epsilon \rightsquigarrow \kappa'; \psi$.
(3) For every $L : \overline{\kappa_i} \to \mathsf{pred}$ in the source program, we have $L : \overline{\forall\psi_i.\kappa_i' \to} \mathsf{o}$ in the internal language, where $\kappa_i; \epsilon \rightsquigarrow \kappa_i'; \psi_i$.
(4) When $C : \overline{\kappa_i} \to s$ in the source language and lookup$(C) = \overline{\pi_i}$, we have $\pi_i : \forall\psi_{1<j\leq i}.\kappa_i'$ where $\kappa_i; \epsilon \rightsquigarrow \kappa_i'; \psi_i$.
(5) Classes L are treated identically to type constants C for the previous assumption.
(6) For every L, lookup(L) is a list of types $\overline{\top_{\kappa_i'}}$, where the length of the list equals the length of the list in $L : \overline{\kappa_i} \to \mathsf{o}$ and the κ_i are chosen to uphold the previous assumption.
(7) There is a solver for predicates π such that solve$(\pi) \rightsquigarrow v$ implies that, when $\epsilon \mid \epsilon \vdash \pi\ \mathsf{pred} \rightsquigarrow_\epsilon \pi'$, then $\epsilon \vdash_i v : \pi'$.

The kind compilation judgment takes two inputs and produces two outputs. This is necessary because compiling a kind produces a kind that quantifies over fresh variables, and so we have to communicate these variables out to the caller. In addition, a compound kind, such as $\star \to \star \to \star$, will quantify over more variables at each arrow. (See the `lift` example in the main text.) Thus, the input ψ are all the bindings that we have created so far, scanning left to right; the output ψ are those that will have to be quantified over when quantifying over a variable of the output kind.

Definition 29 (Vector typing). We write $\Delta \vdash_i \overline{\tau} : \psi$ to mean that the types $\overline{\tau}$ have the right kinds to be used as arguments to a type of kind $\forall\psi.\kappa$.

Lemma 30 (Kind compilation). *For all source-language kinds κ and telescopes ψ, if $\Delta \vdash_i \psi$ tele and $\kappa; \psi \rightsquigarrow \kappa'; \psi'$, then $\Delta \vdash_i \psi'$ tele and $\Delta, \psi' \vdash_i \kappa'$ kind.*

Proof. By induction on the structure of κ.

Case $\kappa = \star$: The result is trivial.

Case $\kappa = \kappa_1 \to \kappa_2$: We use the metavariables as they occur in the rule:

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

We consider the resulting kind one piece at a time:

- We know $\Delta, \psi_2' \vdash_i \psi_1$ tele by the induction hypothesis. We thus must show that $\Delta, \psi_2', \psi_1 \vdash_i (\alpha{:}\kappa_1') \to \ell\ \mathit{dom}(\psi)\ \mathit{dom}(\psi_1)\ \alpha \Rightarrow \kappa_2'$ kind.
- We know $\Delta, \psi_2', \psi_1 \vdash_i \kappa_1'$ kind by the induction hypothesis. We thus must show that $\Delta, \psi_2', \psi_1, \alpha{:}\kappa_1' \vdash_i \ell\ \mathit{dom}(\psi)\ \mathit{dom}(\psi_1)\ \alpha \Rightarrow \kappa_2'$ kind.
- We must show $\Delta, \psi_2', \psi_1, \alpha{:}\kappa_1' \vdash_i \ell\ \mathit{dom}(\psi)\ \mathit{dom}(\psi_1)\ \alpha : \mathrm{o}$. Let the kind environment in that judgment be denoted by Δ'. By construction of ψ_2', we see that $\Delta' \vdash_i \ell : \forall\psi.\forall\psi_1.\kappa_1' \to \mathrm{o}$. By repeated use of the application rule for types and using the weakening lemma (Lemma 17), we have our desired outcome. (Note that the substitution in the kind of the application rule has no effect, because we end up substituting variables for themselves.)
- It remains only to show that $\Delta', \delta{:}\ell\ \mathit{dom}(\psi)\ \mathit{dom}(\psi_1)\ \alpha \vdash_i \kappa_2'$ kind. In order to use the induction hypothesis on $\kappa_2; \psi' \rightsquigarrow \kappa_2'; \psi_2$, we must show that $\Delta \vdash_i \psi'$ tele. We get this result from a fresh use of the induction hypothesis on $\kappa_1; \epsilon \rightsquigarrow \kappa_1'; \psi_1$ and weakening (Lemma 28). We know use the induction hypothesis to get $\Delta, \psi_2 \vdash_i \kappa_2'$ kind, and weakening (Lemma 17) gives us our desired result.

□

Lemma 31 (Kind compilation yields fresh bindings). *If $\kappa; \psi \rightsquigarrow \kappa'; \psi'$, all variables bounds in ψ' are fresh. In particular, $\mathit{dom}(\psi) \cap \mathit{dom}(\psi') = \emptyset$.*

Proof. Straightforward induction. □

Lemma 32 (Bindings during kind compilation). *If $\kappa; \psi_0 \rightsquigarrow \kappa'; \psi_1$, then $\kappa; \psi_2, \psi_0 \rightsquigarrow \kappa''; \psi_3$, where $\kappa'' = [\overline{\tau}/\mathit{dom}(\psi_1)]\kappa'$ and $\overline{\tau} = [\tau_0\ \mathit{dom}(\psi_2) \mid \tau_0 \leftarrow \mathit{dom}(\mathit{binds}_3)]$. Furthermore, for every i, if $\psi_{1i} = \ell_1 : \kappa_i$, then $\psi_{3i} = \ell_2 : \forall\psi_2.\kappa_i$.*

Proof. By induction on the structure of κ.

Case $\kappa = \star$: Trivially true.

Case $\kappa = \kappa_1 \to \kappa_2$: From the rule, we see that $\kappa' = \forall\psi_1'.(\alpha{:}\kappa_1') \to \ell_1\ \mathit{dom}(\psi_0)\ \mathit{dom}(\psi_1')\ \alpha \Rightarrow \kappa_2'$, where $\kappa_2; \psi_0, \psi_1', \alpha{:}\kappa_1' \rightsquigarrow \kappa_2'; \psi_2'$. We also see that $\kappa'' = \forall\psi_1'.(\alpha{:}\kappa_1') \to \ell_2\ \mathit{dom}(\psi_2)\ \mathit{dom}(\psi_0)\ \mathit{dom}(\psi_1')\ \alpha \Rightarrow \kappa_2''$, where $\kappa_2; \psi_2, \psi_0, \psi_1', \alpha{:}\kappa_1' \rightsquigarrow \kappa_2''; \psi_2''$. We must show that $\kappa'' = [\overline{\tau}/(\ell_1, \mathit{dom}(\psi_2'))]\kappa'$, where $\overline{\tau} = [\tau_0\ \mathit{dom}(\psi_2) \mid \tau_0 \leftarrow \ell_2, \mathit{dom}(\psi_2'')]$. In other words, $\overline{\tau} = (\ell_2\ \mathit{dom}(\psi_2)), [\tau_0\ \mathit{dom}(\psi_2) \mid \tau_0 \leftarrow \mathit{dom}(\psi_2'')]$. Propagating the substitution into κ' (and using Lemmas 31 and 30 to discard identity substitutions), we get that we want to show $\kappa'' = \forall\psi_1'.(\alpha{:}\kappa_1') \to \ell_2\ \mathit{dom}(\psi_2)\ \mathit{dom}(\psi_0)\ \mathit{dom}(\psi_1')\ \alpha \Rightarrow$

$[\mathsf{tail}(\overline{\tau})/\mathit{dom}(\psi_2')]\kappa_2'$. It remains only to show that $\kappa_2'' = [\mathsf{tail}(\overline{\tau})/\mathit{dom}(\psi_2')]\kappa_2'$. This comes directly from the induction hypothesis, and so we are done.

The relationship between the ψ_3 and the ψ_1 is by straightforward use of the inductive hypothesis.

□

Lemma 33 (Kind compilation bindings are independent). *If $\kappa; \psi_0 \rightsquigarrow \kappa'; \psi_1$, then no variable bound in ψ_1 is used in a kind in ψ_1.*

Proof. Straightforward induction. □

Lemma 34 (Type compilation). *Assume $\Delta \mid P \rightsquigarrow \Delta'; \mu$.*

(1) If $\kappa; \epsilon \rightsquigarrow \kappa'; \psi$ and $P \mid \Delta \vdash \sigma : \kappa \rightsquigarrow_\mu \tau; \overline{\tau'}$, then $\Delta' \vDash_i \tau' : [\overline{\tau'}/\mathit{dom}(\psi)]\kappa'$ and $\Delta' \vDash_i \overline{\tau'} : \psi$.
(2) If $P \mid \Delta \vdash \pi$ pred $\rightsquigarrow_\mu \pi'$, then $\Delta' \vDash_i \pi' : \mathsf{o}$.
(3) If $P \mid \Delta \vdash \pi$ pred $\rightsquigarrow_\mu \pi'$ and $P \Vdash \pi \rightsquigarrow_\mu v$, then $\Delta' \vDash_i v : \pi'$.

Proof. By mutual induction on the provided type derivation.

Case $\sigma = \alpha$**:** We must show $\Delta' \vDash_i \alpha : [\mathit{dom}(\psi)/\mathit{dom}(\psi)]\kappa'$ (noting that $\alpha \mapsto \psi \in \mu$). This follows from the definition of Δ' and the compilation of κ. Note that ψ is a sub-context of Δ'.

Case $\sigma = C$**:** We must show $\Delta' \vDash_i C : [\mathsf{lookup}(C)/\mathit{dom}(\psi)]\kappa'$. This is one of our compilation assumptions.

Case $\sigma = \forall\alpha{:}\kappa_2.\sigma_2$**:** We must show $\Delta' \vDash_i \forall\psi_2.\forall\alpha{:}\kappa_2'.\tau : \star$, where $\kappa_2; \epsilon \rightsquigarrow \kappa_2'; \psi_2$ and $P \mid \Delta, \alpha{:}\kappa_2 \vdash \sigma_2 : \star \rightsquigarrow_{\mu,\alpha\mapsto\psi_2} \tau;$. We take this in pieces:

- First, we show $\Delta' \vDash_i \psi_2$ tele. This comes from the kind-compilation lemma (Lemma 30). Thus, we now must show $\Delta', \psi_2 \vDash_i \forall\alpha{:}\kappa_2'.\tau : \star$.
- Our next step is to show $\Delta', \psi_2 \vDash_i \kappa_2'$ kind. This also comes directly from Lemma 30. We now must show $\Delta', \psi_2, \alpha{:}\kappa_2' \vDash_i \tau : \star$.
- By the definition of compilation, we see that $\Delta, \alpha{:}\kappa_2 \mid P \rightsquigarrow \Delta', \psi_2, \alpha{:}\kappa_2'; \mu, \alpha \mapsto \psi_2$. We can thus use the induction hypothesis to get our desired result.

Case $\sigma = \tau_1\,\tau_2$**:** We adopt the metavariable names from the rule:

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

We must show $\Delta' \vDash_i \tau_1'\,\overline{\tau'}\,\tau_2'\,v : [\overline{\tau''}/\mathit{dom}(\psi_2)]\kappa_2'$ where $\kappa_2; \epsilon \rightsquigarrow \kappa_2'; \psi_2$.

- Let κ_1' and ψ_1 be defined by $\kappa_1; \epsilon \rightsquigarrow \kappa_1'; \psi_1$.
- Then $\kappa_1 \to \kappa_2; \epsilon \rightsquigarrow \forall\psi_1.(\alpha{:}\kappa_1') \to \ell\ \mathit{dom}(\psi_1)\,\alpha \Rightarrow \kappa_2''; \ell{:}(\forall\psi_1.\kappa_1' \to \mathsf{o}), \psi_2'$ where $\kappa_2; \psi_1, \alpha{:}\kappa_1' \rightsquigarrow \kappa_2''; \psi_2'$. Let κ_0 denote the output kind of compiling $\kappa_1 \to \kappa_2$.
- Thus, the induction hypothesis tells us that $\Delta' \vDash_i \tau_1' : [\overline{\tau}/(\ell, \mathit{dom}(\psi_2'))]\kappa_0$.
- Lemmas 31 and 30, taken together, tell us that this substitution does not affect ψ_1 nor κ_1'.
- Propagating the substitution gives us $\Delta' \vDash_i \tau_1' : \forall\psi_1.(\alpha{:}\kappa_1') \to \mathsf{head}(\overline{\tau})\,\mathsf{tail}(\overline{\tau})\,\alpha \Rightarrow [\overline{\tau}/(\ell, \mathit{dom}(\psi_2'))]\kappa_2''$.
- We now must show that $\Delta' \vDash_i \overline{\tau'} : \psi_1$. This comes directly from the induction hypothesis.
- Using the application typing rule, we see that $\Delta' \vDash_i \tau_1'\,\overline{\tau'} : (\alpha{:}([\overline{\tau'}/\psi_1]\kappa_1')) \to \mathsf{head}(\overline{\tau})\,\mathsf{tail}(\overline{\tau})\,\alpha \Rightarrow [\overline{\tau'}/\psi_1][\overline{\tau}/(\ell, \mathit{dom}(\psi_2'))]\kappa_2''$, where we have omitted substitutions that are guaranteed not to have an effect.
- We now must show $\Delta' \vDash_i \tau_2' : [\overline{\tau'}/\psi_1]\kappa_1'$. This comes directly from the induction hypothesis.
- We now know $\Delta' \vDash_i \tau_1'\,\overline{\tau'}\,\tau_2' : \mathsf{head}(\overline{\tau})\,\mathsf{tail}(\overline{\tau})\,\tau_2' \Rightarrow [\tau_2'/\alpha][\overline{\tau'}/\psi_1][\overline{\tau}/(\ell, \mathit{dom}(\psi_2'))]\kappa_2''$.

- We now must show $\Delta' \models_i v : \mathsf{head}(\overline{\tau})\,\mathsf{tail}(\overline{\tau})\,\tau_2'$. The induction hypothesis tells us that, if $P \mid \Delta \vdash \tau_1 \mathbin{@} \tau_2 \text{ pred} \rightsquigarrow_\mu \pi'$, then $\Delta' \models_i v : \pi'$. Examining the rule in the compilation of predicates, we see that $\pi' = \mathsf{head}(\overline{\tau})\,\mathsf{tail}(\overline{\tau})\,\tau_2'$, as desired.
- We now know $\Delta' \models_i \tau_1'\,\overline{\tau'}\,\tau_2'\,v : [\tau_2'/\alpha][\overline{\tau'}/\psi_1][\overline{\tau}/(\ell, dom(\psi_2'))]\kappa_2''$. By Lemma 32, we see that $\kappa_2'' = [[\tau_0\; dom(\psi_1)\,\alpha \mid \tau_0 \leftarrow dom(\psi_2')]/dom(\psi_2)]\kappa_2'$. To finish this part of the proof, we need only show that $\overline{\tau''} = [\tau_0\,\overline{\tau'}\,\tau_2' \mid \tau_0 \leftarrow \mathsf{tail}(\overline{\tau})]$, which it does by definition.

We still must show $\Delta' \models_i \overline{\tau''} : \psi_2$.

- The induction hypothesis tells us that $\Delta' \models_i \overline{\tau} : \ell{:}(\forall\psi_1.\kappa_1' \to \mathrm{o}), \psi_2'$. Looking at only the tail of the list, we see $\Delta' \models_i \mathsf{tail}(\overline{\tau}) : \psi_2'$ (using the fact that, according to the definition of kind compilation, ℓ does not appear later in the bindings output from that function).
- The induction hypothesis also tells us that $\Delta' \models_i \overline{\tau'} : \psi_1$.
- By Lemma 33, we know that we can view the judgment we must show simply as a list of typing judgments—there is no dependency among the components.
- Fix i. We must show $\Delta' \models_i \tau_i'' : \mathsf{range}(\psi_{2i})$, where $\tau_i'' = \tau_{i+1}\,\overline{\tau'}\,\tau_2'$.
- We see that $\Delta' \models_i \tau_{i+1} : \mathsf{range}(\psi_{2i}')$.
- Lemma 32 tells us that $\mathsf{range}(\psi_{2i}') = \forall\psi_1.\forall\alpha{:}\kappa_1'.\mathsf{range}(\psi_{2i})$.
- Thus, $\Delta' \models_i \tau_{i+1} : \forall\psi_1.\forall\alpha{:}\kappa_1'.\mathsf{range}(\psi_{2i})$.
- We can thus say that $\Delta' \models_i \tau_{i+1}\,\overline{\tau'} : \forall\alpha{:}([\overline{\tau'}/dom(\psi_1)]\kappa_1').[\overline{\tau'}/dom(\psi_1)]\mathsf{range}(\psi_{2i})$.
- The induction hypothesis tells us that $\Delta' \models_i \tau_2' : [\overline{\tau'}/dom(\psi_1)]\kappa_1'$.
- We thus conclude that $\Delta' \models_i \tau_{i+1}\,\overline{\tau'}\,\tau_2' : [\tau_2'/\alpha][\overline{\tau'}/dom(\psi_1)]\mathsf{range}(\psi_{2i})$. That is $\Delta' \models_i \tau_i'' : [\tau_2'/\alpha][\overline{\tau'}/dom(\psi_1)]\mathsf{range}(\psi_{2i})$.
- We notice that ψ_2 comes from a context that does not mention ψ_1 nor α. Thus, the substitution has no effect, and we are done.

Case $\sigma = \pi \Rightarrow \rho$**:** We must show $\Delta' \models_i (\delta{:}\pi') \Rightarrow \tau : \star$, where $P \mid \Delta \vdash \pi \text{ pred} \rightsquigarrow_\mu \pi'$ and $P, \pi \mid \Delta \vdash \rho : \star \rightsquigarrow_{\mu,\pi\mapsto\delta} \tau; \overline{\tau}$.

- We first show $\Delta' \models_i \pi' : \mathrm{o}$. This comes directly from the induction hypothesis.
- We then must show $\Delta', \delta{:}\pi' \models_i \tau : \star$. In order to use the induction hypothesis, we first must show $\Delta \mid P, \pi \rightsquigarrow \Delta', \delta{:}\pi'; \mu, \pi \mapsto \delta$. This fact comes from the definition of compilation of P, and so we are done.

Case $\pi = \tau_1 \mathbin{@} \tau_2$**:** We must show $\Delta' \models_i \pi'\,\overline{\tau'}\,\tau_2' : \mathrm{o}$, where $P \mid \Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \rightsquigarrow_\mu \tau_1'; \pi', \overline{\tau}$ and $P \mid \Delta \vdash \tau_2 : \kappa_1 \rightsquigarrow_\mu \tau_2'; \overline{\tau'}$.

- The induction hypothesis (and the definition of kind compilation) tells us $\Delta' \models_i \pi', \overline{\tau} : \ell{:}(\forall\psi_1.\kappa_1' \to \mathrm{o}), \psi_2$, where $\kappa_1; \epsilon \rightsquigarrow \kappa_1'; \psi_1$ and $\kappa_2; \psi_1, \alpha{:}\kappa_1' \rightsquigarrow \kappa_2'; \psi_2$.
- Thus, $\Delta' \models_i \pi' : \forall\psi_1.\kappa_1' \to \mathrm{o}$.
- The application rule tells us $\Delta' \models_i \pi'\,\overline{\tau'} : [\overline{\tau'}/\psi_1]\kappa_1' \to \mathrm{o}$.
- The induction hypothesis tells us $\Delta' \models_i \tau_2' : [\overline{\tau'}/\psi_1]\kappa_1'$.
- One more use of the application rule gives us our desired outcome.

Case $\pi = L\,\overline{\tau_i}$**:** This case follows from the induction hypothesis and our assumption of the compilation of class kinds.

Case $P \Vdash \pi \rightsquigarrow_\mu v$ **where** $\pi \mapsto \delta \in \mu$**:** We must show $\Delta' \models_i \delta : \pi'$, where $P \mid \Delta \vdash \pi \text{ pred} \rightsquigarrow_\mu \pi'$. This comes directly from the definition of compilation of predicate environments.

Case $P \Vdash \pi \rightsquigarrow_\mu v$ **where** $\mathsf{solve}(\pi) \rightsquigarrow v$**:** By assumption.

□

Lemma 35 (Compilation of function types). *If* $P \mid \Delta \vdash \tau_1 \to \tau_2 : \star \rightsquigarrow_\mu \tau_3; \epsilon$, *then* $\tau_3 = \tau_4 \to \tau_5$ *where* $P \mid \Delta \vdash \tau_1 : \star \rightsquigarrow_\mu \tau_4; \epsilon$ *and* $P \mid \Delta \vdash \tau_2 : \star \rightsquigarrow_\mu \tau_5; \epsilon$.

Proof. Recall that $\tau_1 \to \tau_2$ means $(\to)\, \tau_1 \lozenge\, \tau_2 \lozenge$. We can then get the desired result recalling that compiling a type of kind $\star$ produces no list of types as output and that compiling $(\to)$ @ τ_1 and $(\to)\, \tau_1$ @ τ_2 both produce $\top$ constraints. □

Lemma 36 (Expression compilation). *If* $\Delta \mid P \rightsquigarrow \Delta'; \mu$ *and* $P \mid \Delta; \Gamma \vdash E : \sigma \rightsquigarrow_\mu E'$, *then* $P \mid \Delta \vdash \sigma : \star \rightsquigarrow_\mu \tau; \epsilon$ *and* $\Delta'; \Gamma' \vdash_{\text{i}} E' : \tau$, *where* Γ' *is the result of compiling each type in* Γ.

Proof. If E is well-typed, then $P \mid \Delta \vdash \sigma : \star$ by Theorem 1. Thus, we know $P \mid \Delta \vdash \sigma : \star \rightsquigarrow_\mu \tau; \epsilon$ (noting that compiling a type of kind $\star$ always yields an empty list of types as its second output) by the similarity in the structure of source-language typing and compilation.

For the second result, proceed by induction, frequently appealing to Lemma 34.

Case (var): By the definition of Γ'.
Case (let): By induction and straightforward use of typing rules.
Case (→E): By induction, appeal to Lemma 35, and straightforward use of typing rules.
Case (→I): By induction, appeal to Lemma 35, and straightforward use of typing rules.
Case (⇒E): By induction and straightforward use of typing rules.
Case (⇒I): By induction and straightforward use of typing rules.
Case (∀E): By induction and straightforward use of typing rules.
Case (∀I): By induction, appeal to Lemma 30, and straightforward use of typing rules.

□

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

Proof. Corollary of Lemma 36. □

D PROTOTYPE IMPLEMENTATION DETAILS

D.1 Polymorphic Kinds

The three-parameter constraint `f @ a = r` implemented in our prototype works at any kinds κ_1 and κ_2, as long as `f ::` $\kappa_1 \to \kappa_2$, `a ::` κ_1, and `r ::` κ_2. However, Hugs does not support general polymorphism in kinds. We thus had to add custom code to check the kinds of the components of the three-part constraint. For similar reasons, Hugs' normal functional dependencies could not fix `r` from `f` and `a`, requiring more custom handling.

D.2 Syntactic Sugar

Our implementation supports both @ constraints and also `wft` constraints, where we understand a `wft` constraint to expand to a sequence of @ constraints. For example, we write `wft (a b c)` to mean `(a @ b, a b @ c)` (or, in the 3-parameter version, `a @ b = ab`, `ab @ c = abc` for fresh type variables `ab`, `abc`).

D.3 Limitations

Our implementation is just an experimental prototype and it is not ready or intended for use in general program development. In particular, the prototype does not implement general support for user definitions of partial type constructors (i.e., for definitions of datatypes that require a context to specify constraints on their parameters). Even so, our test scenario is not trivial because the implementation must still generate and handle constraints for arbitrary type applications of the form `m t`, where `m` is a variable of some higher-kinded type. Any such example leaves open the possibility that `m` might later be instantiated to either a partial or a total type constructor, at some other point in the program, and so each such occurrence will generate a constraint of the form `m @ t = u` for some type variable `u`.

This limitation in the prototype also means that we were unable to handle datatype definitions with higher-kinded parameters, like the following example from the `Control.Arrow` library:

```
newtype Kleisli m a b = Kleisli (a → m b)
```

For proper use in our system, this definition should be written as follows, with the `Monad m` constraint reflecting intended usage (something that the authors of this particular file had not already chosen to do) and the `m @ b` constraint ensuring that the type `m b` is well defined:

```
newtype (Monad m, m @ b) ⇒ Kleisli m a b = Kleisli (a → m b)
```

Our workaround for this particular example was to keep the original **newtype** definition in place but comment out the associated **instance** declarations, which, in any case, were not used elsewhere in our collection of files, and amounted to 19 lines of code (out of 320 lines in the full `Control.Arrow` source file). We encountered a handful of similar examples, particularly for definitions of monad transformers in some of the `Control.Monad.*` libraries, that cannot be handled by our current implementation, and so we opted not to include these in our tests (i.e., they are not counted in the collection of 169 source files listed above). Nevertheless, given our experiences with the `Control.Applicative`, `Control.Arrow`, and other parts of `Control.Monad` reported below, we would not anticipate any fundamental problems with including these examples if the prototype were extended to support datatype definitions like the one for `Kleisli` shown above.

We also found that it was necessary to comment out some of the default definitions that were included in class definitions in five of our test files. However, we attribute this to a bug in the Hugs implementation that we used as a starting point for our prototype, and believe that it has no bearing on the type system described in this paper. To illustrate this, consider the following fragment of the definition of the `Foldable` class in the `Data.Foldable` library:

```
class Foldable t where
    fold :: Monoid m ⇒ t m → m
    fold = foldMap id
    ...
```

In our system, the declared type for the `fold` member will automatically be translated to include an additional constraint of the form `t @ m = a` for some fresh type variable `a` that is functionally dependent on `t` and `m`. Unfortunately, the Hugs type checker rejects this particular definition because it tries (incorrectly) to prove that the definition is fully polymorphic in `a`, failing to account for the dependency that indicates, instead, that there is at most one valid choice of `a` for any given combination of `t` and `m`. We concluded that fixing this oversight in Hugs was beyond the scope of our experiment, and were satisfied to note that the definition does type check as expected when it is lifted outside the **class** definition in an appropriate way.