Coherent Explicit Dictionary Application for Haskell

Thomas Winant
imec-DistriNet, KU Leuven
Belgium
thomas.winant@cs.kuleuven.be

Dominique Devriese
imec-DistriNet, KU Leuven
Belgium
dominique.devriese@cs.kuleuven.be

Abstract

Type classes are one of Haskell's most popular features and extend its type system with ad-hoc polymorphism. Since their conception, there were useful features that could not be offered because of the desire to offer two correctness properties: coherence and global uniqueness of instances. Coherence essentially guarantees that program semantics are independent from type-checker internals. Global uniqueness of instances is relied upon by libraries for enforcing, for example, that a single order relation is used for all manipulations of an ordered binary tree.

The features that could not be offered include explicit dictionary application and local instances, which would be highly useful in practice. In this paper, we propose a new design for offering explicit dictionary application, without compromising coherence and global uniqueness. We introduce a novel criterion based on GHC's type argument roles to decide when a dictionary application is safe with respect to global uniqueness of instances. We preserve coherence by detecting potential sources of incoherence, and prove it formally. Moreover, our solution makes it possible to use local dictionaries. In addition to developing our ideas formally, we have implemented a working prototype in GHC.

CCS Concepts • **Theory of computation → Type structures**; • **Software and its engineering → Functional languages**;

Keywords Haskell, type classes, dictionaries, coherence

ACM Reference Format:
Thomas Winant and Dominique Devriese. 2018. Coherent Explicit Dictionary Application for Haskell. In *Proceedings of the 11th ACM SIGPLAN International Haskell Symposium (Haskell '18), September 27-28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3242744.3242752

Haskell '18, September 27-28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5835-4/18/09…$15.00
https://doi.org/10.1145/3242744.3242752

1 Introduction

Haskell's type class mechanism, introduced by Wadler and Blott [19], is a powerful abstraction providing a disciplined form of *ad-hoc* polymorphism. A type class consists of a number of *methods* that may be implemented differently for different types. For example, the following *Eq* type class defines a single method (==) which determines when two values of a type a are equal and the instance declaration instantiates the method for argument type *Int*.

class *Eq a* **where** (==) :: $a \to a \to$ *Bool*

instance *Eq Int* **where** (==) = …

We can also define that a *Person*, consisting of a name (*String*) and a Social Security number (*Int*), is equal to another *Person* when the Social Security numbers are equal:

data *Person* = *Person* { *name* :: *String*, *ssn* :: *Int* }

instance *Eq Person* **where** (*p1* == *p2*) = (*ssn p1* == *ssn p2*)

Another function f can then freely use the type class's methods on arbitrary types, as long as (1) an instance is defined for those types, or (2) the burden is passed to f's caller by annotating a *type class constraint* in f's type. Such a constraint specifies that f can only be used on types implementing the type class. Type class instances are resolved implicitly, freeing the programmer from passing the right instance each time a type class constraint is used. For example, two *Person*s can be checked for equality as follows:

```
> Person "Arnold" 2 == Person "Bernard" 2
True
```

However, since the introduction of the type class mechanism there has been an important restriction: *only one type class instance per type is allowed* and it is not possible to resolve type class constraints with alternative instances. Although this restriction was introduced for good reasons (see below), it is also one that programmers often bump into. Take, for example, the *Semigroup* type class which models an algebraic structure with an associative binary operation:

class *Semigroup a* **where** ($\diamond$) :: $a \to a \to a$

For integer numbers (*Integer*), a wide variety of useful instances exist:

instance *Semigroup Integer* **where** ($\diamond$) = (+)
instance *Semigroup Integer* **where** ($\diamond$) = ($\times$)
instance *Semigroup Integer* **where** $x \diamond y = x + y$ `` `mod` `` 10

The last example uses addition modulo 10, but ideally a parameter would be used for this instead of hard-coding it to 10. Unfortunately, Haskell only allows a single instance for *Semigroup Integer*.

Over the years, a number of workarounds and partial solutions have been proposed for this instance uniqueness restriction. For example, one workaround is based on the idea of defining **newtype** wrapper types that are identical to an underlying type, except that they have different type class instances. Another workaround is to define copies of ad-hoc polymorphic functions that avoid type classes and instead take method implementations as explicit arguments, for example *sortBy* as an alternative for *sort* in the Haskell Prelude. Perhaps the most general workaround was proposed by Kiselyov and Shan [11], in the form of a primitive for generating a fresh local type for which arbitrary instances can then be defined. This last workaround has the advantage that the instances can be defined in terms of values that are only available locally. Unfortunately, all of these workarounds are cumbersome to use and require boilerplate code like wrapper type definitions or instances for artificial types.

Existing proposals for adding explicit dictionary instantiations or local instances in the context of Haskell have failed to gain adoption [7, 10]. One reason is that they break one or both of the following two important properties of Haskell: *coherence* and *global uniqueness of instances*. The former, coherence, is a general sanity property related to type-inference performed by compilers: it states that whenever a type system can derive more than one valid typing derivation for a given piece of code, this choice should not influence the behaviour of the code. In other words, a programmer must be able to freely ignore the internal workings of type inference and trust that their code's behaviour does not depend on any choices the type inferencer makes.

The property of global uniqueness of instances (also known as *class* or *instance coherence*) is more Haskell-specific: some existing Haskell code relies on the property that a given type class will only ever be instantiated in a single way for a given type. For example, the Haskell containers package defines an abstract type of finite sets (implemented using balanced search trees), which can be used for any type of elements a for which $Ord\ a$ is defined. The functions in this library all have an $Ord\ a$ constraint, but their correctness relies on the fact that the same $Ord\ a$ instance will be used for every operation on any given set. For example, one can easily break the search tree invariant of a *Set*, by using *Data.Set.insert* with different $Ord\ a$ instances.

1.1 Contributions and Outline

In this paper, we propose a new form of explicit dictionary instantiation that preserves coherence and that is safe with respect to global uniqueness of instances, but can be directly applied to common use cases. More concretely, we contribute the following:

We propose a design for explicit dictionary instantiation in Haskell, with the following desirable characteristics:

- It directly applies to most use cases we found in the literature and libraries.
- It is relatively easy to implement and avoids modifying the constraint solver.
- Explicit dictionaries are allowed to mention local variables, i.e. we provide the expressiveness of local instances.
- It is coherent and does not break code relying on global instance uniqueness (see below).

We also propose two (optional) features that complement our design and provide useful expressive power:

- A way to obtain the dictionary for an implicitly resolved constraint,
- *Dictionary instances*: a way to implement an instance by providing a dictionary.

We discuss the design in Section 2 and formalise it in Section 3. A core part of our design is a criterion that detects when an explicit dictionary application can be allowed without compromising coherence. In Section 5, we explain the criterion and formally prove coherence. The proof is itself technically novel: it is based on prior work, but avoids the complexity of an algorithmic typing relation or constraint solver. The full proof can be found in [20].

Section 4 presents a novel criterion that detects when an explicit dictionary application can be safely allowed without breaking existing code that relies on global instance uniqueness. The criterion uses GHC roles and we show informally that it prevents breaking existing code that relies on global instance uniqueness. Specifically, we demonstrate that any explicit dictionary application can be simulated (at the cost of readability and additional boilerplate) by regular code using newtypes, regular type class instances and safe coercions [3].

We have a prototype implementation in GHC of our proposal to demonstrate its practical use, see Section 6. While it is not polished enough for production use, it successfully compiles all the examples shown in this paper.[1]

Finally, we discuss some applications in Section 7, related work in Section 8, and we conclude in Section 9.

2 Our Proposal in More Detail

In this section, we discuss syntactic elements of our proposed Haskell extension. These are discussed first for ease of presentation, but they are not the most innovative parts of our proposal (those are the safety criterions, discussed in the next sections). All syntactic aspects should be considered tentative as there may still be syntactic conflicts with other features or similar problems that we have not noticed.

[1] For brevity, some details were omitted in the paper. The full code can be found at https://github.com/mrBliss/ghc.

2.1 Exposing Type Class Dictionaries

First, we expose the dictionary records that the compiler generates, to the user. Consider a class $C\ \overline{a}$ with superclasses $\overline{C_{super}}$ and methods $\overline{m}$ with types $\overline{\sigma}$, where a horizontal bar is used to indicate that there can be zero or more:

class $\overline{C_{super}} \Rightarrow C\ \overline{a}$ **where** $\overline{m :: \sigma}$

The following dictionary record is then exposed to the user:

data $C.Dict\ \overline{a} = C.Dict\ \{\overline{parent_i :: C_{super}.Dict}, \overline{m :: \sigma}\}$

- As type classes and data types are in the same namespace, and the type class and data type have different kinds ($\ldots \rightarrow Constraint$ vs $\ldots \rightarrow Type$), we cannot reuse the name C for the dictionary. Therefore, we currently opt for appending *.Dict* to the type class name. While there is no such conflict for the dictionary data constructor, we use the same suffix for consistency.
- Following GHC's dictionary translation, superclasses are translated to additional fields with as types the corresponding superclass dictionaries. For simplicity, we currently just use *parent1*, *parent2*, ... as the names for these fields where the order is determined by the superclass order in the type class declaration. This requires the `-XDuplicateRecordFields` extension, as each type class with a superclass will have a field named *parent1*.
- Following GHC, a **newtype** is used in case of a single method.
- Currently, default methods and the `MINIMAL` pragma are ignored, so implementations for all methods have to be provided when creating a new dictionary. In practice, one can define smart constructors to make life easier. It should be possible to provide the ability to fill in the missing fields of a dictionary record based on the provided fields by reusing the existing default methods machinery.
- We do not generate field projections for the dictionary records, as these would conflict with the type class methods. One can use pattern-matching to extract the fields, or use the class methods in combination with explicit dictionary application.
- Associated types [5] are not supported (yet).

2.2 Explicit Dictionary Application

The second part of our proposal is to provide syntax to explicitly pass a dictionary to a function expecting one. The syntax we have chosen for now is the following:

$$e\ @\{e_{dict}\}$$

For example, consider the Prelude function $nub :: Eq\ a \Rightarrow [a] \rightarrow [a]$, which removes duplicates from a list. To make a case-insensitive version for *String*s, one can write:

```
eqOn :: Eq b ⇒ (a → b) → Eq.Dict a
eqOn f = Eq.Dict {(==) = λv1 v2 → f v1==f v2}

nubCI :: [String] → [String]
nubCI = nub @{eqOn (map toLower)}
```

In case of ambiguity, i.e. when there are multiple constraints of the same type class in the context, an annotation is needed. For example, consider the following program:

```
eqTuple :: (Eq a, Eq b) ⇒ (a, b) → (a, b) → Bool
eqTuple (a1, b1) (a2, b2) = a1== a2 && b1== b2

alwaysEq :: Eq.Dict t
alwaysEq = Eq.Dict (λ_ _ → True)

> eqTuple @{alwaysEq} (True, 1) (True, 2)
```

Should the *alwaysEq* dictionary be used for $Eq\ a$ or $Eq\ b$? To disambiguate such cases, we allow the user to annotate the constraint to which the dictionary should be passed. The syntax we suggest for this is as follows:

$$e\ @\{e_{dict}\ \textbf{as}\ C\ a\}$$

Where $C\ a$ is the constraint in the context of the type of e to which the dictionary should be passed. For example:

```
> eqTuple @{alwaysEq as Eq b} (True, 1) (True, 2)
True
```

Note that the annotation mentions the original constraint, not the instantiated constraint (*Eq Int* in this case).

To prevent ambiguities, we also cannot allow explicit dictionary applications to terms whose type is inferred. Consider the following example:

```
let eqTuple (a1, b1) (a2, b2) = (a1== a2, b1== b2)
in eqTuple @{alwaysEq as Eq b} (True, 1) (True, 2)
```

The annotation is needed, as there is ambiguity because of the two *Eq* constraints. However, the inferred type of the local *eqTuple* binding could be any of the following:

```
(Eq a, Eq b) → (a, b) → (a, b) → Bool
(Eq a, Eq b) → (b, a) → (b, a) → Bool
(Eq t1, Eq t2) → (t1, t2) → (t1, t2) → Bool
```

If the first type were inferred, the result of the program would be *True*. In case of the second type, *False*. In case of the third type, the explicit dictionary application would not even be used at all. As the programmer cannot predict the type variables chosen during type inference, we require that the type signature of e be specified. This can be done simply by writing a type signature, or by annotating the expression itself with its type. This is very similar to the requirement that types should be specified when using explicit type applications in GHC [8].

Past proposals have often focused on declaring multiple named top-level instances or (nested) local instances [7, 10]. We have intentionally taken a different approach, namely that of explicit dictionary application. An important reason

for this is that we do not have to touch the constraint solver in any way. All the complexity is concentrated in the explicit dictionary application construct, instead of permeating the whole constraint solver. As the constraint solver is a complex beast [18], not having to modify it minimises the cost of our proposal. Additionally, explicit dictionary application and the ability to construct the right dictionary is more explicit and flexible than bringing the right instances in scope. The loss of usability is limited and, in our opinion, acceptable for a feature that we expect to be used sparingly.

This choice also prevents a problem described in previous work [7, 11, 19], namely the loss of principal types in the presence of local instances:

f = **let instance** *Eq Int* **where** ... **in** (==)

Two different types can be inferred for f: either $Eq\ a \Rightarrow a \to a \to Bool$ (ignoring the local instance) or $Int \to Int \to Bool$. Neither type is more general than the other, and there is no principal type for f. If we translate this program to our proposal, we get:

f = **let** *eqInt* = *Eq.Dict* ... **in** (==)

As the programmer does not explicitly apply the dictionary, we can assign the type $Eq\ a \Rightarrow a \to a \to Bool$ to f. If the programmer wants to use the local dictionary, (s)he must explicitly pass it, and we end up with the instantiated type. By forcing the programmer to be more explicit, we avoid this whole issue.

2.3 Dictionary Instances and Instance Dictionaries

A third, optional, part of our proposal are *dictionary instances*, which define a global instance based on a dictionary record instead of implementing all methods. This idea was inspired by [7]. For example, recall the instance declaration for *Person* we defined before. That instance can be defined more concisely as a dictionary instance:

instance *Eq Person* = *eqOn ssn*

The general syntax is (where *TC* is a type class, see Fig. 2):

instance $\overline{C_{context}} \Rightarrow TC\ \overline{\tau} = e$

This powerful construct allows for greater code reuse when writing instances declarations. It generalises the *default signatures* and *generalised newtype deriving* features of GHC. In fact, the functionality of *deriving via* [1] can be replicated using this construct. Consider the following example from [1] that defines an *Arbitrary* instance for the *Year* type:

```
newtype Between (l :: Nat) (u :: Nat) = Between Integer

instance (KnownNat l, KnownNat u)
  ⇒ Arbitrary (Between l u) where
  arbitrary = Between ⟨$⟩ choose (natVal @l Proxy,
                                  natVal @u Proxy)

newtype Year = Year Integer
  deriving Arbitary via (Between 1900 2100)
```

We can define the same instance using a dictionary instance:

```
between :: Integer → Integer → Arbitrary.Dict Integer
between lower upper = Arbitrary.Dict
  { arbitrary = choose (lower, upper)
  , shrink    = const [ ] }

newtype Year = Year Integer

instance Arbitrary Year = coerce (between 1900 2100)
```

For more applications of this construct, see Section 7.

An odd aspect of dictionary instances (and the reason we keep it as an optional part of our proposal) is that it introduces the possibility of top-level instances that are not values or trivially-terminating functions. In fact, there is nothing to guarantee that the dictionary term in a dictionary instance will even terminate. We are not sure how to handle this: insert the unevaluated dictionary term during elaboration or perhaps termination-check and/or normalise the dictionary term somehow? More investigation and feedback from the community seems needed before committing to this feature.

A second optional feature we propose is a way to obtain the dictionary for an implicitly-resolved constraint, similar to the query operator ("?") from the implicit calculus [13]. This proposal is optional because details still need to be worked out. It is often useful to obtain the dictionary for an implicit constraint and we can already manually implement functions like the following:

```
eqDict :: Eq a ⇒ Eq.Dict a
eqDict = Eq.Dict {(==) = (==)}
```

However, the compiler should be able to help avoid such boilerplate. Specifically, we imagine it could provide a type class with a method *getDict* and an associated type *DictOf*, that convert a type class constraint to its corresponding dictionary data type:

```
class HasDict (c :: Constraint) where
  type DictOf c :: Type
  getDict :: c ⇒ DictOf c
```

The *getDict* function can be implemented at GHC Core level as a cast. We would forbid the programmer to provide instances of this type class, but instead make the compiler auto-generate them for declared type classes, for example:

```
instance HasDict (Functor f) where
  type DictOf (Functor f) = Functor.Dict f
  getDict = ...
```

2.4 Local Instances

It is worth pointing out that our explicit dictionaries can be local: they are allowed to reference variables from the local environment. For example, we can parameterise a *Semigroup*

v	::=	$a \mid v_1 \to v_2 \mid \forall\, a.\, v \mid TC.\texttt{Dict}\ v$	*Type*
t	::=	$x \mid \lambda(x : v).\, t \mid t_1\ t_2 \mid \Lambda a.\, t \mid t\ v$	*Term*
t_{ev}	::=	$d \mid t_{ev}\ \overline{v}\ \overline{t_{ev}}$	*Evidence term*
Γ_v	::=	$\epsilon \mid (x : v), \Gamma_v \mid a, \Gamma_v$	*Typing environment*
η	::=	$[\cdot] \mid [d \mapsto t_{ev}, \eta]$	*Dictionary evidence substitution*
d			*Dictionary evidence variables*
$fdv(\cdot)$			*Free dictionary variables*

Fig. 1. Syntax of the target language

dictionary over the modulo factor and the underlying operation (alternatively, the second parameter could be a *Semigroup* dictionary instead of a binary function):

```
modSemigroup :: Integer → (Integer → Integer → Integer)
             → Semigroup.Dict Integer
modSemigroup n op =
  Semigroup.Dict (λx y → (x `op` y) `mod` n)
```

Local instances were originally forbidden because they caused problems for coherence, although workarounds and restricted versions have been proposed in the past [7, 11].

3 Formalisation

To precisely describe explicit dictionary application, we use a formalisation of Haskell based on previous work [4, 18].

Besides type-checking rules, we also present *elaboration* rules, which detail the translation from the source language to the target language, System F. In practice, GHC's target language is GHC Core, but as our formalisation does not include local assumptions, kinding, type families, etc., we restrict ourselves to System F. During elaboration things implicit in the source language, e.g. the passing of dictionaries (dictionary translation), and type abstraction and application, are made explicit in the target language. We use the term *evidence* for dictionary instances in the target language.

We first present the simple target language, then the source language along with the elaboration from the latter to the former. Parts marked in red are only relevant for the elaboration and can be ignored until Section 3.4.

Both languages have the following syntax in common:

x, y, f	*Variables*
a, b	*Type variables (skolems)*
TC	*Type classes*

A type class (TC) does not include its type arguments, e.g., *Eq* is a TC, *Eq a* not. For simplicity, we only consider type classes with exactly one argument.

3.1 Target Language

Fig. 1 shows the mostly standard syntax of the target language, System F. The type $TC.\texttt{Dict}\ v$ is the type of the dictionary record corresponding to a type class $TC\ v$. Note that

τ	::=	$a \mid \tau_1 \to \tau_2 \mid TC.\texttt{Dict}\ \tau$	*Monotype*
ρ	::=	$C \Rightarrow \rho \mid \tau$	*Qualfied type*
σ	::=	$\forall a.\, \sigma \mid \rho$	*Type scheme*
C	::=	$TC\ \tau$	*Type-class constraint*
Q	::=	$\epsilon \mid Q_1 \wedge Q_2 \mid t_{ev} : C$	*Constraints*
A	::=	$\forall\, \overline{a}.\, \overline{C} \Rightarrow C$	*Axiom scheme*
$\boldsymbol{Q}$	::=	$\epsilon \mid \boldsymbol{Q}_1 \wedge \boldsymbol{Q}_2 \mid t_{ev} : A$	*Top-level axiom scheme*
e	::=	$x \mid \lambda x.\, e \mid e_1\ e_2 \mid e_1\ @\{e_2\ \texttt{as}\ TC\ a\}$	*Term*
Γ	::=	$\epsilon \mid (x : \sigma), \Gamma \mid a, \Gamma$	*Typing environment*
$ftv(\cdot)$			*Free type variables*

Fig. 2. Syntax of the source language

dictionary evidence variables (d) are also variables, we simply use a different letter for clarity. To explain evidence terms (t_{ev}), consider the following program in the source language:

```
f :: Eq a ⇒ Maybe Int → a → a → Bool
f mbN x y = mbN == Just 1 && x == y
```

This will be elaborated to the following program in the target language:

```
f = Λa. λ(d :: Eq.Dict a). λ(mbN :: Maybe Int).
      λ(x :: a). λ(y :: a).
        (&&)
          ((==) (Maybe Int) ($fEqMaybe Int $fEqInt)
                            mbN (Just Int 1))
          ((==) a d x y)
```

Note that type abstraction and application are now explicit, the types of binders are annotated, and dictionary *evidence* is abstracted over and applied. For example, the dictionary evidence variable d of type *Eq.Dict a* represents the evidence that *a* implements the *Eq* type class, i.e. the dictionary record containing the implementations of the methods. In the equality check of *mbN* and *Just* 1, the composed evidence term (*$fEqMaybe Int $fEqInt*), found by the constraint solver, is used, where *$fEqMaybe* and *$fEqInt* correspond to the evidence terms produced by the respective global instances:

```
instance Eq a ⇒ Eq (Maybe a) where ...
instance Eq Int where (==) = ...
```

Note that evidence terms are simply a subset of terms.

For brevity, we defer the standard typing rules of the target language to a companion technical report [20].

3.2 Source Language

Fig. 2 shows the syntax of the source language, based on [4, 18]. We omit parts that are not important with regards to explicit dictionary application: **case** expressions, **let** bindings, equality constraints, etc. The top-level axiom scheme ($\boldsymbol{Q}$) contains instance declarations, for example:

$$(\$fEqMaybe : \forall a.\, Eq\ a \Rightarrow Eq\ (Maybe\ a)) \wedge$$
$$(\$fEqInt : Eq\ Int)$$

$$\boxed{Q\,;\,\Gamma \vdash e : \sigma \rightsquigarrow t}$$

$$\frac{Q\,;\,a, \Gamma \vdash e : \sigma \rightsquigarrow t \qquad a \notin ftv(Q, \Gamma)}{Q\,;\,\Gamma \vdash e : \forall a.\, \sigma \rightsquigarrow \Lambda a.\, t}\ \forall\text{I} \qquad \frac{Q\,;\,\Gamma \vdash e : \forall a.\, \sigma \rightsquigarrow t \qquad \tau \overset{\tau}{\rightsquigarrow} v}{Q\,;\,\Gamma \vdash e : [a \mapsto \tau]\sigma \rightsquigarrow t\ v}\ \forall\text{E}$$

$$\frac{d : C \wedge Q\,;\,\Gamma \vdash e : \rho \rightsquigarrow t \qquad C \overset{C}{\rightsquigarrow} v \qquad d \notin fdv(Q)}{Q\,;\,\Gamma \vdash e : C \Rightarrow \rho \rightsquigarrow \lambda(d : v).\, t}\ \Rightarrow\text{I} \qquad \frac{Q\,;\,\Gamma \vdash e : C \Rightarrow \rho \rightsquigarrow t \qquad \mathcal{Q} \wedge Q \Vdash t_{ev} : C}{Q\,;\,\Gamma \vdash e : \rho \rightsquigarrow t\ t_{ev}}\ \Rightarrow\text{E}$$

$$\frac{(x : \sigma) \in \Gamma}{Q\,;\,\Gamma \vdash x : \sigma \rightsquigarrow x}\ \textsc{Var} \qquad \frac{Q\,;\,(x : \tau_1), \Gamma \vdash e : \tau_2 \rightsquigarrow t \qquad \tau_1 \overset{\tau}{\rightsquigarrow} v_1}{Q\,;\,\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2 \rightsquigarrow \lambda(x : v_1).\, t}\ \textsc{Abs} \qquad \frac{Q\,;\,\Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow t_1 \qquad Q\,;\,\Gamma \vdash e_2 : \tau_1 \rightsquigarrow t_2}{Q\,;\,\Gamma \vdash e_1\ e_2 : \tau_2 \rightsquigarrow t_1\ t_2}\ \textsc{App}$$

$$\frac{\begin{array}{c} Q\,;\,\Gamma \overset{spec}{\vdash} e_1 : \forall \overline{b_1}\,\overline{a}\,\overline{b_2}.\, (\overline{C_1}, TC\ a, \overline{C_2}) \Rightarrow \tau_1 \rightsquigarrow t_1 \qquad \tau_2 \overset{\tau}{\rightsquigarrow} v_2 \\ Q\,;\,\Gamma \vdash e_2 : TC.\texttt{Dict}\ \tau_2 \rightsquigarrow t_2 \qquad [a \mapsto \tau_2]\overline{C_1} \overset{C}{\rightsquigarrow} \overline{v_1} \end{array}}{Q\,;\,\Gamma \vdash e_1\ @\{e_2\ \texttt{as}\ TC\ a\} : \forall \overline{b_1}\,\overline{b_2}.\, [a \mapsto \tau_2]((\overline{C_1}, \overline{C_2}) \Rightarrow \tau_1) \rightsquigarrow \Lambda \overline{b_1}\,\overline{b_2}.\, \lambda(\overline{d : v_1}).\, t_1\ \overline{b_1}\ v_2\ \overline{b_2}\ \overline{d}\ t_2}\ \textsc{DictApp}$$

$$\boxed{Q\,;\,\Gamma \overset{spec}{\vdash} e : \sigma \rightsquigarrow t}$$

$$\frac{\begin{array}{c} Q\,;\,\Gamma \vdash e : \sigma \rightsquigarrow t \\ \text{The type of } e \text{ is specified to be } \sigma \\ \text{The principal type of } e \text{ is unambiguous} \end{array}}{Q\,;\,\Gamma \overset{spec}{\vdash} e : \sigma \rightsquigarrow t}\ \textsc{Spec}$$

$$\boxed{\mathcal{Q}\,;\,\Gamma \overset{top}{\vdash} e : \sigma \rightsquigarrow t}$$

$$\frac{\begin{array}{cc} Q\,;\,\Gamma \vdash e : \tau \rightsquigarrow t & \mathcal{Q} \wedge \overline{d : C} \Vdash Q\,;\,\eta \\ \overline{a} = ftv(\overline{C}, \tau) & \overline{C} \overset{C}{\rightsquigarrow} \overline{v} \end{array}}{\mathcal{Q}\,;\,\Gamma \overset{top}{\vdash} e : \forall \overline{a}.\, \overline{C} \Rightarrow \tau \rightsquigarrow \Lambda \overline{a}.\, \lambda(\overline{d : v}).\, \eta(t)}\ \textsc{Top}$$

Fig. 3. Declarative typing rules of the source language

Any class constraint C can be considered a degenerate axiom scheme A with $\overline{a}$ and $\overline{C}$ empty. Similarly, any Q can be considered a degenerate $\mathcal{Q}$. We use $C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_n \Rightarrow \tau$, $(C_1, C_2, \ldots, C_n) \Rightarrow \tau$, or $\overline{C} \Rightarrow \tau$ when convenient, all three forms mean the same.

3.3 Type Checking

The typing rules depend on the *constraint entailment* relation [18]: $\mathcal{Q} \Vdash Q$. This relation can be read as: "from the top-level axiom scheme $\mathcal{Q}$, we can derive the constraints Q." Following OutsideIn(X), we leave the details of entailment deliberately unspecified, because it is a parameter of the type system [18]. Compared to OutsideIn(X), we extend this relation to produce evidence for each entailment, i.e. the t_{ev} in $t_{ev} : C$, which we will use to elaborate typing judgements. For example:

$$(\$fEqMaybe : \forall a.\, Eq\ a \Rightarrow Eq\ (Maybe\ a)) \wedge$$
$$(\$fEqInt : Eq\ Int)$$
$$\Vdash \$fEqMaybe\ Int\ \$fEqInt : Eq\ (Maybe\ Int)$$

We now present the declarative typing rules of the source language in Fig. 3. The typing judgment can be read as: "under assumptions Q and in context Γ, term e has type σ." Except for the new DictApp rule, the rules are standard and based on [4]. We derive polytypes σ instead of just monotypes τ because the DictApp rule requires polytypes.

Let us now discuss the new DictApp rule. For simplicity, the annotation "**as** *TC a*" is mandatory in the formalisation, whereas it is optional in the implementation when there is only one constraint in the context matching the type of the dictionary. As discussed in Section 2.2, the type of e_1, to which the dictionary will be passed, must be specified. This is expressed using the $Q\,;\,\Gamma \overset{spec}{\vdash} e : \sigma \rightsquigarrow t$ judgment. This judgment also states that the principal type of e must be unambiguous, a requirement that is needed for coherence (even without explicit dictionary application), see Section 5.2.

The type class constraint $(TC\ a)$ to which a dictionary is passed may occur at any place in the type class context of e_1. The same is true for the corresponding type variable (a). This is captured by the zero or more constraints $\overline{C_1}$ and $\overline{C_2}$ coming before and after the type class constraint in question, and the zero or more type variables $\overline{b_1}$ and $\overline{b_2}$ coming before and after the type variable in question. The dictionary e_2 must have a type $(TC.\texttt{Dict}\ \tau_2)$, matching the type class of the constraint. After passing the dictionary, the type variable a is instantiated with τ_2 and the constraint in question is removed from the type class context. For simplicity, multiple explicit dictionary applications cannot be chained one after the other in the formalisation, but this is supported in the implementation.

Top-Level Typing The judgment for top-level terms can be read as "under top-level axiom scheme $\mathcal{Q}$ and in context Γ, term e has type σ." Compared to the regular typing judgment, we make sure no free type variables occur by quantifying over them. We require a monotype τ to be derived for e even though the judgment can derive a polytype σ. Using the rules $\forall$E and $\Rightarrow$E, every polytype can be instantiated to a monotype. Also, the top-level axiom scheme $(\mathcal{Q})$ is used to simplify the required constraints. For example, if e assumes $Eq\ (Maybe\ a)$ and $\mathcal{Q}$ contains the axiom $\forall a.\ Eq\ a \Rightarrow Eq\ (Maybe\ a)$, we want to qualify over the "extra information" needed to satisfy the assumption, i.e. $Eq\ a$, as:

$$(\$fEqMaybe : \forall a.\, Eq\ a \Rightarrow Eq\ (Maybe\ a)) \wedge (d : Eq\ a)$$
$$\Vdash d' : Eq\ (Maybe\ a)\,;\,[d' \mapsto \$fEqMaybe\ a\ d]$$

Algorithmic Typing Rules We omit algorithmic typing rules as they are standard [4, 18] and the algorithmic variant of DictApp can easily be derived from the declarative one.

3.4 Elaboration

We now turn to the matter of elaborating programs in the source language into programs in the target language. This translation is characterised by the typing rules in Fig. 3, marked in red. We use an additional judgment to elaborate types ($\tau \overset{\tau}{\rightsquigarrow} v$), constraints ($C \overset{C}{\rightsquigarrow} v$), ... from the source to the target language. For example, to elaborate a constraint to the target language, we have the following rule:

$$\frac{\tau \overset{\tau}{\rightsquigarrow} v}{TC\ \tau \overset{C}{\rightsquigarrow} TC.\mathtt{Dict}\ v}\ \textsc{Constraint}$$

As the judgment is defined by straightforward inductive rules, we omit it here, but it can be found in [20].

In Fig. 3, we see that the type of the binder is made explicit in Abs. Type abstractions and applications become explicit in $\forall$I and $\forall$E. Evidence abstractions and applications become explicit in $\Rightarrow$I and $\Rightarrow$E.

The elaboration in DictApp is more elaborate, as type variables and constraints must be rearranged. Consequently, type and evidence abstractions and applications must be added to align the resulting type with the resulting term, i.e. the type v_2 must be applied before $\overline{b_2}$ are applied and t_2 must be applied after the $\overline{d}$ corresponding with $\overline{C_1}$. We have η-reduced the evidence abstractions and applications corresponding with $\overline{C_2}$. The crux of the rule is that t_2, the term corresponding to the dictionary, is passed as an evidence argument to t_1.

In the Top rule, note the "; η" in the entailment, where η is a dictionary evidence substitution. Let us explain this with an example, say we have that:

$$(d' : Eq\ (Maybe\ a))\ ;\ ((x : Maybe\ a), a) \vdash (==)\ x\ x : Bool$$
$$\rightsquigarrow (==)\ (Maybe\ a)\ d'\ x\ x$$

When simplifying Q, in this case $(d' : Eq\ (Maybe\ a))$, as demonstrated at the end of Section 3.3, we get $\overline{d : C} = d : Eq\ a$. However, the elaborated term t still contains the dictionary variable d'. Therefore, we must substitute the original dictionary variable d' with the simplified evidence *$fEqMaybe a d*.

4 Global Uniqueness of Instances

A major reason why past proposals have not been adopted is that they broke existing code that relies on global uniqueness of instances. The standard example of this, and the most-heard counterargument against allowing multiple instances or explicit dictionary application, are the *Set* and *Map* abstract data types. These data types use ordered binary trees under the hood, and their implementation relies on the fact that the same *Ord* instance is passed for each operation manipulating them. Let us show in more detail how an explicit dictionary application can cause problems for the *Set* example from the introduction:

```
insert :: Ord a ⇒ a → Set a → Set a
empty :: Set a

reverseOrd :: Ord a ⇒ Ord.Dict a
reverseOrd = Ord.Dict {compare = flip compare}

> insert @{reverseOrd} 1 (insert 1 (insert 2 empty))
fromList [1, 2, 1]
```

We insert 2 and 1 into an empty set using the default instance of *Ord*, before we insert 1 again, but this time using a dictionary for *Ord* that reverses the ordering. The resulting set contains the element 1 twice, violating the invariant that each element occurs at most once.

In our proposal, we want to forbid the above explicit dictionary application *insert* @{*reverseOrd*}. A similar case that we want to allow is the case-insensitive *nub* we saw before:

```
nubCI :: [String] → [String]
nubCI = nub @{eqOn (map toLower)}
```

The *nub* method does not rely on global instance uniqueness, so this dictionary application is harmless.

Our proposal is to restrict explicit dictionary application using a criterion that distinguishes harmful cases from harmless ones. To understand how we make this distinction, it is useful to consider what we call the "newtype translation" of explicit dictionary application, a way to simulate the explicit dictionary translation using wrapper **newtype**s.

4.1 Newtype Translation

Let us first translate the safe *nubCI* example:

```
newtype W a = W a

instance Eq (W String) where
  (==) = coerce $ λw1 w2 →
    map toLower w1 == map toLower w2
```

We first define a **newtype** *W*, with the desired *Eq* instance. The desired equality for *String*s is converted to work on *W String*s using the function *coerce* [3]. This function converts between a **newtype** and its wrapped type without runtime cost, even when the newtype is nested in a bigger type, e.g., a *W String* → *W String* → *Bool* can be coerced from/to a *String* → *String* → *Bool*. Alternatively, this instance could be written as a dictionary instance:

```
instance Eq (W String) = coerce (eqOn (map toLower))
```

Now we can define *nubCI′*, which invokes *nub* at type *W String* so that the custom instance of *Eq* is used, and then coerces the resulting function of type [*W String*] → [*W String*] back to [*String*] → [*String*].

```
nubCI′ :: [String] → [String]
nubCI′ = coerce (nub @(W String))
```

This program type-checks and works as expected:

```
> nubCI' ["Foo", "foo", "bar", "bar"]
["Foo", "bar"]
```

Now let us try the same translation for the unsafe program:

newtype $W\ a = W\ a$ **deriving** *Eq*

instance $Ord\ a \Rightarrow Ord\ (W\ a)$ **where**
  $compare = coerce\ (flip\ (compare\ @a))$

Or, alternatively:

instance $Ord\ a \Rightarrow Ord\ (W\ a) = coerce\ (reverseOrd\ @a)$

Like before, we define *insertRev'* to call *insert* at type $W\ a$ with the custom *Ord* instance, and coerce the resulting function of type $W\ a \to Set\ (W\ a) \to Set\ (W\ a)$ back to a $a \to Set\ a \to Set\ a$:

$insertRev' :: \forall a.\ Ord\ a \Rightarrow a \to Set\ a \to Set\ a$
$insertRev' = coerce\ (insert\ @(W\ a))$

Interestingly, this program does not type-check! We get the following type error:

```
Couldn't match type 'a' with 'W a' arising
from a use of 'coerce'
```

The reason for this is that the coercion is not allowed by GHC because the type parameter of the *Set* data type has role Nominal. This means that a $Set\ a$ can only be coerced to a $Set\ b$ if a is nominally equal to b, i.e. they are the exact same type. The types a and $W\ a$ are only Representationally equal, as they have the same run-time representation, but are not the same exact type. In the *nubCI* example, the type $[String]$ can be coerced to $[W\ String]$, as the type parameter of the list data type has role Representational. This is why the *nubCI'* function type-checked.

Set and *Map* are mentioned in [3] as an example where library authors must assign the correct roles to the type arguments to prevent abusive coercions that produce invalid sets or maps. Our insight is that the problem we are facing, of distinguishing safe from unsafe cases of explicit dictionary application, is related to the problem faced by the authors of [3] (which motivated the introduction of roles in GHC). The fact that the newtype translation of the safe case type-checks and that the newtype translation of the unsafe case did not type-check, demonstrates that we can reduce our problem to the problem of safely coercing between (new)types and reuse their solution (roles) for constructing our criterion.

4.2 Role Criterion

Although this newtype translation could in principle be used by the compiler as an implementation technique for explicit dictionary application,[2] we prefer the much simpler solution of passing the custom dictionary as evidence at the level of GHC Core, as shown in Section 3.4. However, we use the newtype translation to construct a *role criterion* that will check safety without actually performing the newtype translation in practice.

Say we want to pass a dictionary for the type class constraint $TC\ a$ to a function with the following type:

$$\forall \overline{b_1} a \overline{b_2}.\ (\overline{C_1}, TC\ a, \overline{C_2}) \Rightarrow \tau_1$$

This is safe iff a has role $\rho \leq$ Representational (i.e. Phantom or Representational) in τ_1, in $TC.\texttt{Dict}\ a$ and in all constraints in $\overline{C_1} \wedge \overline{C_2}$ that mention a. Particularly, a may not occur in any equality constraints in $\overline{C_1} \wedge \overline{C_2}$ and their superclasses to prevent programs like the following:

$naughtyInsert :: (Ord\ a, a \sim b) \Rightarrow a \to b \to Set\ b \to Set\ b$
$naughtyInsert\ \_ = insert$

These conditions must be included in the DictApp rule. When *TC* has more than one type argument, it suffices that at least one of them satisfies these conditions. Note that the role of a in constraints mentioning a in $\overline{C_1} \wedge \overline{C_2}$ matters, because they need to be reimplemented for the wrapper type in the newtype translation, by coercing their dictionary. This means we rely on correct roles for arguments of these type class constraints. GHC currently infers roles of type class arguments conservatively to Nominal by default, which will unnecessarily prevent some valid explicit dictionary applications in cases with multiple constraints. This default should either be modified, or we could simply check the role of a in the corresponding dictionaries instead, where they are inferred less conservatively. The role of a in $TC.\texttt{Dict}\ a$ can be nominal if a occurs in a Nominal position in one of the class methods' types, but this is uncommon in practice.

Let us look at the two cases (in both cases the role in $TC.\texttt{Dict}\ a$ is Representational):

Case	τ_1	*Role of a in* τ_1	*Safe?*
nub	$[a] \to [a]$	Representational	✓
insert	$a \to Set\ a \to Set\ a$	Nominal	✗

Only type-variable arguments are allowed, as it is impossible to detect the role of a concrete type, e.g., what is the role of *Int* in $Ord\ Int \Rightarrow Int \to Int \to Set\ Int \to Set\ Int$? The implementation of this function might be $\lambda x\ y\ s \to$ **if** $x > y$ **then** s **else** *empty*,[3] which does not exhibit the same unsafety as the previous example. For this reason, combined with the issues regarding incoherence, discussed in Section 5, we disallow explicit dictionary when the argument of the type class is not a type variable.

To determine the role of a in τ_1, we use a simple algorithm based on the role inference algorithm described in [3], with as major simplification the fact that we can simply look at the roles of the arguments of type constructors instead of inferring them.

To show that the role criterion corresponds to the newtype translation, consider the steps of the translation and what

[2] Except for the fact that dictionaries referring to the local environment cannot be newtype translated.

[3] With $Ord\ a \Rightarrow a \to a \to Set\ b \to Set\ b$ as principal type.

could go wrong. After introducing a newtype, an instance declaration is generated based on the custom dictionary, and forwarding instances are generated for other constraints. The dictionaries for these instances are all coerced to the newtype, leading to the role requirement for a in their types. Also, these instances must not overlap. For example, passing a custom instance to the first $Eq\ a$ in $(Eq\ a, Eq\ a)$ would cause overlapping Eq instances. Such ambiguities are rejected by the coherence check, discussed in Section 5. Finally, the function is instantiated to the newtype (which cannot fail) and then coerced back, which works under the role requirement for a in τ_1.

5 Coherence

We use the definition from [14] for coherence: *every different valid typing derivation for a program leads to a resulting program that has the same dynamic semantics.* So how does this translate to our setting? Consider the following program:

$foo :: Eq\ Int \Rightarrow Bool$
$foo = 1 == 3$

There are two valid typing derivations for this program: one that uses the global instance of $Eq\ Int$ and one that uses the instance passed to foo. It does not matter which instance or dictionary was chosen, because with global uniqueness of instances, it is the same instance in both cases. With the ability to explicitly pass a dictionary, it suddenly does matter which typing derivation is used, because the different typing derivations can use potentially different dictionaries, which will directly influence the dynamic semantics. The following cases have a similar risk of incoherence, as the compiler can choose between multiple dictionaries:

-- Two $Eq\ a$ dictionaries...
$two :: (Eq\ a, Eq\ a) \Rightarrow a \to a \to Bool$
-- A second $Eq\ a$ hidden in $Ord\ a$...
$three :: (Eq\ a, Ord\ a) \Rightarrow a \to a \to Bool$
-- A second $Eq\ a$ dictionary thanks to constraint $a \sim b$
$four :: (Eq\ a, a \sim b, Eq\ b) \Rightarrow a \to b \to Bool$

As for foo above, global instances can create potential incoherence, also with type variables:

instance $Eq\ a$ **where** $\_ == \_ = False$
$five :: Eq\ a \Rightarrow a \to a \to Bool$

5.1 Detecting and Preventing Incoherence

All of the contrived programs above are valid and coherent in Haskell as long as global uniqueness of instances holds. However, when we add explicit dictionary application, they become incoherent: in each case, the compiler has more than one choice for which dictionary the functions will use, so we cannot allow one of those dictionaries to be instantiated to something else than the others. To preserve coherence, we restrict explicit type application using another safety criterion. In this section, we explain this criterion and how we have proven that it effectively salvages coherence.

So how do we detect cases like the above? Say we pass a dictionary for class constraint $TC\ a$ to a function with the following type:

$$\forall \overline{b_1} a \overline{b_2}.\ (\overline{C_1}, TC\ a, \overline{C_2}) \Rightarrow \tau_1$$

What all the examples had in common was that the type class constraint $TC\ a$ or any constraint that could be derived from it in combination with the top-level axiom scheme Q, could also be derived from the remaining constraints and the top-level axiom scheme $(Q \wedge \overline{C_1} \wedge \overline{C_2})$. As this is also trivially true for any constraint in Q, we also require that the constraint cannot be derived from Q while producing the same evidence. Since we allow explicit type application to non-top-level expressions, we have to add to this list the local assumptions (Q). Generally, an explicit type application can cause incoherence if:

$$\exists C.(Q \wedge TC\ a \Vdash t_{ev} : C) \wedge (Q \wedge Q \wedge \overline{C_1} \wedge \overline{C_2} \Vdash C) \wedge Q \not\Vdash t_{ev} : C$$

To forbid such cases and to safeguard coherence, we add the following condition to the DictApp rule:

$$\forall C.Q \wedge TC\ a \Vdash t_{ev} : C \Rightarrow ((Q \wedge Q \wedge \overline{C_1} \wedge \overline{C_2} \not\Vdash C) \vee Q \Vdash t_{ev} : C)$$

In other words: to ensure coherence, the type class constraint that we provide an explicit dictionary for, or any constraint implied by it and the global instances, is either not implied by the remaining constraints and the global instances, or it is implied by the global instances while yielding the exact same evidence.

All of the examples above are caught by this check. In practice, this check does not often fire, as programmers tend not to use such signatures. GHC even has two flags to detect and warn about many such cases: `-Wsimplifiable-class-constraints` which warns about class constraints in a type signature that can be simplified using a top-level instance declaration, and `-Wredundant-constraints`, which warns about redundant constraints in type signatures.

However, some superclass hierarchies have only recently been established: Functor-Applicative-Monad and Semigroup-Monoid. Programs that have to work with or without these superclass relations in place, will intentionally have redundant constraints in their signatures, e.g., both $Functor\ m$ and $Monad\ m$, even though $Monad\ m$ entails $Functor\ m$. Similarly, programs written to work with or without global instances that were later introduced, will also fail the check.

The case where $TC\ a$ is not implied by the remaining constraints, but one of its superclasses is, is less artificial. For example, consider a function

$go :: (MonadState\ s\ m, MonadWriter\ w\ m) \Rightarrow m\ ()$

In this case, our coherence criterion prevents instantiating either constraint because they share a common superclass *Monad m*. In this scenario, it would make sense to allow an explicit application of dictionary d, if we can be sure that the parent dictionary of d for *Monad m* is equal to the default one. This could be done by statically determining whether the two GHC Core expressions are equal, or by allowing a dictionary application with some form of partial dictionary not containing the parent dictionary. A simpler solution we have chosen in our prototype, is to let the coherence check generate a warning instead of an error. This defers the responsibility to the programmer to ensure coherence in cases where it cannot be statically determined, instead of forbidding it. This warning can easily be turned into an error using the `-Werror=incoherence` flag.

5.2 Proof

To ensure that our coherence criterion is safe, we prove coherence of our formalisation. That is: we prove that any two typing derivation for a program lead to the same dynamic semantics. More concretely, we show that the two derivations elaborate to target-language terms that are equivalent, according to an axiomatic target-language equivalence relation $t_1 \approx t_2$. This relation, based on [9], is described in more detail in [20].

Theorem 1 (Coherence). *Given typing derivations* $Q\,;\,\Gamma \vdash e : \sigma \rightsquigarrow t_1$ *and* $Q\,;\,\Gamma \vdash e : \sigma \rightsquigarrow t_2$*, if the principal type* σ_0 *of* e *is context-unambiguous, and the constraint solver produces canonical evidence, it must be that* $t_1 \approx t_2$.

As established in previous work [9], coherence in the presence of type classes requires that the principal type of e is unambiguous.[4] For space reasons, we cannot explain this restriction here, but refer to Jones [9] as the issue is not specific to our proposed extension.

In a companion technical report [20], we prove this result for our system. The proof is itself technically novel and simpler than previous proofs in the literature (particularly [9]), as it avoids the use of an algorithmic typing relation (instead simply assuming that principal typing holds) and is parametric in the constraint implication judgement (simply assuming that it produces canonical evidence when given canonical evidence for assumptions). The proof is essentially a big induction on the typing judgement, using a suitably chosen but quite complicated induction hypothesis. More details and the full proof can be found in [20].

It is worth noting that our proof relies on an assumption about constraint entailment. When an explicit dictionary is provided for constraint C^p to a function with other constraints $\overline{C}$, and the coherence criterion is satisfied, then we require the following result: For any constraint C, there exist a t'_{ev} and $\overline{C'}$ such that:

- $Q \wedge \overline{d' : C'} \wedge d : C^p \Vdash t'_{ev} : C$
- For all t_{ev}, if $Q \wedge \overline{C} \wedge d : C^p \Vdash t_{ev} : C$, then $t_{ev} \approx [\overline{d' \mapsto t''_{ev}}]t'_{ev}$ where $Q \wedge \overline{C} \Vdash \overline{t''_{ev} : C'}$

For every constraint C, there are constraints $\overline{C'}$ that represent the extra assumptions needed to obtain C when C^p is given. Any evidence of C that can be derived from top-level instances, $\overline{C}$ and C^p, must factor through the evidence that derives C from $\overline{C'}$ and C^p. We believe this assumption holds when both C and C^p are type classes that only have other type classes as ancestors, but we are less convinced if they use type families, etc. Nevertheless, we point out that it is not necessary to have this property for all C and C^p, but just for those for which we allow explicit dictionary applications, so we expect our criterion to be adaptable to such settings.

6 Implementation

We have implemented a proof of concept of our proposal within GHC. Our implementation successfully type-checks all examples shown in this paper. The code can be found at https://github.com/mrBliss/ghc.

GHC generates a dictionary record for each type class. Instead of exposing these existing dictionary data types using the proposed *.Dict* suffix, we generate new dictionary data types with exactly the same fields and the *.Dict* suffix. The main reason for choosing this path was to minimise the impact on the compiler. An impactful change we hereby managed to sidestep was the reconciliation of the different kinds of the dictionary data types: the existing dictionary data types have kind $\ldots \to$ *Constraint* whereas the newly exposed dictionary data types have kind $\ldots \to$ *Type*.

In line with the dictionary translation already performed by GHC, an explicit dictionary application is translated to a simple function application in GHC Core. To maintain well-typedness of GHC Core, a Coercion is used to cast the exposed dictionary data type (of kind *Type*) to the internal one (of kind *Constraint*) [3].

As expected, no changes had to be made to GHC Core.

The implementation is still in a proof of concept stage. The coherence criterion does not yet check the non-entailment for all constraints entailed by *TC a* but only for the class and its superclasses.

The newly exposed dictionary data types need an import/export policy. We have some principles in mind, but have not implemented them yet. First, to avoid having to update existing code to export dictionary types, dictionaries should be exported along with the type class, unless explicitly specified otherwise. Second, the user can import a dictionary type independently from the type class and vice versa.

For now, we have hijacked the following syntax for explicit dictionary application instead of modifying the parser to support the proposed syntax:

- e $((dict))$ instead of e `@`$\{dict\}$
- e $((dict :: C\ a))$ instead of e `@`$\{dict$ **as** $C\ a\}$

[4] The *context*-prefix is only relevant for the proof and follows directly when a type scheme is well-formed.

To avoid breaking existing code, the `-XDictionaryApplications` flag first has to be enabled.

We currently support dictionary instances, without a termination check, and we have not yet implemented instance dictionaries (see Section 2.3).

7 Interesting Applications

While there are many interesting use cases of our proposal, we have chosen a few to showcase in this section.

7.1 Retrofitting Superclasses

As mentioned before, the functionality of Deriving Via [1] can be replicated using dictionary instances, even without using the big hammer of explicit dictionary application.[5] We will not repeat all these examples, but present one here.

In the latest version of GHC, *Semigroup* has become a superclass of *Monoid*. Dictionary instances can be used to easily define an instance of the *Semigroup* class based on the existing *Monoid* instance:

```
semigroupFromMonoid :: Monoid a ⇒ Semigroup.Dict a
semigroupFromMonoid = Semigroup.Dict {(⋄) = mappend}

instance Monoid X where ...
instance Semigroup X = semigroupFromMonoid
```

Similarly, *Eq* instances can be derived from *Ord* instances, *Applicative* and *Functor* from *Monad*, etc.

7.2 Reference-Based *MonadState*

The *MonadState s m* monad class models the interface of a state monad. The *StateT* monad transformer is typically used as the implementation. In monad stacks that have *IO* or *ST* at the bottom, adding this extra *StateT* layer to the stack should in principle not be needed, because they already offer efficient mutable references that can be allocated as desired. However, without local instances, these references cannot be used directly to instantiate a *MonadState* constraint. For performance reasons, some larger programs (e.g., GHC and the Agda compiler) work around this limitation by adding a *ReaderT* layer to the monad stack that passes an *IORef* around, which is then used to resolve the *MonadState* constraint. This means that the monad stack must have *IO* or *ST* at the bottom.

Now that we have explicit application of local dictionaries, we can discharge a *MonadState* constraint using a locally created dictionary based on a mutable reference, without having to embed the reference and the *IO* monad in the monad stack:

```
eval :: (∀m. MonadState state m ⇒ m a) → state → IO a
eval m initState = do ref ← newIORef initState
                      m @{ioRefState ref}

ioRefState :: IORef state → MonadState.Dict state IO
ioRefState ref = MonadState.Dict
  {parent1 = getDict @(Monad IO)
  , get    = readIORef ref
  , put    = writeIORef ref}
```

This can even be done purely, without *IO*, by using an *STRef* instead of an *IORef*.

We believe this pattern might allow for significant performance improvements in large code bases that use abstract effect interfaces like *MonadState*. Examples of this are the Agda or PureScript compiler, but we do not yet have quantitative measurements. Initial micro-benchmarks of different ways to discharge a *MonadState* constraint suggest that the above pattern outperforms *StateT* consistently except when the underlying monad is *Identity*, presumably because of some optimisations kicking in.

8 Related Work

We categorise related work in two categories: previous proposals for related features in the context of Haskell, and work in other languages.

Previous Haskell Proposals The two main proposals for multiple instances and dictionary application in Haskell are Making Implicit Parameters Explicit [7] and Named Instances [10]. Ignoring syntax, our proposal resembles [7] most: dictionary records are exposed, there is an explicit dictionary application construct, and even a restricted form of dictionary instances. The authors claim coherence based on the fact that local instances take precedence over global ones, but there is little evidence that this does indeed suffice to guarantee coherence. In [10], *named instances* and *instances supply* are proposed to draw a parallel with the ML module system. In particular, type classes correspond to module signatures, and to allow multiple modules with the same instantiation of a module signature, one must allow multiple named instances, which share the module namespace.

We use a simple annotation to resolve ambiguity when passing a dictionary, e.g., to which *Eq* constraint should the dictionary be passed in $\forall a\ b.\ (Eq\ a, Eq\ b) \Rightarrow a \to b \to Bool$? In [10], this is solved by distinguishing *ordered* from *unordered* type class constraints: $\forall a\ b.\ Eq\ a \Rightarrow Eq\ b \Rightarrow a \to b \to Bool$ vs $\forall a\ b.\ \{Eq\ a, Eq\ b\} \Rightarrow a \to b \to String$. In [7], a different type system is used, that would infer the following type: $\forall a.\ Eq\ a \Rightarrow a \to \forall b.\ Eq\ b \Rightarrow b \to Bool$.

In [11], Kiselyov and Shan propose a form of local instances, restricted to constraints that mention a fresh local type variable. They first show that local instances restricted in this way can be implemented in terms of relatively standard Haskell features, albeit in a convoluted and inefficient way. The idea is to round-trip dictionaries through the type system. Subsequently, they propose a simpler direct implementation that avoids the detour through the type system,

[5] In fact, dictionary instances and instance dictionaries seem useful independently from dictionary application.

but retain the restriction to constraints mentioning fresh type variables, that the encoding suggests. This idea was implemented and fine-tuned in the reflection library.[6] Compared to our work, Kiselyov and Shan's idea also seems to ensure coherence and does not break code relying on global uniqueness of instances. However, the proposed criterion for allowing local instances is much more restrictive than ours, e.g. most of the examples we show cannot be directly instantiated. This restriction requires considerable boilerplate to work around in practical use: artificial types mentioning phantom type variables and forwarding instances, as well as (sometimes?) the unpopular `-XUndecidableInstances` extension, or some clever trickery to avoid it.

Other Languages Cochis, the Calculus of CoHerent ImplicitS, based upon the *implicit calculus* [13] and inspired by the proof-search technique called *focussing*, supports local scoping, overlapping rules, first-class and higher-order rules, while remaining type safe, coherent and unambiguous [15]. The authors of Cochis mention that addressing the problems with global uniqueness of instances is a future line of work, and are looking at the approach used in Genus [21] which tracks the types of instances to enforce their consistent use. This approach of parameterising the types by the used instances can also be applied in dependently typed languages. When Haskell gains support for fully dependent types, a redesign of the *Set* and *Map* data structures in the containers package would still be required in this approach, whereas our proposal is compatible with existing designs.

Other languages also support multiple instances and explicit instance application. Agda has *instance arguments* [6], implicit arguments that are solved by an instance resolution algorithm. Instances are declared as records and can also be explicitly passed. Coq has a similar type class system [17]. Idris' interface system [2] resembles Haskell's, but interfaces can be named and explicitly passed (using the same syntax we propose). Scala's implicits mechanism is more powerful as implicit arguments can be of any type. It can be used to model a Haskell-like type class system with local instances and explicit instance application [12]. The language $\mathcal{G}$ was intended to inspire C++'s concepts, which are quite close to Haskell type classes [16]. The work mentioned in this paragraph make no guarantees about global uniqueness of instances, and none come with coherence proofs.

9 Conclusion

Since the introduction of type classes in Haskell, there has been the restriction that only one instance can be defined per type to maintain coherence and to ensure global uniqueness of instances. Our proposal relaxes this restriction by exposing dictionary records and adding support for explicit dictionary application. What differentiates our proposal from past ones is that we have found a way to distinguish safe from unsafe explicit dictionary applications with respect to global uniqueness of instances, using GHC's roles mechanism. Moreover, we preserve coherence by detecting potential source of incoherence, and have formally proved coherence of our system. Based on the nature of our criteria, we expect a large majority of functions would satisfy them, although it remains future work to measure this quantitatively. Since it is backwards-compatible and does not require major new infrastructure in the compiler, we believe our proposal only needs some more bikeshedding to be adopted in GHC. If this happens, we have little doubt that explicit dictionary application, especially with local dictionaries, will find many applications, and enable important new design patterns.

[6] http://hackage.haskell.org/package/reflection

Acknowledgments

We would like to thank Tom Schrijvers for his thoughts on the coherence problem. This research is partially funded by the Research Fund KU Leuven and the Agency for Innovation by Science and Technology in Flanders (IWT). Dominique Devriese holds a postdoctoral fellowship of the Research Foundation - Flanders (FWO).

References

[1] Baldur Blöndal, Andres Löh, and Ryan Scott. 2018. Deriving Via: or, How to Turn Hand-Written Instances into an Anti-Pattern *(Haskell '18)*.
[2] Edwin Brady. 2018. The Idris Tutorial, Interfaces. http://docs.idris-lang.org/en/latest/tutorial/interfaces.html
[3] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2014. Safe Zero-cost Coercions for Haskell *(ICFP '14)*.
[4] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms *(ICFP '05)*.
[5] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated Types with Class *(POPL '05)*.
[6] Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda *(ICFP '11)*.
[7] Atze Dijkstra and Doaitse S. Swierstra. 2005. *Making Implicit Parameters Explicit.* Technical Report UU-CS-2005-032. Universiteit Utrecht.
[8] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application *(ESOP '16)*, Vol. 9632.
[9] Mark P. Jones. 1992. *Qualified Types: Theory and Practice.* Ph.D. Dissertation. Oxford University.
[10] Wolfram Kahl and Jan Scheffczyk. 2001. Named Instances for Haskell Type Classes *(Haskell '01)*.
[11] Oleg Kiselyov and Chung-chieh Shan. 2004. Functional pearl: implicit configurations–or, type classes reflect the values of types *(Haskell '04)*.
[12] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes As Objects and Implicits *(OOPSLA '10)*.
[13] Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. 2012. The Implicit Calculus: A New Foundation for Generic Programming *(PLDI '12)*.
[14] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space *(Haskell '97)*.
[15] Tom Schrijvers, Bruno C.d.S. Oliveira, and Philip Wadler. 2017. *Cochis: Deterministic and Coherent Implicits.* Technical Report 705. Department of Computer Science, KU Leuven.

[16] Jeremy G. Siek and Andrew Lumsdaine. 2011. A Language for Generic Programming in the Large. *Science of Computer Programming* 76, 5 (May 2011). Special Issue on Generative Programming and Component Engineering (Selected Papers from GPCE 2004/2005).

[17] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes *(TPHOLs '08)*.

[18] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular type inference with local assumptions. *J. Funct. Program.* 21, 4-5 (September 2011).

[19] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc *(POPL '89)*.

[20] Thomas Winant and Dominique Devriese. 2018. Coherent Explicit Dictionary Application for Haskell: Formalisation and Coherence Proof. *ArXiv e-prints* (July 2018). arXiv:1807.11267

[21] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. 2015. Lightweight, Flexible Object-oriented Generics *(PLDI '15)*.