Bidirectional Type Class Instances

Koen Pauwels
KU Leuven
Belgium
koen.pauwels@cs.kuleuven.be

Georgios Karachalias
KU Leuven
Belgium
gdkaracha@gmail.com

Michiel Derhaeg
Guardsquare
Belgium
michiel@derhaeg.be

Tom Schrijvers
KU Leuven
Belgium
tom.schrijvers@cs.kuleuven.be

Abstract

GADTs were introduced in Haskell's eco-system more than a decade ago, but their interaction with several mainstream features such as type classes and functional dependencies has a lot of room for improvement. More specifically, for some GADTs it can be surprisingly difficult to provide an instance for even the simplest of type classes.

In this paper we identify the source of this shortcoming and address it by introducing a conservative extension to Haskell's type classes: *Bidirectional Type Class Instances*. In essence, under our interpretation class instances correspond to logical bi-implications, in contrast to their traditional uni-directional interpretation.

We present a fully-fledged design of bidirectional instances, covering the specification of typing and elaboration into System FC, as well as an algorithm for type inference and elaboration. We provide a proof-of-concept implementation of our algorithm, and revisit the meta-theory of type classes in the presence of our extension.

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

Keywords Haskell, type classes, type inference, elaboration

ACM Reference Format:
Koen Pauwels, Georgios Karachalias, Michiel Derhaeg, and Tom Schrijvers. 2019. Bidirectional Type Class Instances. In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3331545.3342596

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Haskell '19, August 22–23, 2019, Berlin, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342596

1 Introduction

Type classes were first introduced by Wadler and Blott [1989] as a principled way to support ad-hoc polymorphism in Haskell, have since appeared in other declarative languages like Coq [The Coq development team 2004] and Mercury [Henderson et al. 1996], and have influenced the design of similar features (e.g., concepts for C++ [Gregor et al. 2006]).

One of the main reasons type classes have been so successful is that they support sound, decidable, and efficient type inference [Jones 1992], while being a simple extension of the well-understood Hindley-Damas-Milner system (HM) [Damas and Milner 1982; Hindley 1969]. Furthermore, as Wadler and Blott [1989] have shown, they can be straightforwardly translated to parametric polymorphism in intermediate languages akin to System F [Girard 1972; Reynolds 1974, 1983a].

Since the conception of type classes, instances have been interpreted as logical implications, due to their straightforward implementation as System F functions. For example, the well-known equality instance for lists

$$\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]$$

can be read as *"if a is an instance of Eq, then so is $[a]$"*. This interpretation has worked pretty well so far, but falls short in the presence of advanced features such as *Generalized Algebraic Data Types* (GADTs) [Peyton Jones et al. 2006].

More specifically, with the current interpretation of type classes a large class of GADTs cannot be made an instance of the simplest of type classes. In this work we alleviate this problem by introducing a conservative extension[1] to Haskell: *Bidirectional Instances*. Under our interpretation, instances like the above can be read as *"a is an instance of Eq* **if and only if** *$[a]$ is an instance of Eq"*.[2]

The main problem is that, unlike basic type classes, this extension requires a non-parametric encoding (which possibly explains its initial omission). We overcome this problem

[1] By "conservative", we mean that our system can type strictly more programs than plain Haskell does.
[2] Which we believe reflects what Haskell users already have in mind. Indeed, most prior research on type classes—such as the work of Sulzmann et al. [2007b]—treat $Eq\ a$ and $Eq\ [a]$ as denotationally equivalent.

with System F_C coercions [Sulzmann et al. 2007a] and add a non-parametric witness for the instance context to the otherwise parametric dictionary representation. Our specific contributions are:

- We present a detailed overview of the shortcomings in the interaction between GADTs and type classes, as well as other class-based extensions (Sec. 2).
- We identify the major challenges of interpreting and elaborating type class instances bidirectionally (Sec. 3).
- We describe an elaboration strategy that addresses all such challenges, while making Haskell strictly more expressive (Sec. 4).
- We formalize superclasses, an important—yet often neglected—aspect of type classes. Our formalization includes a specification of typing, elaboration, and an algorithm for type inference with elaboration (Sec. 5).
- We provide a formalization of typing and evidence translation from source terms to System F_C for type classes with bidirectional instances, as well as an algorithm for type inference with elaboration (Sec. 6). Our approach reuses most of the infrastructure needed by superclasses; existing systems need to be minimally extended for the additional expressive power.
- We elaborate on the changes bidirectional instances induce to the meta-theory of type classes; notably termination of type inference and principality of types (Sec. 7).
- We provide a prototype implementation of our algorithm for type inference with elaboration at https://github.com/KoenP/bidirectional-instances.

2 Motivation

2.1 Structural Induction Over Indexed Data Types

Ever since GADTs were introduced in Haskell [Peyton Jones et al. 2006], they have been put to good use by programmers for dataflow analysis and optimization [Ramsey et al. 2010], accelerated array processing,[3] automatic differentiation,[4] and much more. Yet, their interaction with existing features such as type classes [Wadler and Blott 1989] and functional dependencies [Jones 2000] can lead to surprising problems.

For example, consider (a simplified version of) the *Term* datatype, as given by Johann and Ghani [2008]:

```
data Term :: ⋆ → ⋆ where
  Con :: a → Term a
  Tup :: Term b → Term c → Term (b, c)
```

The GADT *Term* encodes a simple expression language, with constants (constructed by data constructor *Con*) and tuples (constructed by data constructor *Tup*).

[3] https://hackage.haskell.org/package/accelerate
[4] https://hackage.haskell.org/package/ad

Making (*Term a*) an instance of even the simplest of type classes can be challenging. For example, the following straightforward instance is not typeable under the current specification of type classes:

```
instance Show a ⇒ Show (Term a) where
  show (Con x)   = show x
  show (Tup x y) = unwords ["(", show x, ",", show y, ")"]
```

Loading the above program into `ghci` emits the following error message:

```
Bidirectional.hs:14:33:
    Could not deduce (Show b) arising from a use of `show'
    from the context (Show a) or from (a ~ (b, c))
Bidirectional.hs:14:44:
    Could not deduce (Show c) arising from a use of `show'
    from the context (Show a) or from (a ~ (b, c))
```

As the message indicates, the sources of the errors are the two recursive calls to *show* in the second clause: the instance context (*Show a*) and the local constraint (exposed via GADT pattern matching) $a \sim (b, c)$ are not sufficient to prove (*Show b*) and (*Show c*), which are required by the recursive calls to *show*. In summary, the type system cannot derive the following implications:

$$\forall b.\ \forall c.\ \mathit{Show}\ (b, c) \Rightarrow \mathit{Show}\ b$$
$$\forall b.\ \forall c.\ \mathit{Show}\ (b, c) \Rightarrow \mathit{Show}\ c$$

The Problem Both implications above constitute the inversion of the implication derived by the predefined *Show* instance for tuples:

```
instance (Show b, Show c) ⇒ Show (b, c) where { ... }
```

Indeed, the interpretation of type classes in existing systems is not *bidirectional*: the system can only derive *Show* (*b*, *c*) from *Show b* and *Show c*, but not the other way around.

2.2 Functional Dependencies and Associated Types

Unfortunately, the lack of bidirectionality of type class instances does not only affect the expressive power of simple type classes, but also the expressive power of features based on them, such as functional dependencies [Jones 2000] and associated type synonyms [Chakravarty et al. 2005a].

For example, let us consider an example of type-level programming using functional dependencies.[5] First, we define type-level natural numbers and length-indexed vectors:

```
data Nat :: ⋆ where
  Z :: Nat
  S :: Nat → Nat

data Vec :: Nat → ⋆ → ⋆ where
  VN :: Vec Z a
  VC :: a → Vec n a → Vec (S n) a
```

On the left, we define type-level natural numbers *Nat*. Type *Nat* is automatically promoted into a kind and data constructors *Z* and *S* into type constructors of the same name, using the GHC extension `DataKinds` [Yorgey et al. 2012].

[5] A similar example has been presented by Hallgren [2000], who implemented insertion sort at the level of types using functional dependencies.

Length-indexed vectors *Vec* utilize *Nat* to index data constructors *VN* and *VC* with the appropriate length: *VN* represents the empty vector (and thus has length *Z*), and *VC* represents concatenation (and thus constructs vectors of length (*S n*), where *n* is the length of the tail).

Equipped with type-level natural numbers, we can encode type-level addition (using the Peano [1889] axioms) by means of a multi-parameter type class and a functional dependency:

```
class Add (n :: Nat) (m :: Nat) (k :: Nat) | n m → k
instance                 Add Z     m m
instance Add n m k ⇒ Add (S n) m (S k)
```

Parameters *n* and *m* represent the operands, and parameter *k* represents the result, which is uniquely determined by the choice of *n* and *m*. The two Peano axioms for addition correspond to two instances for class *Add*, one for each form *n* can take.

The above can be combined to define function *append*, which concatenates two length-indexed vectors:

```
append :: Add n m k ⇒ Vec n a → Vec m a → Vec k a
append VN        ys = ys
append (VC x xs) ys = VC x (append xs ys)
```

The implementation of *append* is identical to the corresponding one for simple lists but its signature is much richer: *append* takes two vectors of lengths *n* and *m*, and computes a vector of length *k*, where $n + m = k$. Types like those above are extremely useful, for example in linear algebra libraries (see for example Hackage package `linear`), to ensure that operations respect the expected dimensions.

Unfortunately, examples like the one above are known not to type-check, mainly due to the lack of an evidence-based translation of functional dependencies. Yet, even with the recent advances of Karachalias and Schrijvers [2017], the above program is ill-typed.

Once again, the key missing element is bidirectional instances. In the second clause of *append*, the recursive invocation of *append* requires (*Add n′ m k′*), while the signature provides (*Add* (*S n′*) *m* (*S k′*)).[6] That is, we need the following implication:

$$\forall n'\ m\ k'.\ Add\ (S\ n')\ m\ (S\ k') \Rightarrow Add\ n'\ m\ k'$$

which can be obtained by interpreting the second *Add* instance bidirectionally.

As has been speculated by many and has recently been illustrated by Karachalias and Schrijvers [2017], associated type synonyms [Chakravarty et al. 2005a] share—for the most part—the same semantics with functional dependencies. Thus, the problem we are presenting here applies to associated type families as well; shortcomings of type classes affect all their extensions (indeed, rewriting the above example to use associated type synonyms instead of functional dependencies does not obviate the need for bidirectionality).

In summary, the lack of bidirectionality of type class instances severely reduces the expressive power of type class extensions, such as associated types [Chakravarty et al. 2005b], associated type synonyms [Chakravarty et al. 2005a], and functional dependencies [Jones 2000].

2.3 Summary

In summary, the lack of a bidirectional elaboration of class instances seriously undermines the interaction between GADTs and type classes, as well as type class extensions. This is precisely the issue we address in this paper.

3 Technical Challenges

Though bidirectional instances are sorely needed for applications involving GADTs, the problem is more general. For example, no Haskell compiler accepts programs where *Eq a* needs to be derived from *Eq* [*a*]. This is the case for the following type-annotated function: [7]

```
cmp :: Eq [a] ⇒ a → a → Bool
cmp x y = x == y
```

Though contrived, function *cmp* is a minimal example that exhibits all problems that arise in elaborating class instances bidirectionally in the well-established dictionary-passing translation [Hall et al. 1996]. Thus, we use it as our running example throughout the remainder of this section to discuss the technical challenges of interpreting type class instances bidirectionally.

3.1 Key Idea

Why Are Instances Bidirectional Existing systems with type classes ensure *coherence*[8] by disallowing instance heads to overlap. In turn, if no instance heads overlap, there can be at most one derivation for making a concrete type an instance of a certain type class. For example, given that instances are non-overlapping, the only way one can derive *Eq* [*Int*] is by combining the following two instances:

```
instance Eq Int
instance Eq a ⇒ Eq [a]
```

That being said, the only way one can create a dictionary of type *Eq* [*a*], for *any type a*, is by using the *Eq* instance for lists. Consequently, if a constraint *Eq* [*a*] is *given*, one can safely assume that *Eq a* is also available: modus ponens is invertible if there is no overlap in the implication heads.

[6] In fact, the signature provides (*Add n m k*), which we can refine using $n \sim S\ n'$ (obtained by GADT pattern matching), and the type-level function introduced by the functional dependency [Karachalias and Schrijvers 2017].

[7] The language used for our examples is equivalent to Haskell 98 plus the `FlexibleContexts` and `GADTs` extensions.

[8] Elaboration is said to be coherent if all valid typing derivations for a given program lead to target programs with the same dynamic semantics.

General Strategy In order to integrate bidirectionality in the system, we need to show how to derive the instance context from the instance head *constructively*. To achieve this, our strategy is simple: *reuse the approach of superclasses*.

According to the traditional dictionary-passing translation of type classes [Hall et al. 1996], superclass dictionaries are stored within subclass dictionaries. Hence, a superclass constraint (e.g., $Eq\ a$) can always be derived from a subclass constraint (e.g., $Ord\ a$), which is constructively reflected in a System F projection function. Thus, our key idea is to store the instance context within the class dictionary and retrieve it when necessary using System F projection functions.

This technique poses several technical challenges, which we elaborate on in the remainder of this section.

3.2 Challenge 1: Lack of Parametricity

Possibly the biggest challenge in interpreting class instances bidirectionally lies in the non-parametric dictionary representation. To explain what that means, let us consider the standard equality class Eq

$$\textbf{class}\ Eq\ a\ \textbf{where}\ \{\ (==) :: a \to a \to Bool\ \}$$

along with three instances:

$$\begin{array}{lll} \textbf{instance} & Eq\ Int & \textbf{where}\ \{\ (==) = \ldots\ \} \\ \textbf{instance}\ Eq\ b & \Rightarrow Eq\ [b] & \textbf{where}\ \{\ (==) = \ldots\ \} \\ \textbf{instance}\ (Eq\ c, Eq\ d) & \Rightarrow Eq\ (c,d) & \textbf{where}\ \{\ (==) = \ldots\ \} \end{array}$$

The instance context for each instance varies, depending on the instance parameter a. In the well-established dictionary-passing elaboration approach [Hall et al. 1996], these contexts correspond to the following System F types:

$$\begin{array}{lcl} a = Int & \Longrightarrow & Ctx\ a = () \\ a = [b] & \Longrightarrow & Ctx\ a = T_{Eq}\ b \\ a = (c,d) & \Longrightarrow & Ctx\ a = (T_{Eq}\ c, T_{Eq}\ d) \end{array}$$

where T_{Eq} is the System F type constructor for the class dictionary. Obviously, the System F representation of the instance context is not uniform but varies, depending on how we refine the class parameter a. This means that parametricity [Reynolds 1983b] as offered by System F is not sufficient for interpreting instances bidirectionally; a more powerful calculus is needed as our target language.

3.3 Challenge 2: Termination of Type Inference

The specification of typing is not affected much by bidirectional instances but this is not the case for type inference. Consider for example the inversion of the $Eq\ [b]$ instance:

$$\forall b.\ Eq\ [b] \Rightarrow Eq\ b$$

If such axioms are not used with care, the termination of the type inference algorithm is threatened. The standard backwards-chaining entailment algorithm [Kowalski 1974] cannot use such axioms to simplify goals and terminate. For example, we can "simplify" $Eq\ Int$ to $Eq\ [Int]$ using the above axiom. Not only is the size of the constraint bigger than the one we started with, but the axiom can also be applied infinitely many times (to capture that all nested list types are instances of Eq): the resolution tree now contains infinite paths. Thus, even a backtracking approach (such as the one used by Bottu et al. [2017]) cannot handle bidirectional instances in an obvious way: bidirectional axioms need to be used selectively to ensure the termination of type inference.

3.4 Challenge 3: Principality of Types

Finally, the introduction of bidirectional instances threatens the principality of types. In the absence of bidirectional instances, function cmp has a single most general type:

$$cmp :: Eq\ a \Rightarrow a \to a \to Bool$$

Constraint $Eq\ a$ can entail constraint $Eq\ [a]$ but not the other way around. In a system equipped with bidirectional instances, cmp can have multiple most general types. All the following types are equally general:

$$\begin{array}{ll} cmp :: Eq\ a & \Rightarrow a \to a \to Bool \\ cmp :: Eq\ [[a]] & \Rightarrow a \to a \to Bool \\ cmp :: Eq\ [Maybe\ [a]] & \Rightarrow a \to a \to Bool \end{array}$$

This makes specifying the correctness of type inference more difficult, as we should now infer one type from a set of equally general types.

In vanilla Haskell 98, only the first type annotation is well-formed. By using the `FlexibleContexts` extension, all three become well-formed type annotations, but they are not equivalent: the second and third annotations are implied by the first, but not the other way around; for the implementation of cmp as given at the start of Section 3 specifically, only the first annotation will type check. This annotation is also the only valid (and principal) type. With bidirectional instances, all three are acceptable types for cmp, and in fact, all three are *principal types*.

Although it may seem alarming that principal types are not unique in our extension, this is in fact not new. The HM system has the same issue, as well as its extension with qualified types [Jones 1995a]. For HM, principality of types is refined to take into account the possibilities for positioning universal quantifiers. Similarly, type classes exhibit the same problem in terms of the order of constraints, as well as by means of *simplification* [Jones 1995b].

In summary, in the presence of bidirectional instances a function can have infinitely many—equivalent to each other—principal types. This is not necessarily a problem but in order to ensure well-defined semantics for our extension, it is imperative that we revisit the notion of *type subsumption*, as well as the definition of the *principal type* property.

The next section describes our strategy for dealing with bidirectionality in intuitive terms; all formal aspects of our extension are described in Section 6.

4 Bidirectional Instances, Informally

In this section we describe our approach to interpreting type class instances bidirectionally, using as an example the elaboration of a variation of function *cmp* (Section 3):

$$cmp_2 :: Eq\ (b, c) \Rightarrow b \to b \to c \to c \to Bool$$
$$cmp_2\ x_1\ x_2\ y_1\ y_2 = (x_1 == x_2)\ \&\&\ (y_1 == y_2)$$

Though our formalization targets System F_C [Sulzmann et al. 2007a] (GHC's intermediate language), we avoid such formality here and we translate type classes to GHC-flavored Haskell dictionaries instead.

Dictionary Representation First, we show how to elaborate declarations. For example, we elaborate class *Eq*

$$\textbf{class}\ Eq\ a\ \textbf{where}\ \{\ (==) :: a \to a \to Bool\ \}$$

into the following declarations:

$$\textbf{type family}\ F_{Eq}\ a$$
$$\textbf{data}\ T_{Eq}\ a = K_{Eq}\ (F_{Eq}\ a)\ (a \to a \to Bool)$$
$$(==) :: T_{Eq}\ a \to (a \to a \to Bool)$$
$$(==)\ d = \textbf{case}\ d\ \textbf{of}\ \{\ K_{Eq}\ ctx\ eq \to eq\ \}$$

Traditionally, class declarations are elaborated into a dictionary type (T_{Eq}) and n functions, each corresponding to a class method. We extend this approach with an open type function F_{Eq} [Schrijvers et al. 2008], which is meant to capture the dependency between the class parameter and the instance context. The dictionary type is extended so that the instance context of type $F_{Eq}\ a$ is also stored.

The use of type families (and in general the choice of System F_C instead of plain System F) addresses the challenge of Section 3.2; System F_C offers native support for open, non-parametric type-level functions, which is exactly what we need, given (a) the non-parametric nature of bidirectionality, and (b) the open nature of type classes.

Inversion Functions Particularly interesting is the elaboration of class instances. Take for example the elaboration of the *Eq* instance for tuples

$$\textbf{instance}\ (Eq\ b, Eq\ c) \Rightarrow Eq\ (b, c)\ \textbf{where}\ \{\ eq = \ldots\ \}$$

which we elaborate into two kinds of declarations.

The first is a type family instance, mapping the class parameter (b, c) to the corresponding instance context representation $(T_{Eq}\ b, T_{Eq}\ c)$:

$$\textbf{type instance}\ F_{Eq}\ (b, c) = (T_{Eq}\ b, T_{Eq}\ c)$$

Its purpose is illustrated below. The next three (d_0, d_1, and d_2) are the *dictionary constructors* introduced by the instance. The first one—known as the *instance axiom*—captures the traditional meaning of the instance: *if* $Eq\ c$ *and* $Eq\ d$ *hold, then so does* $Eq\ (c, d)$:

$$d_0 :: T_{Eq}\ b \to T_{Eq}\ c \to T_{Eq}\ (b, c)$$
$$d_0\ d_b\ d_c = K_{Eq}\ (d_b, d_c)\ (\ldots)$$

The next two functions (or, better, dictionary constructors) witness the inversions of the instance axiom, so we refer to them as the *inverted instance axioms*:

$$d_1 :: T_{Eq}\ (b, c) \to T_{Eq}\ b$$
$$d_1\ (K_{Eq}\ ctx\ x) = \textbf{case}\ ctx\ \textbf{of}\ \{\ (d_b, d_c) \to d_b\ \}$$
$$d_2 :: T_{Eq}\ (b, c) \to T_{Eq}\ c$$
$$d_2\ (K_{Eq}\ ctx\ x) = \textbf{case}\ ctx\ \textbf{of}\ \{\ (d_b, d_c) \to d_c\ \}$$

d_0, d_1, and d_2 are witnesses of the following introduction and elimination rules, respectively:

$$\frac{Eq\ c \qquad Eq\ d}{Eq\ (c, d)} \qquad \frac{Eq\ (c, d)}{Eq\ c} \qquad \frac{Eq\ (c, d)}{Eq\ d}$$

The significance of the type family instance also becomes apparent in the definition of d_0, d_1, and d_2: to store (in the definition of d_0) and extract (in the definitions of d_1 and d_2) the instance context, we need to change its type from $(T_{Eq}\ b, T_{Eq}\ c)$ to $(F_{Eq}\ a)$, and vice versa. In source-level Haskell such conversions are implicit (like in the code above), but in System F_C, they are explicit (see Section 5.2). Our elaboration algorithm (Section 6) explains this translation in detail.

Finally, it is worth mentioning that the example already illustrates one of our design choices: instead of directly introducing a logical biconditional connective into our calculus, we simplify matters by generating the inversions as separate functions. This allows us to reuse existing infrastructure and the well-established dictionary-passing elaboration method.

Additional Derivations Finally, function cmp_2 is elaborated as follows:

$$cmp_2 :: T_{Eq}\ (b, c) \to b \to b \to c \to c \to Bool$$
$$cmp_2\ d\ x_1\ x_2\ y_1\ y_2 = \textbf{let}\ d_1' : T_{Eq}\ b = d_1\ d\ \textbf{in}$$
$$\textbf{let}\ d_2' : T_{Eq}\ c = d_2\ d\ \textbf{in}$$
$$((==)\ d_1'\ x_1\ x_2)\ \&\&\ ((==)\ d_2'\ y_1\ y_2)$$

The implementation of cmp_2 requires $Eq\ b$ and $Eq\ c$, but the signature provides $Eq\ (b, c)$. This is remedied by using the dictionary constructors d_1 and d_2 defined above to locally extract the needed information from the given dictionary d. As we illustrate below (Section 5.5.4), the creation of such a context might need several steps, but is guaranteed to terminate if the instances respect well-established termination conditions (Section 7.1). Hence, our approach also addresses the challenge described in Section 3.3.

5 Type Classes with Superclasses

Before we can present our formalization of bidirectional instances in Section 6, in this section we present a formalization of type classes with superclasses, including the specification of typing and elaboration to System F_C, as well as a type inference and elaboration algorithm.

This detour serves two purposes. Firstly, our extension reuses the infrastructure used by superclasses, so introducing superclasses first allows us to focus on the feature-specific

$$
\begin{array}{lll}
pgm ::= \overline{decl} & \tau ::= a \mid \tau_1 \to \tau_2 & S ::= \forall \overline{a}.\ C \Rightarrow Q \\
decl ::= cls \mid ins \mid val & \rho ::= \tau \mid Q \Rightarrow \rho & C ::= \bullet \mid C, Q \\
 & \sigma ::= \rho \mid \forall a.\ \sigma & Q ::= \mathsf{TC}\ \tau
\end{array}
$$

$$
\begin{array}{l}
e ::= x \mid \lambda x.\ e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
cls ::= \textbf{class}\ \forall a.\ C \Rightarrow \mathsf{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \} \\
ins ::= \textbf{instance}\ \forall \overline{b}.\ C \Rightarrow \mathsf{TC}\ \tau\ \textbf{where}\ \{\ f = e\ \} \\
val ::= x = e \mid x :: \sigma = e
\end{array}
$$

(a) Basic System: Syntax

$$
\begin{array}{lll}
\upsilon & ::= a \mid T \mid \upsilon_1\ \upsilon_2 \mid \forall a.\ \upsilon \mid F(\overline{\upsilon}) \mid \phi \Rightarrow \upsilon & \\
u & ::= a \mid T \mid u_1\ u_2 & \\
\phi & ::= \upsilon_1 \sim \upsilon_2 & \\
\gamma & ::= \langle \upsilon \rangle \mid \mathsf{sym}\ \gamma \mid \mathsf{left}\ \gamma \mid \mathsf{right}\ \gamma \mid \gamma_1 \fatsemi \gamma_2 \mid \phi \Rightarrow \gamma & \\
 & \quad \mid F(\overline{\gamma}) \mid \forall a.\ \gamma \mid \gamma_1[\gamma_2] \mid g\ \overline{\upsilon} \mid \omega \mid \gamma_1 @ \gamma_2 \mid \gamma_1\ \gamma_2 & \\
t & ::= x \mid K \mid \Lambda a.\ t \mid t\ \upsilon \mid \lambda(x : \upsilon).\ t \mid t_1\ t_2 \mid \Lambda(\omega : \phi).\ t & \\
 & \quad \mid t\ \gamma \mid t \triangleright \gamma \mid \textbf{case}\ t_1\ \textbf{of}\ \overline{p \to t_2} \mid \textbf{let}\ x : \upsilon = t_1\ \textbf{in}\ t_2 & \\
p & ::= K\ \overline{b}\ \overline{(\omega : \phi)}\ \overline{(x : \upsilon)} & \\
decl & ::= \textbf{data}\ T\ \overline{a}\ \textbf{where}\ \{\ \overline{K : \upsilon}\ \} \mid \textbf{type}\ F(\overline{a}) & \\
 & \quad \mid \textbf{axiom}\ g\ \overline{a} : F(\overline{u}) \sim \upsilon \mid \textbf{let}\ x : \upsilon = t &
\end{array}
$$

(b) System $\mathrm{F_C}$: Syntax

Figure 1. Source and Target Syntax

changes alone in the next section. Secondly, to our knowledge, we are the first to formalize type inference and elaboration of type classes in the presence of superclass constraints, so this section is a contribution in its own right (in particular Section 5.5).

The presentation of this section is deliberately technical, as it is aimed to serve as a specification for verification and implementation of our feature. Indeed, our prototype, which can be found at https://github.com/KoenP/bidirectional-instances, follows our specification closely.

The remainder of this section is structured as follows: Section 5.1 presents the syntax of source programs and Section 5.2 gives the syntax of System $\mathrm{F_C}$. The specification of typing and elaboration is given in Section 5.4, while Section 5.5 presents a type inference with elaboration algorithm. To simplify the presentation, throughout the whole section we highlight the parts of the rules that relate to elaboration.

A note on notation. From this section onwards, we will use overline notation to denote indexed sequences. For instance, when we write $\overline{x}^n$, we mean a sequence $x_1, x_2, ..., x_n$. Sometimes we omit the multiplicity superscript if the number of elements is of no interest (so $\overline{x}$ means $x_1, x_2, ..., x_n$ for some n). In some cases we use the overline notation on more complex structures than just variables, if we believe the meaning is clear from context (for instance, we might write $\lambda\ \overline{(x : d)}^n.\ t$ to mean $\lambda\ (x_1 : d_1)\ (x_2 : d_2)\ \ldots\ (x_n : d_n).\ t$).

5.1 Source Syntax

The syntax of the basic system is presented in Figure 1a. A program *pgm* consists of a list of declarations *decl*, which can be class declarations *cls*, instance declarations *ins*, or value bindings *val*. The syntax of class declarations, instances, and value bindings is standard. In order to reduce the notational burden, we omit all mention of kinds and assume that each class has exactly one method[9]. Additionally, we explicitly quantify over the type variables $\overline{a}$ that are bound in the class/instance head and context. Expressions comprise a λ-calculus, extended with let bindings. Types are stratified in monotypes τ, qualified types ρ, and polytypes σ. This is standard practice for HM extended with qualified types [Jones 1992]. Next, the syntax of constraints is straightforward: constraint schemes S capture implications generated by class and instance declarations. Sets of constraints (like superclass constraints or instance contexts) are denoted by C and single class constraints are denoted by Q.

5.2 Target Syntax

The syntax of System $\mathrm{F_C}$ programs is presented in Figure 1b. In contrast to prior specifications of type classes that use System F as the target language for elaboration, our elaboration targets System $\mathrm{F_C}$. Though for plain type classes this is not required (System F is a strict subset of System $\mathrm{F_C}$), it is essential for bidirectional instances, as we explained in Section 3.2.

Types υ include all System F types, extended with type family applications $F(\overline{\upsilon})$, and qualified types $(\phi \Rightarrow \upsilon)$. Qualified types classify terms that use coercion abstraction. Type patterns u are—as expected—the predicative subset of types.

Next, Figure 1b presents proposition types ϕ, capturing equalities between types. In the same way that types classify terms, propositions classify coercions γ; a coercion is nothing more than a proof of type equality and can take any of the following forms:

Reflexivity $\langle \upsilon \rangle$, symmetry ($\mathsf{sym}\ \gamma$) and transitivity ($\gamma_1 \fatsemi \gamma_2$) express that type equality is an equivalence relation. Forms $F(\overline{\gamma})$ and $(\gamma_1\ \gamma_2)$ capture injection, while ($\mathsf{left}\ \gamma$) and ($\mathsf{right}\ \gamma$) capture projection, which follows from the injectivity of type application. Equality for universally quantified and qualified types is witnessed by forms $\forall a.\ \gamma$ and $\phi \Rightarrow \gamma$, respectively. Similarly, forms $\gamma_1[\gamma_2]$ and $\gamma_1 @ \gamma_2$ witness the equality of type instantiation or coercion application, respectively.

Additionally, System $\mathrm{F_C}$ introduces two new symbol classes: coercion variables ω and axiom names g. The former represent local constraints and are introduced by explicit coercion abstraction or GADT pattern matching. The latter constitute the axiomatic part of the theory, and are generated from top-level axioms, which correspond to type family instances or newtype declarations [Peyton Jones 2003]. As we illustrated in the previous section, our bidirectional interpretation of class instances also gives rises to such axioms.

The semantics of the coercion forms we gave above is formally captured in coercion typing $\Gamma \vdash_{\mathrm{co}} \gamma : \phi$, which can

[9] Adding multiple methods would only increase verbosity without significant gains, since we would only have to add (many) overbars to the typing rules.

be found in the appendix of the extended version of this article (available at https://arxiv.org/abs/1906.12242).

System F_C terms t also conservatively extend System F terms. The interesting new forms are coercion abstraction $(\Lambda(\omega : \phi).\ t)$, coercion application $(t\ \gamma)$, and type casts $(t \triangleright \gamma)$. In simple terms, if γ is a proof that v_1 is equal to v_2 and t has type v_1, then $(t \triangleright \gamma)$ has type v_2. Patterns p capture existential variables $\overline{b}$ and local equality constraints $(\overline{\omega : \phi})$ in addition to term variables $\overline{x}$, to account for GADTs.

Programs consist of declarations *decl*, which consist of datatype declarations, type family declarations, type equality axioms, and variable bindings.

5.3 Additional Constructs

In order to state the specification of typing and elaboration succinctly, we first introduce some additional notation. First, we introduce typing environments and program theories:

$$\begin{array}{lr} \Gamma ::= \bullet \mid \Gamma, a \mid \Gamma, x : \sigma & \textit{typing environment} \\ P ::= \langle \mathcal{A}_S, \mathcal{A}_I, C_L \rangle & \textit{program theory} \end{array}$$

Typing environments are standard. The program theory P contains schemes generated by class and instance declarations, and gets extended with *local constraints*, when going under a qualified type. We explicitly represent the program theory as a triple of the superclass axioms $\mathcal{A}_S$, the instance axioms $\mathcal{A}_I$, and the local axioms C_L. We use the notation $P,_{\text{L}}\ d : Q$ to denote that we extend the local component of the triple, and similar notation for the other components.

Note that the specification we present below treats P as a conflated constraint set (that is, if $P = \langle \mathcal{A}_S, \mathcal{A}_I, C_L \rangle$ we write $(d : S) \in P$ to mean $(d : S) \in \mathcal{A}_S \cup \mathcal{A}_I \cup C_L$), while the inference algorithm we present in Section 5.5 distinguishes between the subsets; such a formalization is closer to actual implementations of type classes.

Finally, we define evidence-annotated axiom sets $\mathcal{A}$, local axioms C, and constraints Q:

$$\begin{array}{lr} \mathcal{A} ::= \bullet \mid \mathcal{A},\ d : S & \textit{variable-annotated axiom set} \\ C ::= \bullet \mid C,\ d : Q & \textit{variable-annotated constraint set} \\ Q ::= d : Q & \textit{variable-annotated class constraint} \end{array}$$

This notation allows us to present typing and elaboration succinctly below.

5.4 Specification of Typing and Elaboration

5.4.1 Term, Type, and Constraint Typing

Since most of the specification of typing for our core calculus can be found in prior work (see for example the work of Karachalias and Schrijvers [2017]), we omit the definitions for term typing, type well-formedness, and constraint well-formedness from our main presentation; they can be found in the extended version of this article. Their signatures are the following:

$$\begin{array}{lr} \Gamma \vdash_{\text{TY}} \sigma \rightsquigarrow v & \textit{type well-formedness} \\ \Gamma \vdash_{\text{CT}} Q \rightsquigarrow v & \textit{constraint well-formedness} \\ P; \Gamma \vdash_{\text{TM}} e : \sigma \rightsquigarrow t & \textit{term typing} \end{array}$$

Type well-formedness ensures that σ is well-formed under typing environment Γ and elaborates it into System F_C type v. Constraint well-formedness ensures that the type class constraint Q is well-formed under typing environment Γ and elaborates it into System F_C dictionary type v. The term typing relation ensures that e has type σ under typing environment Γ and program theory P, and elaborates e into System F_C term t.

We focus here on the more relevant aspects of the specification: constraint entailment and declaration typing.

5.4.2 Constraint Entailment

The notion of constraint entailment refers to the resolution of *wanted* constraints, arising from calling overloaded functions, using *given* constraints, provided by type signatures or GADT pattern matching [Vytiniotis et al. 2011]. This procedure is captured by relation $P; \Gamma \models t : Q$, read as *"under given constraints P and typing environment Γ, System F_C term t is a proof for constraint Q"*. [10] It is given by a single rule:

$$\frac{(d : \forall \overline{a}^n.\ \overline{Q}^m \Rightarrow \mathsf{TC}\ \tau) \in P \qquad \text{for each } \tau_i \in \overline{\tau}^n.\ \Gamma \vdash_{\text{TY}} \tau_i \rightsquigarrow v_i \qquad \text{for each } Q_i \in \overline{Q}.\ P; \Gamma \models t_i : [\overline{\tau}/\overline{a}]Q_i}{P; \Gamma \models d\ \overline{v}^n\ \overline{t}^m : [\overline{\tau}^n/\overline{a}^n](\mathsf{TC}\ \tau)}$$

This method of entailment—known as *Selective Linear Definite (SLD) clause resolution* [Kowalski 1974] or *backwards chaining*—is the standard sound and complete resolution for Horn clauses. Essentially, we match the head of a given Horn clause in the program theory P with the goal, and recursively resolve the premises of the clause. Dictionary construction behaves accordingly: the selected dictionary transformer d is instantiated appropriately (i.e., applied to types $\overline{v}^n$), and then applied to the proofs for the premises, $\overline{t}^m$.

5.4.3 Declaration Typing

The specification of typing with elaboration of declarations is presented in Figure 2.[10] We do not clutter the rules with freshness conditions by adopting the Barendregt [1981] convention.

Class Declarations Judgment $\Gamma \vdash_{\text{CLS}} cls : P_S; \Gamma_c \rightsquigarrow \overline{decl}$ handles class declarations and is given by Rule Cls. Apart from checking the well-scopedness of the class context and the method signature, it also gives rise to typing environment extension Γ_c which captures the method type, and program theory extension P_S which captures the superclass axioms. All this information is also captured in the generated declarations $\overline{decl}$, which includes the dictionary type declaration

[10] To aid readability, we highlight all aspects of the rules that are concerned with elaboration.

$\boxed{\Gamma \vdash_{\text{CLS}} cls : P_S; \Gamma_c \rightsquigarrow \overline{decl}}$ Class Declaration Typing

$$\frac{\begin{array}{c}\text{for each } Q_i \in \overline{Q}^n.\ \Gamma, a \vdash_{\text{CT}} Q_i \rightsquigarrow \upsilon_i \qquad \Gamma, a \vdash_{\text{TY}} \sigma \rightsquigarrow \upsilon \qquad P_S = [\text{for each } Q_i \in \overline{Q}^n.\ d_i : \forall a.\ \text{TC}\ a \Rightarrow Q_i] \qquad \Gamma_c = [f : \forall a.\ \text{TC}\ a \Rightarrow \sigma] \\ decl_M = \textbf{let}\ f : \forall a.\ T_{\text{TC}}\ a \to \upsilon = \Lambda a.\ \lambda(d : T_{\text{TC}}\ a).\ proj^{n+1}_{\text{TC}}(d) \\ decls_d = \text{for each } i \in [1..n].\ \textbf{let}\ d_i : \forall a.\ T_{\text{TC}}\ a \to \upsilon_i = \Lambda a.\ \lambda(d : T_{\text{TC}}\ a).\ proj^{i}_{\text{TC}}(d)\end{array}}{\Gamma \vdash_{\text{CLS}} \textbf{class}\ \forall a.\ \overline{Q}^n \Rightarrow \text{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \} : P_S; \Gamma_c \rightsquigarrow [\textbf{data}\ T_{\text{TC}}\ a = K_{\text{TC}}\ \overline{\upsilon}\ \upsilon, decl_M, decls_d]}\ \textsc{Cls}$$

$\boxed{P; \Gamma \vdash_{\text{INS}} ins : P_i \rightsquigarrow decl}$ Class Instance Typing

$$\frac{\begin{array}{c}\textbf{class}\ \forall a.\ \overline{Q^{\text{sup}}}^n \Rightarrow \text{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \} \qquad \Gamma_I = \Gamma, \overline{b} \qquad P_I = P, \text{for each } Q_i \in \overline{Q}^m.\ d_i : Q_i \qquad \text{for each } Q_i \in \overline{Q}^m.\ \Gamma_I \vdash_{\text{CT}} Q_i \rightsquigarrow \upsilon_i \\ \Gamma_I \vdash_{\text{TY}} \tau \rightsquigarrow \upsilon_0 \qquad S = \forall \overline{b}.\ \overline{Q}^m \Rightarrow \text{TC}\ \tau \qquad \text{for each } Q_i^{\text{sup}} \in \overline{Q^{\text{sup}}}^n.\ P_I; \Gamma_I \models t_i : [\tau/a]Q_i^{\text{sup}} \qquad P_{I,\text{I}}\ d : S; \Gamma_I \vdash_{\text{TM}} e : [\tau/a]\sigma \rightsquigarrow t\end{array}}{P; \Gamma \vdash_{\text{INS}} \textbf{instance}\ \forall \overline{b}.\ \overline{Q}^m \Rightarrow \text{TC}\ \tau\ \textbf{where}\ \{\ f = e\ \} : [\ d : S] \rightsquigarrow \textbf{let}\ d : \forall \overline{b}.\ \overline{\upsilon}^m \to T_{\text{TC}}\ \upsilon_0 = \Lambda \overline{b}.\ \lambda(\overline{d : \upsilon}^m).\ K_{\text{TC}}\ \upsilon\ \overline{t}^n\ t}\ \textsc{Ins}$$

$\boxed{P; \Gamma \vdash_{\forall \text{AL}} val : \Gamma' \rightsquigarrow decl}$ Value Binding Typing

$$\frac{\Gamma \vdash_{\text{TY}} \sigma \rightsquigarrow \upsilon \qquad P; \Gamma, x : \sigma \vdash_{\text{TM}} e : \sigma \rightsquigarrow t}{P; \Gamma \vdash_{\forall \text{AL}} x :: \sigma = e : [x : \sigma] \rightsquigarrow \textbf{let}\ x : \upsilon = t}\ \textsc{Sig}$$

$$\frac{\begin{array}{c}\text{for each } Q_i \in \overline{Q}.\ \Gamma, \overline{a} \vdash_{\text{CT}} Q_i \rightsquigarrow \upsilon_i \qquad \Gamma, \overline{a} \vdash_{\text{CT}} \tau \rightsquigarrow \upsilon \qquad P, \overline{d : Q}; \Gamma, \overline{a}, x : \tau \vdash_{\text{TM}} e : \tau \rightsquigarrow t \\ decl = \textbf{let}\ x : \forall \overline{a}.\ \overline{\upsilon} \to \upsilon = \Lambda \overline{a}.\ \lambda(\overline{d : \upsilon}).\ [x\ \overline{a}\ \overline{d}/x]t\end{array}}{P; \Gamma \vdash_{\forall \text{AL}} x = e : [x : \forall \overline{a}.\ \overline{Q} \Rightarrow \tau] \rightsquigarrow decl}\ \textsc{Val}$$

Figure 2. Basic System: Declaration Typing and Elaboration into System F_C

T_{TC}, the superclass axioms $\overline{d}^n$, and the method f. We use $proj^i_{\text{TC}}(d)$ to denote the extraction of the i-th field of a class dictionary d of type $T_{\text{TC}}\ a$:

$$proj^i_{\text{TC}}(d) \equiv \textbf{case}\ d\ \textbf{of}\ \{\ K_{\text{TC}}\ \overline{x} \to x_i\ \}$$

Instance Declarations Judgment $P; \Gamma \vdash_{\text{INS}} ins : P_i \rightsquigarrow decl$ handles instance declarations and is also given by a single rule. Rule Ins is for the most part straightforward: we ensure that all objects are well-scoped, and additionally check (a) the entailment of superclass constraints (denoted by $\overline{Q^{\text{sup}}}^n$) and (b) the method implementation against its expected type. Lastly, the program theory extension induced by the instance is captured in the unidirectional scheme S, which is also elaborated into System F_C dictionary transformer d.

Value Bindings In order to showcase the usability of bidirectional instances, we have included in our source language both annotation-free and type-annotated value bindings.

Rule Val deals with the former, while Rule Sig deals with the latter. Since type inference is undecidable in the presence of polymorphic recursion without type-annotations [Henglein 1993], Rule Val ensures that x is used monomorphically in recursive positions.[11] Apart from that, both rules are straightforward.

[11] Like all Haskell systems, our specification and inference algorithm allow polymorphic recursion in the presence of explicit type annotations (i.e., in class method implementations and annotated top-level bindings).

5.5 Type Inference with Elaboration

Now, we give an algorithm for type inference with elaboration. As is standard practice for HM-based systems, the algorithm proceeds in two phases: constraint generation and constraint solving.

5.5.1 Intermediate Constructs

First, we introduce three intermediate constructs: sets of equality constraints E, type substitutions θ, and evidence substitutions η:

E	$::= \bullet \mid E, \tau_1 \sim \tau_2$	*type equalities*
θ	$::= \bullet \mid \theta \cdot [\tau/a]$	*type substitution*
η	$::= \bullet \mid \eta \cdot [t/d]$	*evidence substitution*

Type equalities E are generated from the source text (alongside wanted class constraints C). Type and evidence substitutions are the results of constraint solving: the former maps unification variables to types, and the latter maps dictionary variables to dictionaries.

5.5.2 Elaboration of Terms, Types, and Constraints

Elaboration of terms, types, and constraints for our core calculus is also standard and can be found in prior work (see for example the work of Bottu et al. [2017]). The signatures of the corresponding judgments are the following:

$\Gamma \vdash_{\text{TM}} e : \tau \rightsquigarrow t \mid C; E$	*term elaboration*
$elab_{\text{TY}}(\sigma) = \upsilon$	*type elaboration*
$elab_{\text{CT}}(Q) = \upsilon$	*constraint elaboration*

Given a typing environment Γ and a source expression e, constraint generation infers a monotype τ for e and generates wanted constraints C and E, while at the same time elaborates e into a System F_C term t.

Elaboration of types and constraints is straightforward: the former elaborates a source type σ into a System F_C type v, and the latter transforms a class constraint Q into a System F_C (dictionary) type v.

Since all three are straightforward, we omit their definition; they can be found in the extended version of this article.

5.5.3 Constraint Solving

The type class and equality constraints derived from terms are solved with the following two algorithms.

Solving Equality Constraints We solve a set of equality constraints E using the standard first-order unification algorithm [Damas and Milner 1982]. Function $unify(\overline{a}; E) = \theta_\perp$ takes a set of equalities E and, if successful, produces as a result a type substitution θ. The additional argument $\overline{a}$ captures the "untouchable" variables introduced by type signatures, that is, variables that cannot be substituted (they can be unified with themselves though). Since its definition is straightforward, we omit its formal definition; it can be found in the extended version of this article.

Solving Class Constraints The judgment for solving class constraints takes the form $\overline{a}; \mathcal{A} \models C_1 \rightsquigarrow C_2; \eta$ and is given by the following rules:

$$\frac{\nexists Q \in C : \overline{a}; \mathcal{A} \models Q \rightsquigarrow C'; \eta}{\overline{a}; \mathcal{A} \models C \rightsquigarrow C\,; \bullet} \qquad \frac{\overline{a}; \mathcal{A} \models Q \rightsquigarrow C_1\,; \eta_1 \quad \overline{a}; \mathcal{A} \models C, C_1 \rightsquigarrow C_2\,; \eta_2}{\overline{a}; \mathcal{A} \models C, Q \rightsquigarrow C_2\,; (\eta_2 \cdot \eta_1)}$$

Given a set of untouchable type variables $\overline{a}$ and an axiom set $\mathcal{A}$, it (exhaustively) replaces a set of constraints C_1 with a set of *simpler* constraints C_2. This simplification it achieves via judgment $\overline{a}; \mathcal{A} \models Q \rightsquigarrow C; \eta$, given by a single rule:

$$\frac{(\,d_I : \forall \overline{b}.\ \overline{Q}^n \Rightarrow \mathtt{TC}\ \tau_2) \in \mathcal{A} \qquad unify(\overline{a}; \tau_1 \sim \tau_2) = \theta \qquad C = \text{for each } Q_i \in \overline{Q}^n.\ \ d_i : \theta(Q_i)}{\overline{a}; \mathcal{A} \models d : \mathtt{TC}\ \tau_1 \rightsquigarrow C\,; [d_I\ \theta(\overline{b})\ \overline{d}^n / d]}\ \textsc{Ent}$$

This form differs from the specification we gave in Section 5.4.2 in three ways.

First, we allow constraints to be partially entailed, which allows for *simplification* [Jones 1995b] of top-level signatures. This is standard practice in Haskell when inferring types. For instance, when inferring the signature for $(f\ x = [x] == [x])$. Haskell simplifies the derived constraint $Eq\ [a]$ to $Eq\ a$, yielding the signature $\forall a.\ Eq\ a \Rightarrow a \to Bool$.

Second, evidence construction is not performed directly, by means of creating a dictionary. Instead, a dictionary substitution η is created, which maps wanted dictionary variables to dictionaries. This strategy is analogous to the unification algorithm, which solves type equalities by creating a type substitution for instantiating the yet-unknown types.

Finally, algorithmic constraint entailment does not take the complete program theory as the specification does, but an axiom set. We make this design choice due to superclass constraint schemes: during simplification we do not want to replace a wanted constraint $(Eq\ a)$ with a more complex $(Ord\ a)$. We elaborate on the transition from the program theory P to an equally expressive axiom set $\mathcal{A}$—which does not contain superclass constraint schemes—next.

5.5.4 Transitive Closure of the Superclass Relation

Superclass axioms often overlap with instance axioms. Consider for example the following two axioms, the first obtained by the *Eq* instance for lists and the second obtained by the *Ord* class declaration:

$$\forall a.\ Eq\ a \Rightarrow Eq\ [a] \qquad \text{(a)}$$
$$\forall b.\ Ord\ b \Rightarrow Eq\ b \qquad \text{(b)}$$

This is a problem for type inference, since the constraint solving algorithm would have to make a choice when faced for example with constraint $Eq\ [c]$. Both (a) and (b) match but to completely entail constraint $Eq\ [c]$ we would require $Eq\ c$ if we were to choose the former and $Ord\ [c]$ if we were to choose the latter. In order to avoid this source of nondeterminism, several implementations of type classes (and notably GHC) treat superclass constraints differently.

In essence, we can pre-compute the transitive closure of the superclass relation on a set of given constraints and omit superclass axioms altogether. This procedure should also be reflected in the elaborated terms. To this end, we introduce dictionary contexts $\mathbb{E}$:

$$\mathbb{E} ::= \square \mid \textbf{let}\ d : v = t\ \textbf{in}\ \mathbb{E} \qquad \textit{dictionary context}$$

During entailment we can replace the program theory P with an axiom set $\mathcal{A}$ which does not contain any superclass axioms and a dictionary context $\mathbb{E}$. This procedure we denote as $\textsc{ScClosure}(\overline{a}, P) = (\mathcal{A}, \mathbb{E})$:

$$\textsc{ScClosure}(\overline{a}, \langle \mathcal{A}_S, \mathcal{A}_I, C_L \rangle) = ((\mathcal{A}_I, C'_L, C_L), \mathbb{E})$$
$$\text{where } (C'_L, \mathbb{E}) = closure(\overline{a}, \mathcal{A}_S, C_L)$$

Function *closure* computes the transitive closure of the following function:

$$\begin{aligned} &mponens(\overline{a}, \mathcal{A}, d : \mathtt{TC}\ \tau_1) = (bimap\ mconcat\ mconcat \cdot unzip) \\ &\quad \{\ (\{d_2 : \theta(Q)\}, \mathbb{E}) \\ &\quad \mid (d_1 : \forall \overline{b}.\ \mathtt{TC}\ \tau_1 \Rightarrow Q) \in \mathcal{A} \\ &\quad ,\ unify(\overline{a}; \tau_1 \sim \tau_2) = \theta \\ &\quad ,\ \mathbb{E} = \textbf{let}\ d_2 : elab_{\mathtt{CT}}(\theta(Q)) = d_1\ \theta(\overline{b})\ d\ \textbf{in}\ \square \\ &\quad \} \end{aligned}$$

Function $mponens(\overline{a}, \mathcal{A}, Q) = (C, \mathbb{E})$ tries to match the left-hand side of every available constraint scheme in $\mathcal{A}$ with the given constraint. If matching is successful, modus ponens is used to derive the right-hand side. This procedure is also reflected in the dictionary context $\mathbb{E}$, which captures a scope

where the derived dictionaries are available. For example, if

$$\begin{aligned}\overline{a} &= m\\ \mathcal{A}_S &= \{\ d_1 : \forall n.\ \mathit{Monad}\ n \Rightarrow \mathit{Applicative}\ n\\ &\quad\ ,\ d_2 : \forall k.\ \mathit{Applicative}\ k \Rightarrow \mathit{Functor}\ k\ \}\\ Q &= d_3 : \mathit{Monad}\ m\end{aligned}$$

then $\mathit{closure}(\overline{a}, \mathcal{A}_S, Q)$ results in the following:

$$\begin{aligned}\mathcal{A} &= \{\ d_4 : \mathit{Applicative}\ m, d_5 : \mathit{Functor}\ m\ \}\\ \mathbb{E} &= \textbf{let}\ d_4 : T_{\mathit{Applicative}}\ m = d_1\ m\ d_3\ \textbf{in}\\ &\quad\ \textbf{let}\ d_5 : T_{\mathit{Functor}}\ m = d_2\ m\ d_4\ \textbf{in}\ \square\end{aligned}$$

In plain type inference, superclasses are never used; the above procedure is required in type *checking*. This is the case for method implementations, explicitly-annotated terms, and the entailment of superclass constraints in class instances. This is better illustrated in the elaboration of declarations, which we discuss next.

5.5.5 Declaration Elaboration

We now turn to type inference and elaboration for top-level declarations. Since type inference with elaboration for class declarations is identical to its specification, we only discuss the judgments for class instances and top-level bindings.

Instance Inference with Elaboration Typing inference for instance declarations takes the form $P; \Gamma \vdash_{\text{INS}} \mathit{ins} : P' \rightsquigarrow \mathit{decl}$ and is given by the following rule:

$$\frac{\begin{array}{c}\textbf{class}\ \forall a.\ \overline{Q^{\text{sup}}}^{n} \Rightarrow \texttt{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \} \qquad \Gamma_I = \Gamma, \overline{b}\\ P_I = P, \overline{d : Q}^{m} \qquad \Gamma_I \vdash_{\text{CT}} \overline{Q}^{m} \rightsquigarrow \overline{v}^{m} \qquad \Gamma_I \vdash_{\text{TY}} \tau \rightsquigarrow \upsilon\\ S = \forall \overline{b}.\ \overline{Q}^{m} \Rightarrow \texttt{TC}\ \tau \qquad \textsc{ScClosure}(\overline{b}, P_I) = (\mathcal{A}, \mathbb{E})\\ \overline{b}; \mathcal{A} \models \overline{d : Q}^{n} \rightsquigarrow \bullet ; \eta \qquad \overline{b}; P_I, \overline{d : S}; \Gamma_I \vdash_{\text{TM}} e : [\tau/a]\sigma \rightsquigarrow t\\ \mathit{decl} = \textbf{let}\ d : \forall \overline{b}.\overline{v_i}^{m} \rightarrow T_{\texttt{TC}}\ \upsilon = \Lambda \overline{b}.\ \lambda(\overline{d : v}^{m}).\ K_{\texttt{TC}}\ \upsilon\ \mathbb{E}[\eta(\overline{d^{\text{sup}}}^{n})]\ t\end{array}}{P; \Gamma \vdash_{\text{INS}} \textbf{instance}\ \forall \overline{b}.\overline{Q}^{m} \Rightarrow \texttt{TC}\ \tau\ \textbf{where}\ \{f = e\} : [\overline{d : S}] \rightsquigarrow \mathit{decl}}$$

For the most part it is identical to the corresponding rule in Figure 2. The most notable differences are concentrated around superclass entailment and type checking of the method implementation.

For the entailment of the superclass constraints we precompute the transitive closure of the superclass relation, and then (a) we generate fresh dictionary variables $\overline{d^{\text{sup}}}^{n}$, to capture the yet-unknown superclass dictionaries, and (b) we exhaustively simplify the superclass constraints (requiring no residual constraints), obtaining an evidence substitution η. η maps dictionary variables $\overline{d^{\text{sup}}}^{n}$ to generated dictionaries; the complete witness for the i-th superclass dictionary takes the form $\mathbb{E}[\eta(\theta(d_i^{\text{sup}}))]$.

Method implementations have their type imposed by their signature in the class declaration. Hence, we need to *check* rather than *infer* their type. This operation is expressed succinctly by relation $\overline{a}; P; \Gamma \vdash_{\text{TM}} e : \sigma \rightsquigarrow t$:

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{TM}} e : \tau_1 \rightsquigarrow t \mid C; E \qquad \theta = \mathit{unify}(\overline{a}, \overline{b}; E, \tau_1 \sim \tau_2)\\ \textsc{ScClosure}(\overline{a}, (P, \overline{d : Q}^{n})) = (\mathcal{A}, \mathbb{E})\\ \overline{a}, \overline{b}; \mathcal{A} \models \theta(C) \rightsquigarrow \bullet ; \eta\end{array}}{\overline{a}; P; \Gamma \vdash_{\text{TM}} e : (\forall \overline{b}.\ \overline{Q}^{n} \Rightarrow \tau_2) \rightsquigarrow \Lambda \overline{b}.\ \lambda(\overline{d : \mathit{elab}_{\text{CT}}(Q)}^{n}).\ \mathbb{E}[\eta(\theta(t))]}$$

Essentially, it ensures that the inferred type for e *subsumes* the expected type σ. A type σ_1 is said to subsume type σ_2 if any expression that can be assigned type σ_1 can also be assigned type σ_2. The above rule performs type inference and type subsumption checking simultaneously: First, it infers a monotype τ_1 for expression e, as well as wanted constraints C and type equalities E. Type equalities E should have a unifier and the inferred type τ_1 should also be unifiable with the expected type τ_2. Finally, the given constraints $\overline{Q}^{n}$ should completely entail the wanted constraints C. For constraint entailment, we (again) pre-populate the given constraints with the transitive closure of the superclass axioms.

Value Binding Inference with Elaboration Finally, type inference for top-level bindings is given by the judgment $P; \Gamma \vdash_{\text{VAL}} \mathit{val} : \Gamma' \rightsquigarrow \mathit{decl}$. The first rule deals with annotation-free bindings:

$$\frac{\begin{array}{c}\Gamma, x : b \vdash_{\text{TM}} e : \tau \rightsquigarrow t \mid C; E \qquad \mathit{unify}(\bullet; E, b \sim \tau) = \theta\\ \overline{a} = \mathit{fv}(\theta(C)) \cup \mathit{fv}(\theta(\tau)) \qquad \overline{a}; \mathcal{A}_I, C_L \models \theta(C) \rightsquigarrow (\overline{d : Q}^{n}) ; \eta\\ \upsilon = \mathit{elab}_{\text{TY}}(\forall \overline{a}.\ \overline{Q}^{n} \Rightarrow \theta(\tau))\\ \text{for each } Q_i \in \overline{Q}^{n}.\ \ v_i = \mathit{elab}_{\text{CT}}(Q_i)\\ \mathit{decl} = \textbf{let}\ x : \upsilon = \Lambda \overline{a}.\ \lambda(\overline{d : v}^{n}).\ \mathbb{E}[\eta(\theta(t))]\end{array}}{\langle \mathcal{A}_S, \mathcal{A}_I, C_L \rangle; \Gamma \vdash_{\text{VAL}} x = e : [x : \forall \overline{a}.\ \overline{Q}^{n} \Rightarrow \theta(\tau)] \rightsquigarrow \mathit{decl}}$$

The rule performs constraint generation, simplification, and generalization of an annotation-free top-level binding. Though straightforward, it is worth noticing that superclass axioms $\mathcal{A}_S$ are ignored, since there are no local (given) constraints.

The second rule deals with explicitly annotated bindings:

$$\frac{\bullet; P; \Gamma, x : \sigma \vdash_{\text{TM}} e : \sigma \rightsquigarrow t}{P; \Gamma \vdash_{\text{VAL}} \textbf{let}\ x : \sigma = e : [x : \sigma] \rightsquigarrow \textbf{let}\ x : \mathit{elab}_{\text{TY}}(\sigma) = t}$$

Essentially, type inference for annotated terms directly corresponds to an inference-and-subsumption-check, as given by judgment $\overline{a}; P; \Gamma \vdash_{\text{TM}} e : \sigma \rightsquigarrow t$ above.

6 Bidirectional Instances, Formally

In this section we present the changes needed for extending the basic system of Section 5 with support for bidirectional instances.

6.1 Syntax Extensions

First, in order to use the inverted axioms selectively and avoid the termination issue we mentioned in Section 3.3, we

extend the syntax of program theory P with an additional component, the inverted instance axioms $\mathcal{A}_B$:[12]

$$P ::= \langle \mathcal{A}_B, \mathcal{A}_S, \mathcal{A}_I, C_L \rangle \qquad \textit{program theory}$$

As we illustrate below—similarly to superclass axioms $\mathcal{A}_S$—inverted instance axioms $\mathcal{A}_B$ are used for type checking but not for type inference. The rest of the syntax is identical to the syntax of the basic system we presented in Figure 1a.

6.2 Specification Extensions

The specification of typing and elaboration is for the most part identical to that of Section 5.4. The changes bidirectional instances introduce are concentrated in class and instance declaration typing, which we now discuss.

6.2.1 Class Declarations

The specification of class typing with elaboration – as well as for the basic system – is given by judgment $\Gamma \vdash_{\text{CLS}} cls : P; \Gamma' \rightsquigarrow \overline{decl}$ (Figure 2). Since for the most part the rule is identical to the basic system, we only highlight the differences. For a class declaration of the form

$$\textbf{class}\ \forall a.\ \overline{Q}^n \Rightarrow \mathsf{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \}$$

we have the following:

1. Instance Context Firstly, in addition to the superclass and method projections, the class declaration gives rise to a System $\mathrm{F_C}$ open type family declaration:

$$\textbf{type}\ F_{\mathsf{TC}}\ a$$

Function $F_{\mathsf{TC}}\ a$ captures the functional dependency between the instance context and the class parameter. Hence, function F_{TC} is populated by TC instances, each mapping its class parameter to the corresponding instance context.

2. Dictionary Representation Secondly, we extend the dictionary declaration, so that it can store the instance context of type $F_{\mathsf{TC}}\ a$:[13]

$$\textbf{data}\ T_{\mathsf{TC}}\ a = K_{\mathsf{TC}}\ (F_{\mathsf{TC}}\ a)\ \overline{v}^n\ v$$

3. Projection Functions Finally, since the data constructor K_{TC} now stores an additional field, we "shift" the superclass and method projections accordingly:

$$\textbf{let}\ d_i : \forall a.\ T_{\mathsf{TC}}\ a \to v_i = \Lambda a.\ \lambda(d : T_{\mathsf{TC}}\ a).\ proj^{i+1}_{\mathsf{TC}}(d)$$
$$\textbf{let}\ f : \forall a.\ T_{\mathsf{TC}}\ a \to v = \Lambda a.\ \lambda(d : T_{\mathsf{TC}}\ a).\ proj^{n+2}_{\mathsf{TC}}(d)$$

6.2.2 Instance Declarations

Typing for instance declarations also preserves the signature we gave in Figure 2. For a class instance of the form

$$\textbf{instance}\ \forall \overline{b}.\ \overline{Q}^m \Rightarrow \mathsf{TC}\ \tau\ \textbf{where}\ \{\ f = e\ \}$$

bidirectional instances introduce the following extensions:

1. Instance Context Axiom Firstly, an additional clause is generated for function F_{TC}, capturing the dependency between the instance parameter τ and the instance context:

$$\textbf{axiom}\ g^{\mathsf{TC}}_{\tau}\ \overline{b} : F_{\mathsf{TC}}\ v \sim (v_1, \ldots, v_m)$$

where v_i is the dictionary type representation of Q_i in the instance context and v is the elaboration of parameter τ.

2. Inverted Instance Axioms Secondly, the program theory extension introduced by the instance now includes the inverted instance axioms, which take the form:

$$S_i = \forall \overline{b}.\ \mathsf{TC}\ \tau \Rightarrow Q_i \qquad i \in [1 \ldots m]$$

Of course, such implications need to be reflected in term-level functions in the generated System $\mathrm{F_C}$ code. Hence, for every implication S_i, we generate a projection function d_i, given by the following definition:

$$\begin{aligned}&\textbf{let}\ d_i\ :\ \forall \overline{b}.\ T_{\mathsf{TC}}\ v \to v_i\\ &\quad = \Lambda \overline{b}.\ \lambda(d : T_{\mathsf{TC}}\ v).\ \textbf{case}\ d\ \textbf{of}\\ &\qquad K_{\mathsf{TC}}\ ctx\ \overline{d}^n\ x \to \textbf{case}\ ctx \triangleright (g^{\mathsf{TC}}_{\tau}\ \overline{b})\ \textbf{of}\ (d_1, ..., d_m) \to d_i\end{aligned}$$

The outer pattern matching exposes the instance context ctx, of type $F_{\mathsf{TC}}\ v$, which we explicitly cast to a tuple of all instance context dictionaries $(d_1, \ldots, d_m)$. Then, the inner pattern matching extracts and returns the corresponding instance context dictionary d_i.

3. Storing the Instance Context Finally, the implementation of the instance dictionary (transformer) needs to store the instance context dictionaries within the dictionary for $\mathsf{TC}\ \tau$. Thus, the instance dictionary (transformer) now takes the form:

$$\begin{aligned}&\textbf{let}\ d\ :\ \forall \overline{b}.\ \overline{v}^m_i \to T_{\mathsf{TC}}\ v\\ &\quad = \Lambda \overline{b}.\ \lambda(\overline{d : v}^m).\ K_{\mathsf{TC}}\ v\ ((d_1, \ldots, d_m) \triangleright \mathrm{sym}\ (g^{\mathsf{TC}}_{\tau}\ \overline{b}))\ \overline{t}^n\ t]\end{aligned}$$

For the constructed dictionary to be well-typed, the tuple $(d_1, \ldots, d_m)$ containing all instance context dictionaries needs to be explicitly cast to have type $F_{\mathsf{TC}}\ v$, as the type of K_{TC} requires. This is exactly what γ proves: $(v_1, \ldots, v_m) \sim F_{\mathsf{TC}}\ v$.

6.3 Algorithm Extensions

Type inference is again for the most part identical to that of the basic system (Section 5.5). The changes bidirectional instances introduce are concentrated in declarations. Type inference for classes is identical to its specification so we only discuss the differences in class instances and value bindings.

Instance Declarations Type inference for instance declarations behaves similarly to its specification. The main difference lies in the type inference and subsumption checking for methods.

In addition to the superclass closure we also compute the transitive closure of the inverted axioms. Thus, we replace function $\textsc{ScClosure}(\overline{a}, P) = (\mathcal{A}, \mathbb{E})$ of Section 5.5.4 with

[12] Subscript B stands for "Bidirectional".

[13] The order of the dictionary arguments is irrelevant, and the choice made here is arbitrary.

$\textsc{InvScClosure}(\overline{a}, P) = (\mathcal{A}, \mathbb{E})$:

$$\textsc{InvScClosure}(\overline{a}, \langle \mathcal{A}_B, \mathcal{A}_S, \mathcal{A}_I, C_L \rangle) = ((C'_L, \mathcal{A}_I, C_L), \mathbb{E})$$
$$\text{where } (C'_L, \mathbb{E}) = \mathit{closure}(\overline{a}, (\mathcal{A}_B, \mathcal{A}_S), C_L)$$

Value Bindings Type inference for value bindings is also mildly affected by bidirectional instances. Bindings without a type annotation ignore the inverted axioms, alongside the superclass axioms; users that enable bidirectional instances can expect type inference to behave as usual.

A top-level value binding with an explicit type annotation behaves differently, similarly to method typing. Just as we do with superclass constraints, we also compute the transitive closure of the inverted axioms, making more derivations possible. This extension—along with all the changes we described in this section—manifests itself in the elaboration of cmp_2 (Section 4), as dictionaries d'_1 and d'_2.

7 Meta-theory

Since the formalization of bidirectional instances conservatively extends that of Section 6, we focus on the most interesting meta-theoretical properties of our extension: termination of type inference and the principal type property.

7.1 Termination

Termination Conditions First, for decidable type inference it is required that type inference terminates on all inputs. The following *Termination Conditions* are sufficient to ensure termination of type inference:

(a) The superclass relation forms a *Directed Acyclic Graph*.
(b) In each class instance (**instance** $\forall \overline{b}.\ C \Rightarrow \mathtt{TC}\ \tau$):
- no variable has more occurrences in a type class constraint in the instance context C than the head ($\mathtt{TC}\ \tau$),
- each class constraint in the instance context C has fewer constructors and variables (taken together, counting repetitions) than the head ($\mathtt{TC}\ \tau$).

The first restriction ensures that the computation of the transitive closure of the superclass relation is terminating [Peyton Jones 2003, Sec. 4.3.1]. The second restriction [Sulzmann et al. 2007b, Def. 11] ensures that instance contexts are *decreasing*, so that class resolution is also terminating. To illustrate why type inference in the presence of bidirectional instances terminates, we first distinguish between type inference and type checking.

Termination of Type Inference In cases where a type is inferred, the algorithm is identical to that of the basic system; the feature manifests itself when there are type signatures. Hence, in these cases decreasing instance contexts are sufficient to ensure termination.

Termination of Type Checking In cases where we need to check an expression against a type, the inverted axioms also come into play, as well as the superclass axioms. Since we compute the closure of the superclass relation and the inverted axioms (by means of function $\textsc{InvScClosure}$), we need to ensure that both superclass and inverted axioms cannot be applied indefinitely. For the former, Condition (a) is sufficient: any uninterrupted sequence of superclass axiom applications is bounded by the height of the superclass graph. For the latter, decreasing contexts are also sufficient. To illustrate why, consider the following inverted axiom:

$$\forall a.\ \forall b.\ \mathit{Eq}\ (a, b) \Rightarrow \mathit{Eq}\ a$$

During completion, $\textsc{InvScClosure}$ applies the axiom to constraints of the form $\mathit{Eq}\ (\tau_1, \tau_2)$, ending up with an additional axiom of a smaller size: $\mathit{Eq}\ \tau_1$. In short, uninterrupted sequences of inverted axioms are bounded by the size of the types in instance heads. In short, any step $\textsc{InvScClosure}$ takes either reduces the size of a constraint, or takes a step in the superclass graph, both of which are bounded.

7.2 Principality of Types

Our specification (Sections 5.4 and 6.2) possesses the principal type property: the definition of a principal type does not specify one type, but rather the properties of it. That is, the following types of *cmp* (Section 3) are both equally general:

$$\mathit{cmp} :: \forall a.\ \mathit{Eq}\ a \quad \Rightarrow a \to a \to \mathit{Bool}$$
$$\mathit{cmp} :: \forall a.\ \mathit{Eq}\ [a] \Rightarrow a \to a \to \mathit{Bool}$$

Hence, the main concern is whether the type inference algorithm of Sections 5.5 and 6.3 infers one of the principal types. The answer is *yes*. Since plain type inference does not exploit the inverted axioms, the algorithm infers backwards-compatible principal types. Backwards-chaining simplifies constraints such as $\mathit{Eq}\ [a]$ to $\mathit{Eq}\ a$ but not the other way around. Thus, the algorithm would never infer type

$$\forall a.\ \forall b.\ \mathit{Eq}\ (a, b) \Rightarrow \ldots$$

but would infer the isomorphic (and also principal) type

$$\forall a.\ \forall b.\ (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow \ldots$$

Expressions with explicit type annotations have only one principal type: the one specified by their signature. In these cases the algorithm will use the inverted axioms to entail the wanted constraints ($\mathit{Eq}\ a$, $\mathit{Eq}\ b$) using the given $\mathit{Eq}\ (a, b)$, thus constructing again the principal type.

That is, in the absence of type annotations the principal type is the principal Haskell98 type, and in the presence of type annotations the type annotation dictates what the principal type is. In either case, our algorithm reconstructs the principal type, therefore addressing the challenge of Section 3.4.

7.3 Other Properties

Preservation of Typing Under Elaboration We are confident that the specification of elaboration we gave in Sections 5.4 and 6.2 is type-preserving. The formal proof of this statement we leave for future work.

Soundness of Generated Code It is known that overlapping instances make the semantics of type classes incoherent but they do not introduce unsoundness. In the presence of bidirectional instances, this is no longer true:

$$\begin{array}{lll} \textbf{instance}\ Eq\ a \Rightarrow Eq\ [a] & \rightsquigarrow & \textbf{axiom}\ g_1\ a : F_{Eq}\ [a] \sim T_{Eq}\ a \\ \textbf{instance}\ Eq\ [b] & \rightsquigarrow & \textbf{axiom}\ g_2\ b : F_{Eq}\ [b] \sim () \end{array}$$

Axioms g_1 and g_2 violate the System F_C *compatibility condition* [Eisenberg et al. 2014, Defn. 10], which means that our elaboration would give rise to unsound System F_C code. Indeed, $(\text{sym}\ (g_1\ Int)) \fatsemi (g_2\ Int)$ is a proof of $T_{Eq}\ Int \sim ()$. We revisit this issue in Section 8.

Coherence In the absence of overlapping instances and ambiguous types, we conjecture that our elaboration is *coherent*. Given the similarity between the handling of superclass constraints and bidirectional instances, we are confident that the recent advances of Bottu et al. [2019] could be easily extended to accommodate bidirectional instances.

Algorithm Soundness and Completeness Finally, we conjecture that the algorithm of Sections 5.5 and 6.3 is sound and complete with respect to its specification.

8 Related Work and Discussion

Class Elaboration Maybe the most relevant line of work is the specification of typing and elaboration (into System F) of type classes with superclasses, given by Hall et al. [1996]. Yet, the work of Hall et al. does not cover an algorithm for type inference and elaboration; we do so here (Section 5.5).

Constrained Type Families Morris and Eisenberg [2017] recently provided compelling arguments for the replacement of open type families with the so-called *Constrained Type Families*. Constrained type families, similarly to associated type families, use the generic notion of qualified types [Jones 1992] to capture the domain of a type family within a predicate, thus simplifying the meta-theory of type families and their extensions.

Within this setting, the bidirectionality of the axioms is essential. Indeed, Morris and Eisenberg use a variation of the the *append* example (Section 2) to motivate the extension of System F_C with the `assume` construct, which axiomatically provides the bidirectionality needed for *append* to type check. Unfortunately, `assume` is not a panacea: axiomatically assuming the satisfiability of constraints does not scale to class methods.[14]

Overlapping Instances As we mentioned in Section 7.3, bidirectional instances can lead to unsound System F_C code in the presence of the (in)famous `OverlappingInstances` GHC extension. Though this extension is considered harmful—and has thus been deprecated since GHC 7.10 in favour of more fine-grained per-instance pragmas—it is still used, making it important to study its interaction with our feature.

Depending on the level of overlap allowed, we can selectively make instances bidirectional: the system is sound if overlap and bidirectionality are aligned. Indeed, instances determine the generated axioms so our strategy is simple: *any instance that overlaps with other instances should not give rise to any inverted axioms.*

In terms of the overlapping Eq instances of the previous section, this means that we would give rise to

$$\begin{array}{lll} \textbf{instance}\ Eq\ a \Rightarrow Eq\ [a] & \rightsquigarrow & \textbf{axiom}\ g_1\ a : F_{Eq}\ [a] \sim () \\ \textbf{instance}\ Eq\ [b] & \rightsquigarrow & \textbf{axiom}\ g_2\ b : F_{Eq}\ [b] \sim () \end{array}$$

thus ensuring safety of the generated code.

Instance Chains Though our design generates "open" equality axioms (to agree with the open nature of type classes), one might also consider bidirectionality in the presence of *"instance chains"* [Morris and Jones 2010]. Instance chains allow for ordered overlap among instances, which we believe can be combined with our interpretation. Instead of a collection of open axioms, an instance chain can give rise to a "closed" equality axiom (like the ones generated by *closed type families* [Eisenberg et al. 2014]), to preserve soundness of the generated code without sacrificing expressive power.

Inversion Principles in Proof Assistants There is also a large body of work concerned with *inversion principles*, with significant applications in the area of proof assistants (see for example tactic `inversion`). Though inversion principles seem like a more natural approach for addressing the problem we target here, the open nature of type classes disallows a direct application to Haskell. Nevertheless, we would like to explore this alternative approach in the future.

Denotational Semantics for Type Classes Morris [2014] gives an—inherently bidirectional—denotational semantics for type classes, rather than through a dictionary-passing translation. Within this work, polymorphic instances are interpreted extensionally, as the set of their ground instantiations. Unfortunately, it has not been studied yet how this semantics relates to the traditional dictionary-based semantics that we target here.

Quantified Class Constraints An interesting avenue for future work is studying the interaction between *Quantified Class Constraints* [Bottu et al. 2017] and Bidirectional Instances. The two key challenges are (a) elaboration and (b) type inference.

Combining the elaboration strategies of the features is a straightforward task. For example, the instance

$$\textbf{instance}\ (\forall a.\ Monoid\ (f\ a)) \Rightarrow Alternative\ f$$

[14] This is also the case for other attempts to tackle the same problem using other type-level features. See for example http://okmij.org/ftp/Haskell/number-parameterized-types.html#binary-arithm.

generates the following System F_C axiom:[15]

$$\textbf{axiom}\ g\ f : F_{Alternative}\ f \sim \forall a.\ T_{Monoid}\ (f\ a)$$

The second aspect, type inference, is more interesting. The main challenge lies in the significantly different constraint entailment strategies: Quantified Class Constraints use backtracking to ensure completeness, but Bidirectional Instances can lead to non-termination in the presence of backtracking (see Section 3.3). We believe that a restricted combination of the two features is possible,[16] and plan to investigate their interaction in the future.

9 Conclusion

We have presented a conservative extension of type classes, which allows class instances to be interpreted bidirectionally, thus significantly improving the interaction of GADTs with type classes, by allowing proper structural induction over GADTs, even in the presence of qualified types.

Acknowledgments

We would like to thank Steven Keuchel, and the anonymous reviewers of Haskell Symposium 2018 and 2019 for careful reading and insightful comments. This research was supported by the Flemish Fund for Scientific Research (FWO).

References

Henk Barendregt. 1981. *The Lambda Calculus: its Syntax and Semantics, volume 103 of Studies in Logic and the Foundations of Mathematics.*

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Haskell 2017.*

Gert-Jan Bottu, Ningning Xie, Klara Marntirosian, and Tom Schrijvers. 2019. Coherence of Type Class Resolution. *Proc. ACM Program. Lang.* (2019). Accepted.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005a. Associated Type Synonyms. *SIGPLAN Not.* 40, 9 (2005), 241–253.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005b. Associated Types with Class. In *POPL '05.* ACM, 1–13.

Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *POPL '82.* ACM, 207–212.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *POPL '14.*

Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation.

Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. 2006. Concepts: Linguistic Support for Generic Programming in C++. *SIGPLAN Not.* 41, 10 (2006), 291–310.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *TOPLAS* 18, 2 (March 1996).

Thomas Hallgren. 2000. Fun with Functional Dependencies. In *Proc. of the Joint CS/CE Winter Meeting.*

Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. 1996. *The Mercury Language Reference Manual.* Technical Report.

Fritz Henglein. 1993. Type Inference with Polymorphic Recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (April 1993), 253–289.

R. Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60.

Patricia Johann and Neil Ghani. 2008. Foundations for Structured Programming with GADTs. *SIGPLAN Not.* 43, 1 (Jan. 2008), 297–308.

Mark P. Jones. 1992. A theory of qualified types. In *ESOP '92*, Bernd Krieg-Brückner (Ed.). LNCS, Vol. 582. Springer Berlin Heidelberg, 287–306.

Mark P. Jones. 1995a. *Qualified Types: Theory and Practice.* Cambridge University Press.

Mark P. Jones. 1995b. Simplifying and Improving Qualified Types. In *FPCA '95.* ACM, 160–169.

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Programming Languages and Systems.* LNCS, Vol. 1782. Springer.

Georgios Karachalias and Tom Schrijvers. 2017. Elaboration on Functional Dependencies: Functional Dependencies Are Dead, Long Live Functional Dependencies! *SIGPLAN Not.* 52, 10 (Sept. 2017), 133–147.

Robert Kowalski. 1974. Predicate Logic as Programming Language. In *Proceedings of IFIP '74.* North Holland, 569–574.

The Coq development team. 2004. *The Coq proof assistant reference manual.* LogiCal Project. http://coq.inria.fr Version 8.0.

J. Garrett Morris. 2014. A Simple Semantics for Haskell Overloading. *SIGPLAN Not.* 49, 12 (2014), 107–118.

J. Garrett Morris and Richard A. Eisenberg. 2017. Constrained Type Families. In *ICFP '17.* ACM.

J. Garrett Morris and Mark P. Jones. 2010. Instance Chains: Type Class Programming Without Overlapping Instances. In *ICFP '10.* ACM.

Giuseppe Peano. 1889. *Arithmetices principia: nova methodo exposita.* Fratres Bocca. https://books.google.be/books?id=UUFtAAAAMAAJ

Simon Peyton Jones. 2003. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-based Type Inference for GADTs. *SIGPLAN Not.* 41, 9 (2006), 50–61.

Norman Ramsey, João Dias, and Simon Peyton Jones. 2010. Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. *SIGPLAN Not.* 45, 11 (Sept. 2010), 121–134.

John C. Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation.* 408–423.

John C. Reynolds. 1983a. Types, Abstraction, and Parametric Polymorphism. In *Information Processing 83*, R.E.A. Mason (Ed.). 513–523.

John C. Reynolds. 1983b. Types, abstraction and parametric polymorphism. (1983).

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type Checking with Open Type Functions. In *ICFP '08.* ACM, 51–62.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. 2007a. System F with Type Equality Coercions. In *TLDI '07.* ACM.

Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. 2007b. Understanding Functional Dependencies via Constraint Handling Rules. *J. Funct. Program.* 17, 1 (2007), 83–129.

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 (Sept. 2011), 333–412.

P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *POPL '89.* ACM, 60–76.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *TLDI '12.* ACM, 53–66.

[15] Notice though that this encoding needs more System F_C power than GHC currently uses; it is impossible to encode Bidirectional Instances combined with Quantified Class Constraints using the current GHC version.

[16] GHC also supports a limited version of Quantified Constraints (see commit 7df589608abb178efd6499ee705ba4eebd0cf0d1), without backtracking.