Quantified Class Constraints

Gert-Jan Bottu
KU Leuven
gertjan.bottu@student.kuleuven.be

Georgios Karachalias
KU Leuven
georgios.karachalias@cs.kuleuven.be

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

Bruno C. d. S. Oliveira
The University of Hong Kong
bruno@cs.hku.hk

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

Abstract

Quantified class constraints have been proposed many years ago to raise the expressive power of type classes from Horn clauses to first-order logic. Yet, while it has been much asked for over the years, the feature was never implemented or studied in depth. Instead, several workarounds have been proposed, all of which are ultimately stopgap measures.

This paper revisits the idea of quantified class constraints and elaborates it into a practical language design. We show the merit of quantified class constraints in terms of more *expressive modeling* and in terms of *terminating type class resolution*. In addition, we provide a declarative specification of the type system as well as a type inference algorithm that elaborates into System F. Moreover, we discuss termination conditions of our system and also provide a prototype implementation.

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

Keywords Haskell, type classes, type inference

ACM Reference format:
Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Proceedings of ACM SIGPLAN Haskell Symposium 2017, Oxford, UK, September 7 – 8, 2017 (Haskell'17),* 14 pages.
DOI: 10.475/123_4

1 Introduction

Since Wadler and Blott [38] originally proposed type classes as a means to make adhoc polymorphism less adhoc, the feature has become one of Haskell's cornerstone features. Over the years type classes have been the subject of many language extensions that increase their expressive power and enable new applications. Examples of such extensions include: multi-parameter type classes [19]; functional dependencies [18]; or associated types [4].

Several of these implemented extensions were inspired by the analogy between type classes and predicates in Horn clauses. Yet, Horn clauses have their limitations. As a small side-product of their work on derivable type classes, Hinze and Peyton Jones [12] have proposed to raise the expressive power of type classes to essentially first-order logic with what they call *quantified class constraints*. Their motivation was to deal with higher-kinded types which seemed to require instance declarations that were impossible to express in the type-class system of Haskell at that time.

Unfortunately, Hinze and Peyton Jones never did elaborate on quantified class constraints. Later, Lämmel and Peyton Jones [21] found a workaround for the particular problem of the derivable type classes work that did not involve quantified class constraints. Nevertheless the idea of quantified class constraints has whet the appetite of many researchers and developers. GHC ticket #2893[1], requesting for quantified lass constraints, was opened in 2008 and is still open today. Commenting on this ticket in 2009, Peyton Jones states that *"their lack is clearly a wart, and one that may become more pressing"*, yet clarifies in 2014 that *"(t)he trouble is that I don't know how to do type inference in the presence of polymorphic constraints."* In 2010, 10 years after the original idea, Hinze [10] rues that the feature has not been implemented yet. As recently as 2016, Chauhan et al. [5] regret that "*Haskell does not allow the use of universally quantified constraints*" and now in 2017 Spivey [34] has to use pseudo-Haskell when modeling with quantified class constraints. While various workarounds have been proposed and are used in practice [20, 31, 36], none has stopped the clamor for proper quantified class constraints.

This paper finally elaborates the original idea of quantified class constraints into a fully fledged language design.

Specifically, the contributions of this paper are:

- We provide an overview of the two main advantages of quantified class constraints (Section 2):
  1. they provide a natural way to express more of a type class's specification, and
  2. they enable terminating type class resolution for a larger class of applications.
- We elaborate the type system sketch of Hinze and Peyton Jones [12] for quantified type class constraints into a full-fledged formalization (Section 3). Our formalization borrows the idea of focusing from Cochis [32], a calculus for Scala-style implicits [26, 27], and adapts it to the Haskell setting. We account for two notable differences: a global set of non-overlapping instances and support for superclasses.
- We present a type inference algorithm that conservatively extends that of Haskell 98 (Section 4) and comes with a dictionary-passing elaboration into System F (Section 5).
- We discuss the termination conditions on a system with quantified class constraints (Section 6).
- We provide a prototype implementation, which incorporates higher-kinded datatypes and accepts all[2] examples in this paper, at https://github.com/gkaracha/quantcs-impl.

Haskell'17, Oxford, UK

DOI: 10.475/123_4

[1] https://ghc.haskell.org/trac/ghc/ticket/2893
[2] except for the *HFunctor* example (Section 2.1), which needs higher-rank types [28].

2 Motivation

This section illustrates the expressive power afforded by quantified class constraints to capture several requirements of type class instances more succinctly, and to provide terminating resolution for a larger group of applications.

2.1 Precise and Succinct Specifications

Monad Transformers Consider the MTL type class for monad transformers [15]:

$$\begin{array}{l}\textbf{class}\ \mathit{Trans}\ t\ \textbf{where}\\ \quad \mathit{lift} :: \mathit{Monad}\ m \Rightarrow m\ a \to (t\ m)\ a\end{array}$$

What is not formally expressed in the above type class declaration, but implicitly expected, is that for any type T that instantiates *Trans* there should also be a *Monad* instance of the form:

$$\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (T\ m)\ \textbf{where}\ \ldots$$

Because the type checker is not told about this requirement, it will not accept the following definition of monad transformer composition.

$$\textbf{newtype}\ (t_1 * t_2)\ m\ a = C\ \{\ \mathit{runC} :: t_1\ (t_2\ m)\ a\ \}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Trans}\ t_1, \mathit{Trans}\ t_2) \Rightarrow \mathit{Trans}\ (t_1 * t_2)\ \textbf{where}\\ \quad \mathit{lift} = C \cdot \mathit{lift} \cdot \mathit{lift}\end{array}$$

The idea of this code is to *lift* from monad m to $(t_2\ m)$ and then to *lift* from $(t_2\ m)$ to $t_1\ (t_2\ m)$. However, the second *lift* is only valid if $(t_2\ m)$ is a monad and the type checker has no way of establishing that this fact holds for all monad transformers t_2. Workarounds for this problem do exist in current Haskell [13, 31, 36], but they clutter the code with heavy encodings.

Quantified class constraints allow us to state this requirement explicitly as part of the *Trans* class declaration:

$$\begin{array}{l}\textbf{class}\ (\forall m.\mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (t\ m)) \Rightarrow \mathit{Trans}\ t\ \textbf{where}\\ \quad \mathit{lift} :: \mathit{Monad}\ m \Rightarrow m\ a \to (t\ m)\ a\end{array}$$

The instance for transformer composition $t_1 * t_2$ now typechecks.

Second-Order Functors Another example can be found in the work of Hinze [11]. He represents parameterized datatypes, like polymorphic lists and trees, as the fixpoint *Mu* of a *second-order functor*:

$$\textbf{data}\ \mathit{Mu}\ h\ a = \mathit{In}\ \{\ \mathit{out} :: h\ (\mathit{Mu}\ h)\ a\ \}$$

$$\textbf{data}\ \mathit{List}_2\ f\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (f\ a)$$

$$\textbf{type}\ \mathit{List} = \mathit{Mu}\ \mathit{List}_2$$

A second-order functor h is a type constructor that sends functors to functors. This can be concisely expressed with the quantified class constraint $\forall f.\mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (h\ f)$, for example in the *Functor* instance of *Mu*:

$$\begin{array}{l}\textbf{instance}\ (\forall f.\mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (h\ f)) \Rightarrow \mathit{Functor}\ (\mathit{Mu}\ h)\\ \quad \textbf{where}\ \mathit{fmap}\ f\ (\mathit{In}\ x) = \mathit{In}\ (\mathit{fmap}\ f\ x)\end{array}$$

Although this is Hinze's preferred formulation he remarks that:

> *Unfortunately, the extension has not been implemented yet. It can be simulated within Haskell 98 [36], but the resulting code is somewhat clumsy.*

Johann and Ghani use essentially the same data-generic representation, the fixpoint of second-order functors, to represent so-called *nested datatypes* [3]. For instance, Hinze [10] represents perfect binary trees with the nested datatype

$$\textbf{data}\ \mathit{Perfect}\ a = \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Perfect}\ (a, a))$$

This can be expressed with the generic representation as *Mu HPerf*, the fixpoint of the second-order functor *HPerf*, defined as

$$\textbf{data}\ \mathit{HPerf}\ f\ a = \mathit{HZero}\ a \mid \mathit{HSucc}\ (f\ (a, a))$$

Johann and Ghani's notion of second-order functor differs slightly from Hinze's.[3] Ideally, their notion would be captured by the following class declaration:

$$\begin{array}{l}\textbf{class}\ (\forall f.\mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (h\ f)) \Rightarrow \mathit{HFunctor}\ h\ \textbf{where}\\ \quad \mathit{hfmap} :: (\mathit{Functor}\ f, \mathit{Functor}\ g)\\ \qquad\quad \Rightarrow (\forall x.f\ x \to g\ x) \to (\forall x.h\ f\ x \to h\ g\ x)\end{array}$$

Like in Hinze's case, the quantified class constraint expresses that a second-order functor takes first-order functors to first-order functors. Additionally, second-order functors provide a second-order *fmap*, called *hfmap*, which replaces replaces f by g, to take values of type $h\ f\ x$ to type $h\ g\ x$. Yet, in the absence of actual support for quantified class constraints, Johann and Ghani provide the following declaration instead:

$$\begin{array}{l}\textbf{class}\ \mathit{HFunctor}\ h\ \textbf{where}\\ \quad \mathit{ffmap} :: \mathit{Functor}\ f \Rightarrow (a \to b) \to (h\ f\ a \to h\ f\ b)\\ \quad \mathit{hfmap} :: (\mathit{Functor}\ f, \mathit{Functor}\ g)\\ \qquad\quad \Rightarrow (\forall x.f\ x \to g\ x) \to (\forall x.h\ f\ x \to h\ g\ x)\end{array}$$

In essence, they inline the *fmap* method provided by the quantified class constraint in the *HFunctor* class. This is unfortunate because it duplicates the *Functor* class's functionality.

2.2 Terminating Corecursive Resolution

Quantified class constraints were first proposed by Hinze and Peyton Jones [12] as a solution to a problem of diverging type class resolution. Consider their generalized rose tree datatype

$$\textbf{data}\ \mathit{GRose}\ f\ a = \mathit{GBranch}\ a\ (f\ (\mathit{GRose}\ f\ a))$$

and its *Show* instance

$$\begin{array}{l}\textbf{instance}\ (\mathit{Show}\ a, \mathit{Show}\ (f\ (\mathit{GRose}\ f\ a))) \Rightarrow \mathit{Show}\ (\mathit{GRose}\ f\ a)\\ \quad \textbf{where}\ \mathit{show}\ (\mathit{GBranch}\ x\ xs) = \mathit{unwords}\ [\mathit{show}\ x, \texttt{"-"}, \mathit{show}\ xs]\end{array}$$

Notice the two constraints in the instance context which are due to the two *show* invocations in the method definition. Standard recursive type class resolution would diverge when faced with the constraint $(\mathit{Show}\ (\mathit{GRose}\ []\ \mathit{Bool}))$. Indeed, it would recursively resolve the instance context: *Show Bool* is easily dismissed, but $\mathit{Show}\ [\mathit{GRose}\ []\ a]$ requires resolving $\mathit{Show}\ (\mathit{GRose}\ []\ \mathit{Bool})$ again. Clearly this process loops.

To solve this problem, Hinze and Peyton Jones proposed to write the *GRose* instance with a quantified type class constraint as:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Show}\ a, \forall x.\mathit{Show}\ x \Rightarrow \mathit{Show}\ (f\ x)) \Rightarrow \mathit{Show}\ (\mathit{GRose}\ f\ a)\\ \quad \textbf{where}\ \mathit{show}\ (\mathit{GBranch}\ x\ xs) = \mathit{unwords}\ [\mathit{show}\ x, \texttt{"-"}, \mathit{show}\ xs]\end{array}$$

This would avoid the diverging loop in the type system extension they sketch, because the two recursive resolvents, *Show Bool* and $\forall x.\mathit{Show}\ x \Rightarrow \mathit{Show}\ [x]$ are readily discharged with the available *Bool* and $[a]$ instances.

When faced with the same looping issue in their *Scrap Your Boilerplate* work, Lämmel and Peyton Jones [22] implemented a

[3] It is more in line with the category theoretical notion of endofunctors over the category of endofunctors.

different solution: *cycle-aware constraint resolution.* This approach detects that a recursive resolvent is identical to one of its ancestors and then ties the (co-)recursive knot at the level of the underlying type class dictionaries.

Unfortunately, cycle-aware resolution is not a panacea. It only deals with a particular class of diverging resolutions, those that cycle. The fixpoint of the second-order functor *HPerf* presented above is beyond its capabilities.

```
instance (Show (h (Mu h) a)) ⇒ Show (Mu h a) where
  show (In x) = show x

instance (Show a, Show (f (a,a))) ⇒ Show (HPerf f a) where
  show (HZero a)  = "(Z " ++ show a ++ ")"
  show (HSucc xs) = "(S " ++ show xs ++ ")"
```

Resolving *Show* (*Mu HPerf Int*) diverges without cycling back to the original constraint due to the nestedness of the perfect tree type:

$$\begin{array}{ll}
 & \underline{Show\ (Mu\ HPerf\ Int)} \\
\rightarrowtail & Show\ (HPerf\ (Mu\ HPerf)\ Int) \\
\rightarrowtail & Show\ Int, \underline{Show\ (Mu\ HPerf\ (Int, Int))} \\
\rightarrowtail & Show\ (HPerf\ (Mu\ HPerf)\ (Int, Int)) \\
\rightarrowtail & Show\ (Int, Int), \underline{Show\ (Mu\ HPerf\ ((Int, Int), (Int, Int)))} \\
\rightarrowtail & \ldots
\end{array}$$

In contrast, with quantified type class constraints we can formulate the instances in a way that resolution does terminate.

```
instance (Show a,
    ∀f x.(Show x, ∀y.Show y ⇒ Show (f y)) ⇒ Show (h f x))
    ⇒ Show (Mu h a) where show (In x) = show x

instance (Show a, ∀x.Show x ⇒ Show (f x)) ⇒ Show (HPerf f a)
  where show (HZero a) = "(Z " ++ show a ++ ")"
        show (HSucc xs)= "(S " ++ show xs ++ ")"
```

2.3 Summary

In summary, quantified type class constraints enable 1) expressing more of a type class's specification in a natural and succinct manner, and 2) terminating type class resolution for a larger group of applications.

In the remainder of this paper we provide a declarative type system for a Haskell-like calculus with quantified class constraints (Section 3). Type inference is shown in Section 4 and Section 5 provides an elaboration into System F. Section 6 presents the conditions we require to ensure termination in the presence of quantified class constraints. Finally, Section 7 discusses related work and Section 8 concludes.

3 Declarative Type System

This section provides the declarative type system specification for our core Haskell calculus with quantified class constraints.

3.1 Syntax

Figure 1 presents the, mostly standard, syntax of our source language. A program *pgm* consists of class declarations *cls*, instance declarations *inst* and a top-level expression e. For simplicity, each class has a single parameter and a single method.

Terms e comprise a λ-calculus extended with let-bindings. By convention, we use f to denote a method name and x, y, z to denote any kind of term variable name.

x, y, z, f	::=	⟨*term variable name*⟩	
a, b, c	::=	⟨*type variable name*⟩	
TC	::=	⟨*class name*⟩	
pgm	::=	$e \mid cls; pgm \mid inst; pgm$	*program*
cls	::=	**class** $A \Rightarrow TC\ a$ **where** $\{ f :: \sigma \}$	*class decl.*
$inst$	::=	**instance** $A \Rightarrow TC\ \tau$ **where** $\{ f = e \}$	*instance decl.*
e	::=	$x \mid \lambda x.e \mid e_1\ e_2 \mid$ **let** $x = e_1$ **in** e_2	*term*
τ	::=	$a \mid \tau_1 \rightarrow \tau_2$	*monotype*
ρ	::=	$\tau \mid C \Rightarrow \rho$	*qualified type*
σ	::=	$\rho \mid \forall a.\sigma$	*type scheme*
A	::=	$\bullet \mid A, C$	*axiom set*
C	::=	$Q \mid C_1 \Rightarrow C_2 \mid \forall a.C$	*constraint*
Q	::=	$TC\ \tau$	*class constraint*
Γ	::=	$\bullet \mid \Gamma, x : \sigma \mid \Gamma, a$	*typing environment*
P	::=	$\langle A_S, A_I, A_L \rangle$	*program theory*

Figure 1. Source Syntax

Types also appear in Figure 1. Like all extensions of the Damas-Milner system [6] with qualified types [14], we discriminate between monotypes τ, qualified types ρ and type schemes σ. Note that, to avoid clutter, our formalization does not feature higher-kinded types, but our prototype implementation does.

Our calculus differs from Haskell'98 in that it conservatively generalizes the language of constraints. In Haskell'98 the constraints that can appear in type signatures and in class and instance contexts are basic class constraints Q of the form $TC\,\tau$. As a consequence, the constraint schemes or axioms that are derived from instances (and for superclasses) are Horn clauses of the form:

$$\forall \bar{a}.Q_1 \wedge \ldots \wedge Q_n \Rightarrow Q_0$$

These axioms are similar to rank-1 polymorphic types in the sense that the quantifiers (and the implication) only occur on the outside. We allow a more general form of constraints C where, in analogy with higher-rank types, quantifiers and implications occur in nested positions. This more expressive form of constraints can occur in signatures and class/instance contexts. Consequently, the syntactic sort C of constraints and axioms is one and the same.

Note that constraint schemes of the form $\forall \bar{a}.(Q_1 \wedge \ldots \wedge Q_n) \Rightarrow Q_0$, used in earlier formalizations of type classes (e.g., [25]), are not valid syntax for our constraints C because we do not provide a notation for conjunction. Yet, we can easily see the scheme notation as syntactic sugar for a curried representation:

$$\forall \bar{a}.(Q_1 \wedge \cdots \wedge Q_n) \Rightarrow Q_0 \quad \equiv \quad \forall \bar{a}.Q_1 \Rightarrow (\ldots (Q_n \Rightarrow Q_0) \ldots)$$

We denote a list of C-constraints as A, short for *axiom set* as we use them to represent, among others, axioms given through type class instances.

Finally, Figure 1 presents typing environments Γ, which are entirely standard, and the program theory P. The latter is a triple of three axiom sets: the superclass axioms A_S, the instance axioms A_I and local axioms A_L. We use the notation $P,_{\text{L}}\,C$ to denote that we extend the local component of the triple, and similar notation

$\boxed{P;\Gamma \vdash_{\mathtt{tm}} e : \sigma}$ Term Typing

$$\frac{(x:\sigma)\in\Gamma}{P;\Gamma \vdash_{\mathtt{tm}} x:\sigma}\ \textsc{TmVar} \qquad \frac{P;\Gamma,x:\tau \vdash_{\mathtt{tm}} e_1:\tau \qquad P;\Gamma,x:\tau \vdash_{\mathtt{tm}} e_2:\sigma}{P;\Gamma \vdash_{\mathtt{tm}} (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2):\sigma}\ \textsc{TmLet} \qquad \frac{P;\Gamma,a \vdash_{\mathtt{tm}} e:\sigma}{P;\Gamma \vdash_{\mathtt{tm}} e:\forall a.\sigma}\ (\forall\mathrm{I})$$

$$\frac{\Gamma \vdash_{\mathtt{ty}} \tau_1 \qquad P;\Gamma,x:\tau_1 \vdash_{\mathtt{tm}} e:\tau_2}{P;\Gamma \vdash_{\mathtt{tm}} \lambda x.e:\tau_1\to\tau_2}\ (\to\mathrm{I}) \qquad \frac{P;\Gamma \vdash_{\mathtt{tm}} e_1:\tau_1\to\tau_2 \qquad P;\Gamma \vdash_{\mathtt{tm}} e_2:\tau_1}{P;\Gamma \vdash_{\mathtt{tm}} e_1\ e_2:\tau_2}\ (\to\mathrm{E}) \qquad \frac{\Gamma \vdash_{\mathtt{ct}} C \qquad P,_{\mathrm{L}} C;\Gamma \vdash_{\mathtt{tm}} e:\rho}{P;\Gamma \vdash_{\mathtt{tm}} e:C\Rightarrow\rho}\ (\Rightarrow\mathrm{I}) \qquad \frac{P;\Gamma \vdash_{\mathtt{tm}} e:C\Rightarrow\rho \qquad P;\Gamma\models C}{P;\Gamma \vdash_{\mathtt{tm}} e:\rho}\ (\Rightarrow\mathrm{E}) \qquad \frac{P;\Gamma \vdash_{\mathtt{tm}} e:\forall a.\sigma \qquad \Gamma \vdash_{\mathtt{ty}} \tau}{P;\Gamma \vdash_{\mathtt{tm}} e:[a\mapsto\tau]\sigma}\ (\forall\mathrm{E})$$

$\boxed{\Gamma \vdash_{\mathtt{cls}} cls : A_S;\Gamma_c}$ Class Declaration Typing

$$\frac{\Gamma,a \vdash_{\mathtt{ct}} C_i \qquad \Gamma,a \vdash_{\mathtt{ty}} \sigma}{\Gamma \vdash_{\mathtt{cls}} \mathbf{class}\ (C_1,\ldots,C_n)\Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \} : [\overline{\forall a.TC\ a\Rightarrow C_i}];[f:\forall a.TC\ a\Rightarrow\sigma]}\ \textsc{Class}$$

$\boxed{P;\Gamma \vdash_{\mathtt{inst}} inst : A_I}$ Class Instance Typing

$$\frac{\begin{array}{c}\overline{b} = fv(\tau) \qquad \Gamma,\overline{b} \vdash_{\mathtt{ax}} A \\ \mathbf{class}\ (C_1,\ldots,C_n)\Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \} \qquad P,_{\mathrm{L}} A;\Gamma,\overline{b}\models[\tau/a]C_i \qquad P,_{\mathrm{L}} A,_{\mathrm{L}} TC\ \tau;\Gamma,\overline{b} \vdash_{\mathtt{tm}} e:[\tau/a]\sigma\end{array}}{P;\Gamma \vdash_{\mathtt{inst}} \mathbf{instance}\ A\Rightarrow TC\ \tau\ \mathbf{where}\ \{\ f = e\ \} : [\forall\overline{b}.A\Rightarrow TC\ \tau]}\ \textsc{Instance}$$

Figure 2. Declarative Type System (Selected Rules)

for the other components. In earlier type class formalizations these separate kinds of axioms are typically conflated into a single axiom set. However, in this paper it is convenient to distinguish them for accurately stating the different restrictions imposed on them. Moreover, it is instructive for contrasting with regular Haskell. In our setting, all three components support the same general form of axioms. In contrast, in Haskell, the local constraints are basic type class constraints Q only, while the instance and superclass axioms have the more expressive Horn clause form.

3.2 The Type System

Figure 2 presents the main judgments of our declarative type system for the language of Figure 1, namely term typing and typing of class and instance declarations.

Type & Constraint Well-Scopedness The judgments for well-scopeness of types, constraints and axiom sets are denoted $\Gamma \vdash_{\mathtt{ty}} \sigma$, $\Gamma \vdash_{\mathtt{ct}} C$ and $\Gamma \vdash_{\mathtt{ax}} A$ respectively. Their definitions are straightforward and can be found in Appendix A.

Term Typing Term typing takes the form $P;\Gamma \vdash_{\mathtt{tm}} e : \sigma$ and can be read as *"under program theory P and typing environment Γ, expression e has type σ"*. The rules are almost literally those of Chakravarty et al. [4]. There are only two differences, which are simplifications for the sake of convenience. Firstly we adopt the Barendregt convention [2], that variables in binders are distinct, throughout this paper. This allows us to omit explicit freshness conditions. Secondly, following Vytiniotis et al. [37] we have opted for recursive let-bindings that are not generalized.

Apart from that, there are no noticeable differences with conventional Haskell in the typing rules. All the interesting differences are concentrated in the definition of the constraint entailment judgment $P;\Gamma \models C$, which is used in the constraint elimination Rule ($\Rightarrow$E). The definition of this auxiliary judgment is discussed in detail in Section 3.3.

Class Declaration Typing Typing for class declarations takes the form $\Gamma \vdash_{\mathtt{cls}} cls : A_S;\Gamma_c$ and is given by Rule Class, presented in Figure 2.

In addition to checking the well-formedness of the method type, we ensure that the class context $(C_1,\ldots,C_n)$ is also well-formed, extending the environment with the local variable a. In turn, this implies that $fv(C_i) \subseteq \{a\}$, in line with the Haskell standard.

As usual, typing a class declaration extends the typing environment with the method typing, and the program's theory with the superclass axioms. For instance, the extended monad transformer class yields the superclass axiom:

$$\forall t.Trans\ t \Rightarrow (\forall m.Monad\ m \Rightarrow Monad\ (t\ m))$$

Class Instance Typing Instance typing takes the form $P;\Gamma \vdash_{\mathtt{inst}} inst : A_I$ and is given by Rule Instance, also presented in Figure 2.

We check the well-formedness of the instance context A under the extended typing environment, and that each superclass constraint C_i is entailed by the instance context.

Finally, we check that the method implementation e has the type indicated by the class declaration, appropriately instantiated for the instance in question.

Program Typing The judgment for program typing ties everything together and takes the form $P;\Gamma \vdash_{\mathtt{pgm}} pgm : \sigma$. Its definition is straightforward and can be found in Appendix A.

3.3 Constraint Entailment

Following the approach of Schrijvers et al. [32] for their Cochis calculus, we present constraint entailment in two steps. First, we provide an easy-to-understand and expressive, yet also highly ambiguous, specification. Then we present a syntax-directed, semi-algorithmic variant that takes the ambiguity away, but has a more complicated formulation inspired by the *focusing* technique used in proof search [1, 23, 24].

Declarative Specification Constraint entailment takes the form $P;\Gamma \models C$, and its high-level declarative specification is given by the following rules:

$$\frac{C \in P}{P; \Gamma \models C}\ (\textsc{SpecC}) \qquad \frac{P; \Gamma, a \models C}{P; \Gamma \models \forall a.C}\ (\forall\text{IC}) \qquad \frac{P; \Gamma \models \forall a.C \quad \Gamma \vdash_{ty} \tau}{P; \Gamma \models [\tau/a]C}\ (\forall\text{EC})$$

$$\frac{P,_{\text{L}} C_1; \Gamma \models C_2}{P; \Gamma \models C_1 \Rightarrow C_2}\ (\Rightarrow\text{IC}) \qquad \frac{P; \Gamma \models C_1 \Rightarrow C_2 \quad P; \Gamma \models C_1}{P; \Gamma \models C_2}\ (\Rightarrow\text{EC})$$

If we interpret constraints C as logical formulas, the above rules are nothing more than the rules of first-order predicate logic. Rule (SpecC) is the standard axiom rule. Rules (⇒IC) and (⇒EC) correspond to implication introduction and elimination, respectively. Similarly, Rules (∀IC) and (∀EC) correspond to introduction and elimination of universal quantification, respectively. These are also essentially the rules Hinze and Peyton Jones [12] propose.

While compact and elegant, there is a serious downside to these rules: They are highly ambiguous and give rise to many trivially different proofs for the same constraint. For instance, assuming $\Gamma = \bullet, a$ and $P = \langle \bullet, \bullet, Eq\,a \rangle$, here are only two of the infinitely many proofs of $P; \Gamma \models Eq\,a$:

$$\frac{Eq\,a \in P}{P; \Gamma \models Eq\,a}\ (\textsc{SpecC})$$

versus

$$\frac{\dfrac{\dfrac{Eq\,a \in P'}{P'; \Gamma \models Eq\,a}\ (\textsc{SpecC})}{P; \Gamma \models Eq\,a \Rightarrow Eq\,a}\ (\Rightarrow\text{IC}) \qquad \dfrac{Eq\,a \in P}{P; \Gamma \models Eq\,a}\ (\textsc{SpecC})}{P; \Gamma \models Eq\,a}\ (\Rightarrow\text{EC})$$

where $P' = P,_{\text{L}} Eq\,a$. Observe that the latter proof makes an unnecessary appeal to implication introduction.

Type-Directed Specification To avoid the trivial forms of ambiguity like in the example, we adopt a solution from proof search known as *focusing* [1]. This solution was already adopted by the Cochis calculus, for the same reason. The key idea of focusing is to provide a syntax-directed definition of constraint entailment where only one inference rule applies at any given time.

Figure 3 presents our definition of constraint entailment with focusing. The main judgment $P; \Gamma \models C$ is defined in terms of two auxiliary judgments, $P; \Gamma \models [C]$ and $\Gamma; [C] \models Q \rightsquigarrow A$, each of which is defined by structural induction on the constraint enclosed in square brackets.

The main entailment judgment is equivalent to the first auxiliary judgment $P; \Gamma \models [C]$. This auxiliary judgment focuses on the constraint C whose entailment is checked – we call this constraint the "goal". There are three rules, for the three possible syntactic forms of C. Rules (⇒R) and (∀R) decompose the goal by applying implication and quantifier introductions respectively. Once the goal is stripped down to a simple class constraint Q, Rule (QR) selects an axiom C from the theory P to discharge it. The selected axiom must *match* the goal, a notion that is captured by the second auxiliary judgment. Matching gives rise to a sequence A of new (and hopefully simpler) goals whose entailment is checked recursively.

The second auxiliary judgment $\Gamma; [C] \models Q \rightsquigarrow A$ focuses on the axiom C and checks whether it matches the simple goal Q. Again, there are three rules for the three possible forms the axiom can take. Rule (QL) expresses the base case where the axiom is identical to the

$\boxed{P; \Gamma \models C}$ Constraint Entailment

$$\frac{P; \Gamma \models [C]}{P; \Gamma \models C}$$

$\boxed{P; \Gamma \models [C]}$ Constraint Resolution

$$\frac{P,_{\text{L}} C_1; \Gamma \models [C_2]}{P; \Gamma \models [C_1 \Rightarrow C_2]}\ (\Rightarrow\text{R}) \qquad \frac{P; \Gamma, b \models [C]}{P; \Gamma \models [\forall b.C]}\ (\forall\text{R})$$

$$\frac{C \in P:\ \Gamma; [C] \models Q \rightsquigarrow A \qquad \forall C_i \in A:\ P; \Gamma \models [C_i]}{P; \Gamma \models [Q]}\ (\text{QR})$$

$\boxed{\Gamma; [C] \models Q \rightsquigarrow A}$ Constraint Matching

$$\frac{\Gamma; [C_2] \models Q \rightsquigarrow A}{\Gamma; [C_1 \Rightarrow C_2] \models Q \rightsquigarrow A, C_1}\ (\Rightarrow\text{L})$$

$$\frac{\Gamma; [[\tau/b]C] \models Q \rightsquigarrow A \qquad \Gamma \vdash_{ty} \tau}{\Gamma; [\forall b.C] \models Q \rightsquigarrow A}\ (\forall\text{L}) \qquad \frac{}{\Gamma; [Q] \models Q \rightsquigarrow \bullet}\ (\text{QL})$$

Figure 3. Tractable Constraint Entailment

goal and there are no new goals. Rule (⇒L) handles an implication axiom $C_1 \Rightarrow C_2$ by recursively checking whether C_2 matches the goal. At the same time it yields a new goal C_1 which needs to be entailed in order for the axiom to apply. Finally, Rule (∀L) handles universal quantification by instantiating the quantified variable in a way that recursively yields a match.

It is not difficult to see that this type-directed formulation of entailment greatly reduces the number of proofs for given goal.[4] For instance, for the example above there is only one proof:

$$\frac{\dfrac{Eq\,a \in P \qquad \Gamma; [Eq\,a] \models Eq\,a \rightsquigarrow \bullet\ (\text{QL})}{P; \Gamma \models [Eq\,a]}\ (\text{QR})}{P; \Gamma \models Eq\,a}$$

3.4 Remaining Nondeterminism

While focusing makes the definition of constraint entailment type-directed, there are still two sources of nondeterminism. As a consequence, the specification is still ambiguous and not an algorithm.

Overlapping Axioms The first source of non-determinism is that in Rule (QR) there may be multiple matching axioms that make the entailment go through. For applications of logic where proofs are irrelevant this is not a problem, but in Haskell where the proofs have computational content (namely the method implementations) this is a cause for concern. Haskell'98 also faces this problem. Consider two instances for the same type:

class *Default a* **where** { *default* :: *a* }

instance *Default Bool* **where** { *default* = *True* }
instance *Default Bool* **where** { *default* = *False* }

The two instances give rise to two different proofs for *Default Bool*, with distinct computational content (*True* vs. *False*). We steer away from this problem in the same was as Haskell'98, by requiring that instance declarations do not overlap. This does not rule out the possibility of distinct proofs for the same goal, but at least distinct

[4] Without loss of expressive power. See for example [30].

proofs have the same computational content. Consider a class hierarchy where C is the superclass of both D and E.

$$\begin{array}{l}\textbf{class}\ C\ a\ \textbf{where}\ \{\ \ldots\ \}\\ \textbf{class}\ C\ a \Rightarrow D\ a\ \textbf{where}\ \{\ \ldots\ \}\\ \textbf{class}\ C\ a \Rightarrow E\ a\ \textbf{where}\ \{\ \ldots\ \}\end{array}$$

This gives rise to the superclass axioms $\forall a.D\ a \Rightarrow C\ a$ and $\forall a.E\ a \Rightarrow C\ a$. Given additionally two local constraints $D\ \tau$ and $E\ \tau$, we have two ways to establish $C\ \tau$. The proofs are distinct, yet ultimately the computational content is the same. This is easy to see as only instances supply the computational content and there can be at most one instance for any given type τ.

In summary, non-overlap of instances is sufficient to ensure *coherence*.

Guessing Polymorphic Instantiation A second source of ambiguity is that Rule ($\forall$L) requires guessing an appropriate type τ for substituting the type variable b. Guessing is problematic because there are an infinite number of types to choose from and more than one of those choices can make the entailment work out. Choosing an appropriate type is a problem for the type inference algorithm in the next section. Different choices leading to different proofs is a more fundamental problem that also manifests itself in Haskell'98. Consider the following instances.

$$\begin{array}{l}\textbf{instance}\ C\ \mathit{Char}\ \textbf{where}\ \{\ \ldots\ \}\\ \textbf{instance}\ C\ \mathit{Bool}\ \textbf{where}\ \{\ \ldots\ \}\\ \textbf{instance}\ C\ a \Rightarrow D\ \mathit{Int}\ \textbf{where}\ \{\ \ldots\ \}\end{array}$$

The third instance gives rise to the axiom $\forall a.C\ a \Rightarrow D\ \mathit{Int}$. When resolving $D\ \mathit{Int}$ with this axiom we can choose a to be either Char or Bool and thus select a different C instance.

Haskell'98 avoids this problem by requiring that all quantified type variables, like a in the example, appear in the head of the axiom. Because our axioms have a more general, recursively nested form, we generalize this requirement in a recursively nested fashion. The predicate $unamb(C)$ in Figure 4 formalizes the requirement in terms of the auxiliary judgment $\overline{a} \vdash_{\mathsf{unamb}} C$, where $\overline{a}$ are type variables that need to be determined by the head of C. Rule (QU) constitutes the base case where Q is the head and contains the determinable type variables $\overline{a}$. Rule ($\forall$U) processes a quantifier by adding the new type variable to the list of determinable type variables $\overline{a}$. Finally, Rule ($\Rightarrow$U) checks whether the head C_2 of the implication determines the type variables $\overline{a}$. It also recursively checks whether C_1 is unambiguous on its own. The latter check is necessary because left-hand sides of implications are themselves added as axioms to the theory in Rule ($\Rightarrow$R); hence they must be well-behaved on their own.

The predicate $unamb(C)$ must be imposed on all constraints that are added to the theory. This happens in four places: the instance axioms added in Rule Instance, the superclass axioms added in Rule Class, the local axioms added when checking against a given signature in Rule ($\Rightarrow$I) and the local axioms added during constraint entailment checking in Rule ($\Rightarrow$R). These four places can be traced back to three places in the syntax: class and instance heads, and (method) signatures.

4 Type Inference

We provide a type inference algorithm with elaboration into System F [8]. To simplify the presentation, this section focuses solely on

$\boxed{unamb(C)}$ Unambiguity

$$\frac{\bullet \vdash_{\mathsf{unamb}} C}{unamb(C)}\ \textsc{Unamb}$$

$\boxed{\overline{a} \vdash_{\mathsf{unamb}} C}$ Unambiguity

$$\frac{\overline{a} \subseteq fv(Q)}{\overline{a} \vdash_{\mathsf{unamb}} Q}\ (\mathrm{QU}) \qquad \frac{\overline{a}, a \vdash_{\mathsf{unamb}} C}{\overline{a} \vdash_{\mathsf{unamb}} \forall a.C}\ (\forall\mathrm{U}) \qquad \frac{unamb(C_1) \quad \overline{a} \vdash_{\mathsf{unamb}} C_2}{\overline{a} \vdash_{\mathsf{unamb}} C_1 \Rightarrow C_2}\ (\Rightarrow\mathrm{U})$$

Figure 4. Unambiguity

type inference. The parts of the rules highlighted in gray concern elaboration and are discussed in Section 5.

To make the connection to the relations of the declarative specification (Section 3.2) more clear, corresponding rules share the same name.

4.1 Preliminaries

Before diving into the details of the algorithm, we first introduce some additional notation and constructs.

Variable-Annotated Constraints & Type Equalities Since our goal is to perform type inference and elaboration to System F simultaneously, we annotate all constraints with their corresponding System F evidence term (dictionary variable d). We keep the notational burden minimal by reusing the same letters as in Figure 1, yet with a calligraphic font:

$$\begin{array}{lcll}\mathcal{P} & ::= & \langle \mathcal{A}_S, \mathcal{A}_I, \mathcal{A}_L \rangle & \textit{variable-annotated theory}\\ \mathcal{A} & ::= & \bullet \mid \mathcal{A}, \mathcal{C} & \textit{variable-annotated axiom set}\\ \mathcal{C} & ::= & d : C & \textit{variable-annotated constraint}\\ \mathcal{Q} & ::= & d : Q & \textit{variable-annotated class constraint}\end{array}$$

Additionally, like every HM(X)-based system, our type-inference algorithm proceeds by first generating type constraints from the program text (constraint generation) and then solving these constraints independently of the program text (constraint solving).

During constraint generation, our algorithm gives rise to both (variable-annotated) constraints $\mathcal{A}$, as well as type equalities E:

$$\begin{array}{lcll}E & ::= & \bullet \mid E, \tau_1 \sim \tau_2 & \textit{type equalities}\end{array}$$

Type & Evidence Substitutions Furthermore, we introduce two kinds of substitutions: type substitutions θ and dictionary substitutions η:

$$\begin{array}{lcll}\theta & ::= & \bullet \mid \theta \cdot [\tau/a] & \textit{type substitution}\\ \eta & ::= & \bullet \mid \eta \cdot [t/d] & \textit{evidence substitution}\end{array}$$

A type substitution θ maps type variables to monotypes, while an evidence substitution η maps dictionary variables d to System F terms t (see Section 5.1 for the formal syntax of System F terms).

4.2 Constraint Generation For Terms

Figure 5 presents constraint generation for terms. The relation takes the form $\Gamma \vdash_{\mathsf{tm}} e : \tau \rightsquigarrow t \mid \mathcal{A}; E$. Given a typing environment Γ and a term e we infer (1) a monotype τ, (2) a set of wanted constraints $\mathcal{A}$, and (3) a set of wanted equalities E. Its definition is standard.

Rule TmVar handles variables. We instantiate the polymorphic type $\forall \overline{a}.\overline{C} \Rightarrow \tau$ of a term variable x with fresh unification variables $\overline{b}$, introducing $\overline{C}$ as wanted constraints, instantiated likewise. Rule TmAbs assigns a fresh unification variable to the abstracted

$\boxed{\Gamma \vdash_{\mathsf{tm}} e : \tau \rightsquigarrow t \mid \mathcal{A}; E}$ Term Typing

$$\frac{\overline{b}, \overline{d} \text{ fresh} \qquad (x : \forall \overline{a}. \overline{C} \Rightarrow \tau) \in \Gamma}{\Gamma \vdash_{\mathsf{tm}} x : [\overline{b}/\overline{a}]\tau \rightsquigarrow x\ \overline{b}\ \overline{d} \mid \overline{(d : [\overline{b}/\overline{a}]C)}; \bullet}\ \text{TmVar}$$

$$\frac{a \text{ fresh} \qquad \Gamma, x : a \vdash_{\mathsf{tm}} e_1 : \tau_1 \rightsquigarrow t_1 \mid \mathcal{A}_1; E_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathsf{tm}} e_2 : \tau_2 \rightsquigarrow t_2 \mid \mathcal{A}_2; E_2 \qquad \vdash_{\mathsf{ty}} \tau_1 \rightsquigarrow \upsilon_1}{\Gamma \vdash_{\mathsf{tm}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \rightsquigarrow \mathbf{let}\ x : \upsilon_1 = t_1\ \mathbf{in}\ t_2 \mid (\mathcal{A}_1, \mathcal{A}_2); (E_1, E_2, a \sim \tau_1)}\ \text{TmLet}$$

$$\frac{a \text{ fresh} \qquad \Gamma, x : a \vdash_{\mathsf{tm}} e : \tau \rightsquigarrow t \mid \mathcal{A}; E}{\Gamma \vdash_{\mathsf{tm}} \lambda x.e : a \to \tau \rightsquigarrow \lambda(x : a).t \mid \mathcal{A}; E}\ \text{TmAbs}$$

$$\frac{a \text{ fresh} \qquad \Gamma \vdash_{\mathsf{tm}} e_1 : \tau_1 \rightsquigarrow t_1 \mid \mathcal{A}_1; E_1 \qquad \Gamma \vdash_{\mathsf{tm}} e_2 : \tau_2 \rightsquigarrow t_2 \mid \mathcal{A}_2; E_2}{\Gamma \vdash_{\mathsf{tm}} e_1\ e_2 : a \rightsquigarrow t_1\ t_2 \mid (\mathcal{A}_1, \mathcal{A}_2); (E_1, E_2, \tau_1 \sim \tau_2 \to a)}\ \text{TmApp}$$

Figure 5. Constraint Generation for Terms with Elaboration

$\boxed{\overline{a}; \mathcal{P} \models \mathcal{A}_1 \rightsquigarrow \mathcal{A}_2; \eta}$ Constraint Solving Algorithm

$$\frac{\nexists C \in \mathcal{A}_1 :\ \overline{a}; \mathcal{P} \models [C] \rightsquigarrow \mathcal{A}_2 ; \eta}{\overline{a}; \mathcal{P} \models \mathcal{A}_1 \rightsquigarrow \mathcal{A}_1; \bullet}\ \text{Stop}$$

$$\frac{\overline{a}; \mathcal{P} \models [C] \rightsquigarrow \mathcal{A}_2 ; \eta_1 \qquad \overline{a}; \mathcal{P} \models \mathcal{A}_1, \mathcal{A}_2 \rightsquigarrow \mathcal{A}_3; \eta_2}{\overline{a}; \mathcal{P} \models \mathcal{A}_1, C \rightsquigarrow \mathcal{A}_3; (\eta_2 \cdot \eta_1)}\ \text{Step}$$

$\boxed{\overline{a}; \mathcal{P} \models [C] \rightsquigarrow \mathcal{A} ; \eta}$ Constraint Simplification

$$\frac{\vdash_{\mathsf{ct}} C_1 \rightsquigarrow \upsilon_1 \qquad \overline{a}; \mathcal{P},_{\mathsf{L}} (d_1 : C_1) \models [d_2 : C_2] \rightsquigarrow \overline{(d : C)} ; \eta \qquad \overline{d}', d_1, d_2 \text{ fresh} \qquad \eta' = [\lambda(d_1 : \upsilon_1).[\overline{d'\ d_1/d}](\eta(d_2))/d_0]}{\overline{a}; \mathcal{P} \models [d_0 : C_1 \Rightarrow C_2] \rightsquigarrow \overline{(d' : C_1 \Rightarrow C)} ; \eta'}\ (\Rightarrow\text{R})$$

$$\frac{\overline{d}', d_C \text{ fresh} \qquad \overline{a}, b; \mathcal{P} \models [d_C : C_0] \rightsquigarrow \overline{(d : C)} ; \eta \qquad \eta' = [\Lambda b.[\overline{d'\ b/d}](\eta(d_C))/d_0]}{\overline{a}; \mathcal{P} \models [d_0 : \forall b.C_0] \rightsquigarrow \overline{(d' : \forall b.C)} ; \eta'}\ (\forall\text{R})$$

$$\frac{C \in \mathcal{P} :\ \overline{a}; [C] \models Q \rightsquigarrow \mathcal{A}; \theta ; \eta}{\overline{a}; \mathcal{P} \models [Q] \rightsquigarrow \mathcal{A} ; \eta}\ (Q\text{R})$$

$\boxed{\overline{a}; [C] \models Q \rightsquigarrow \mathcal{A}; \theta ; \eta}$ Constraint Matching

$$\frac{d_1, d_2 \text{ fresh} \qquad \overline{a}; [d_2 : C_2] \models Q \rightsquigarrow \mathcal{A}; \theta ; \eta}{\overline{a}; [d : C_1 \Rightarrow C_2] \models Q \rightsquigarrow \mathcal{A}, d_1 : \theta(C_1); \theta ; [d\ d_1/d_2] \cdot \eta}\ (\Rightarrow\text{L})$$

$$\frac{d' \text{ fresh} \qquad \overline{a}; [d' : C] \models Q \rightsquigarrow \mathcal{A}; \theta ; \eta}{\overline{a}; [d : \forall b.C] \models Q \rightsquigarrow \mathcal{A}; \theta ; [d\ (\theta(b))/d'] \cdot \eta}\ (\forall\text{L})$$

$$\frac{\theta = unify(\overline{a}; \tau_1 \sim \tau_2)}{\overline{a}; [d' : TC\ \tau_1] \models d : TC\ \tau_2 \rightsquigarrow \bullet; \theta ; [d'/d]}\ (Q\text{L})$$

Figure 6. Constraint Entailment with Dictionary Construction

term variable x, and adds it to the context for checking the body of the abstraction. Rule TmApp handles applications ($e_1\ e_2$). We collect wanted class and equality constraints from each subterm, we generate a fresh type variable a for the result and record that the type of e_1 is a function type ($\tau_1 \sim \tau_2 \to a$). Rule TmLet handles (possibly recursive) let bindings.

4.3 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 by means of unification. The function $unify(\overline{a}; E) = \theta_\perp$ takes the set of equalities and a set of "untouchable" type variables, and returns either the most general unifier θ of the equalities or fails if none exists. The untouchable type variables $\overline{a}$ originate from type signatures; all other type variables are unification variables. The unifier is of course only allowed to substitute unification variables.

The definition of this unification function is folklore, following Damas and Milner [6] and accounting for signatures; it can be found in Appendix A.

Solving Type Class Constraints Figure 6 defines the judgment for solving type class constraints; it takes the form $\overline{a}; \mathcal{P} \models \mathcal{A}_1 \rightsquigarrow \mathcal{A}_2; \eta$. Given a set of untouchable type variables $\overline{a}$ and a theory $\mathcal{P}$, it (exhaustively) replaces a set of constraints $\mathcal{A}_1$ with a set of *simpler*, residual constraints $\mathcal{A}_2$, via the auxiliary judgment $\overline{a}; \mathcal{P} \models [C] \rightsquigarrow \mathcal{A} ; \eta$, explained below.

This form differs from the specification in Figure 3: we allow constraints to be partially entailed, which in turn allows us to perform *simplification* [17] 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 $\forall a. Eq\ a \Rightarrow a \to Bool$.

Simplification Auxiliary judgment $\overline{a}; \mathcal{P} \models [C] \rightsquigarrow \mathcal{A} ; \eta$ uses the theory $\mathcal{P}$ to simplify a single constraint C to a set of simpler constraints without instantiating any of the untouchable type variables $\overline{a}$. Following the focusing approach, the judgment is defined by three rules, one for each of the syntactic forms of the goal C.

Rules ($\Rightarrow$R) and ($\forall$R) recursively simplify the head of the goal. Observe that we add the bound variable b to the untouchables $\overline{a}$ when going under a binder in Rule ($\forall$R). Once the goal is stripped down to a simple class constraint Q, Rule (QR) selects an axiom C whose head matches the goal, and uses it to replace the goal with a set of simpler constraints $\mathcal{A}$ (a process known as *context reduction* [16]). Goal matching is performed by judgment $\overline{a}; [C] \models Q \rightsquigarrow \mathcal{A}; \theta ; \eta$, discussed below.

Matching Auxiliary judgment $\overline{a}; [C] \models Q \rightsquigarrow \mathcal{A}; \theta ; \eta$ focuses on the axiom C and checks whether it matches the simple goal Q. The main difference between this algorithmic relation and its declarative specification in Figure 3 lies in the type substitution θ. Instead of guessing a type for instantiating a polymorphic axiom in Rule ($\forall$L) (top-down), we defer the choice until the head of the axiom is met, in Rule (QL) (bottom-up). Observe that Rule ($\forall$L) does not record b as untouchable, effectively turning it into a unification variable. Thus, by unifying the head of the axiom with the goal we

$$\boxed{\Gamma \vdash_{\mathtt{cls}} cls : \mathcal{A}_S ; \Gamma_c \rightsquigarrow fdata; \overline{fval}} \quad \text{Class Declaration Typing}$$

$$\frac{\begin{array}{c}\Gamma, a \vdash_{\mathtt{ty}} \sigma \qquad \vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon \qquad \Gamma, a \vdash_{\mathtt{ct}} C_i \qquad \vdash_{\mathtt{ct}} C_i \rightsquigarrow \upsilon_i \qquad d, \overline{d}^n \text{ fresh} \qquad fdata = \mathbf{data}\ T_{TC}\ a = K_{TC}\ \overline{\upsilon}^n\ \upsilon \\ fval_1 = \mathbf{let}\ f : (\forall a. T_{TC}\ a \to \upsilon) = \Lambda a. \lambda (d : T_{TC}\ a). proj_{TC}^{n+1}(d) \qquad fval_2^i = \mathbf{let}\ d_i : (\forall a. T_{TC}\ a \to \upsilon_i) = \Lambda a. \lambda (d : T_{TC}\ a). proj_{TC}^{i}(d)\end{array}}{\Gamma \vdash_{\mathtt{cls}} (\mathbf{class}\ (C_1, \ldots, C_n) \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \}) : [\overline{d_i : \forall a. TC\ a \Rightarrow C_i}^n]; [f : \forall a. TC\ a \Rightarrow \sigma] \rightsquigarrow fdata; fval_1, \overline{fval_2}^n} \ \textsc{Class}$$

$$\boxed{\mathcal{P}; \Gamma \vdash_{\mathtt{inst}} inst : \mathcal{A}_I \rightsquigarrow fval} \quad \text{Class Instance Typing}$$

$$\frac{\begin{array}{c}\mathbf{class}\ (C_1', \ldots, C_m') \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \} \qquad \overline{b} = fv(\tau) \qquad \overline{d}, \overline{d}', d_I \text{ fresh} \qquad \mathcal{P}_I = \mathcal{P},_{\mathrm{L}} \overline{d : C} \qquad \Gamma_I = \Gamma, \overline{b} \\ \Gamma_I \vdash_{\mathtt{ct}} C_i \qquad \overline{b}; \mathcal{P}_I,_{\mathrm{L}} (d_I : \forall \overline{b}. \overline{C}^n \Rightarrow TC\ \tau); \Gamma_I \vdash_{\mathtt{tm}} e : [\tau/a]\sigma \rightsquigarrow t \qquad \vdash_{\mathtt{ct}} C_i \rightsquigarrow \upsilon_i \qquad \overline{b}; \mathcal{P}_I \models \overline{d'} : [\tau/a]C' \rightsquigarrow \bullet; \eta\end{array}}{\mathcal{P}; \Gamma \vdash_{\mathtt{inst}} (\mathbf{instance}\ (C_1, \ldots, C_n) \Rightarrow TC\ \tau\ \mathbf{where}\ \{\ f = e\ \}) : [d_I : \forall \overline{b}. \overline{C} \Rightarrow TC\ \tau] \rightsquigarrow \mathbf{let}\ d_I : (\forall \overline{b}. \overline{\upsilon} \to T_{TC}\ \tau) = \Lambda \overline{b}. \lambda \overline{(d : \upsilon)}. K_{TC}\ \tau\ \overline{\eta(d')}\ t} \ \textsc{Instance}$$

Figure 7. Declaration Elaboration

can determine without guessing an instantiation for all top-level quantifiers, captured by the type substitution θ.

As an example, consider the derivation of one-step simplification of $\forall b. Eq\ b \Rightarrow Eq\ [b]$, when $(\forall a. Eq\ a \Rightarrow Eq\ [a]) \in \mathcal{P}$:[5]

$$\cfrac{\cfrac{\cfrac{\cfrac{unify(b; a \sim b) = \theta = [b/a]}{b; [Eq\ [a]] \models Eq\ [b] \rightsquigarrow \bullet; \theta}\ (\mathrm{QL})}{b; [Eq\ a \Rightarrow Eq\ [a]] \models Eq\ [b] \rightsquigarrow Eq\ b; \theta}\ (\Rightarrow\mathrm{L})}{b; [\forall a. Eq\ a \Rightarrow Eq\ [a]] \models Eq\ [b] \rightsquigarrow Eq\ b; \theta}\ (\forall\mathrm{L}) \qquad\qquad}{\cfrac{\cfrac{\cfrac{b; [\forall a. Eq\ a \Rightarrow Eq\ [a]] \models Eq\ [b] \rightsquigarrow Eq\ b; \theta}{b; \mathcal{P}, Eq\ b \models [Eq\ [b]] \rightsquigarrow Eq\ b}\ (\mathrm{QR})}{b; \mathcal{P} \models [Eq\ b \Rightarrow Eq\ [b]] \rightsquigarrow (Eq\ b \Rightarrow Eq\ b)}\ (\Rightarrow\mathrm{R})}{\bullet; \mathcal{P} \models [\forall b. Eq\ b \Rightarrow Eq\ [b]] \rightsquigarrow (\forall b. Eq\ b \Rightarrow Eq\ b)}\ (\forall\mathrm{R})}$$

Search As Section 3.4 has remarked, there may be multiple matching axioms, e.g., due to overlapping superclass axioms. The straightforward algorithmic approach to the involved nondeterminism is search, possibly implemented by backtracking. The GHC Haskell implementation can employ a heuristic to keep this search shallow. It does so by using the superclass constraints very selectively: whenever a new local constraint is added to the theory, it proactively derives all its superclasses and adds them as additional local axioms. When looking for a match, it does not consider the superclass axioms and prefers the local axioms over the instance axioms. If a matching local axiom exists, it immediately discharges the entire goal without further recursive resolution. This is the case because in regular Haskell local axioms are always simple class constraints Q.

In our setting, we can also implement a (modified version) of GHC's heuristic, but this does not obviate the need for deep search. The reason is that our local axioms are not necessarily simple axioms, and matching against them may leave residual goals that require further recursive resolution. When that recursive resolution gets stuck, we have to backtrack over the choice of axiom. Consider the following example.

$$\begin{array}{l}\mathbf{class}\ (E\ a \Rightarrow C\ a) \Rightarrow D\ a \\ \mathbf{class}\ (G\ a \Rightarrow C\ a) \Rightarrow F\ a\end{array}$$

Given local axioms $D\ a$, $F\ a$ and $G\ a$, consider what happens when we resolve the goal $C\ a$. The superclasses $E\ a \Rightarrow C\ a$ and $G\ a \Rightarrow C\ a$ of respectively $D\ a$ and $F\ a$ both match this goal. If we pick the first one, we get stuck when recursively resolving $E\ a$. However, if we backtrack and consider the second one instead, we can recursively resolve $G\ a$ against the given local constraint.

In summary, because we do not see a general way to avoid search, our prototype implementation uses backtracking for choosing between the different axioms.

[5] We omit the evidence substitutions for brevity.

$$\boxed{\overline{a}; \mathcal{P}; \Gamma \vdash_{\mathtt{tm}} e : \sigma \rightsquigarrow t} \quad \text{Explicitly Annotated Term Typing}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{tm}} e : \tau_1 \rightsquigarrow t \mid \mathcal{A}_e; E_e \\ \overline{d} \text{ fresh} \qquad \theta = unify(\overline{a}, \overline{b}; E_e, \tau_1 \sim \tau_2) \\ \vdash_{\mathtt{ct}} C_i \rightsquigarrow \upsilon_i \qquad \overline{a}, \overline{b}; \mathcal{P},_{\mathrm{L}} \overline{d : C} \models \theta(\mathcal{A}_e) \rightsquigarrow \bullet; \eta\end{array}}{\overline{a}; \mathcal{P}; \Gamma \vdash_{\mathtt{tm}} e : (\forall \overline{b}. \overline{C} \Rightarrow \tau_2) \rightsquigarrow \Lambda \overline{b}. \lambda \overline{(d : \upsilon)}. \eta(\theta(t))} \ (\leq)$$

Figure 8. Subsumption Rule

Implementation Our prototype implementation is available at https://github.com/gkaracha/quantcs-impl. It incorporates higher-kinded datatypes and performs type inference, elaboration into System F (as explained in the next section), and type checking of the generated code.

The examples we have tested with the prototype provide confidence that our system is sound and that the elaboration is type preserving. The formal proof of the metatheory is future work.

4.4 Checking Declarations

Figure 7 defines type checking of class and instance declarations.

Class Declaration Typing Typing for class declarations is given by Rule Class. For the purposes of type inference, Rule Class is identical to the corresponding rule of Figure 2, so we defer its analysis to Section 5.5 which discusses elaboration.

Instance Declaration Typing Typing for instance declarations takes the form $\mathcal{P}; \Gamma \vdash_{\mathtt{inst}} inst : \mathcal{A}_I \rightsquigarrow fval$ and is given by Rule Instance. For the most part it is identical to the corresponding rule of Figure 2.

The most notable difference is the handling of the method implementation e: method implementations have their type imposed by the method signature in the class declaration. Hence, we need to *check* rather than *infer* their type.

$$
\begin{array}{llll}
fpgm & ::= & t \mid fval; fpgm \mid fdata; fpgm & \textit{program} \\
fval & ::= & \textbf{let}\ x : \upsilon = t & \textit{value binding} \\
fdata & ::= & \textbf{data}\ T\ a = K\ \overline{\upsilon} & \textit{datatype}
\end{array}
$$

$$
\begin{array}{llll}
t & ::= & x \mid K \mid \lambda(x : \upsilon).t \mid t_1\ t_2 \mid \Lambda a.t \mid t\ \upsilon & \textit{term} \\
 & \mid & \textbf{let}\ x : \upsilon = t_1\ \textbf{in}\ t_2 \mid \textbf{case}\ t_1\ \textbf{of}\ K\ \overline{x} \to t_2 &
\end{array}
$$

$$
\begin{array}{llll}
\upsilon & ::= & a \mid \upsilon_1 \to \upsilon_2 \mid \forall a.\upsilon \mid T\ \upsilon & \textit{type}
\end{array}
$$

Figure 9. System F Syntax

This operation is expressed succinctly by relation $\overline{a}; \mathcal{P}; \Gamma \models_{\mathsf{tm}} e : \sigma \rightsquigarrow t$, presented in Figure 8. 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.

Rule ($\leq$) performs type inference and type subsumption checking simultaneously: First, it infers a monotype τ_1 for expression e, as well as wanted constraints $\mathcal{A}_e$ and type equalities E_e. Type equalities E_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 $\overline{C}$ should completely entail the wanted constraints $\mathcal{A}_e$.

4.5 Program Typing

Type inference and elaboration for programs is straightforward and can be found in Appendix A.

5 Translation to System F

This section discusses the elaboration aspect of the algorithm presented in Section 4.

5.1 Target Language: System F

Syntax The syntax of System F [8] – extended with data types and recursive let-bindings – is presented in Figure 9 and is entirely standard. Like in the source language, we elide all mention of kinds. Without loss of generality, we simplify matters by allowing only data types with a single type parameter and a single data constructor and case expressions with a single branch; this is sufficient for our dictionary-passing translation of type classes.

Semantics & Typing Since the operational semantics and typing for System F with data types are entirely standard and do not contribute to the novelty of this paper, we omit them from our main presentation. They can be found in Appendix B.

5.2 Elaboration of Types & Constraints

Our system follows the traditional approach of translating source type class constraints into explicitly-passed System F terms, the so-called *dictionaries* [9, 38]. This transition is reflected in the translation of types, performed by judgment $\models_{\mathsf{ty}} \sigma \rightsquigarrow \upsilon$:

$$
\frac{}{\models_{\mathsf{ty}} a \rightsquigarrow a}\ \textsc{TyVar}
\qquad
\frac{\models_{\mathsf{ty}} \tau_1 \rightsquigarrow \upsilon_1 \qquad \models_{\mathsf{ty}} \tau_2 \rightsquigarrow \upsilon_2}{\models_{\mathsf{ty}} \tau_1 \to \tau_2 \rightsquigarrow \upsilon_1 \to \upsilon_2}\ \textsc{TyArr}
$$

$$
\frac{\models_{\mathsf{ct}} C \rightsquigarrow \upsilon_1 \qquad \models_{\mathsf{ty}} \rho \rightsquigarrow \upsilon_2}{\models_{\mathsf{ty}} C \Rightarrow \rho \rightsquigarrow \upsilon_1 \to \upsilon_2}\ \textsc{TyQual}
\qquad
\frac{\models_{\mathsf{ty}} \sigma \rightsquigarrow \upsilon}{\models_{\mathsf{ty}} \forall a.\sigma \rightsquigarrow \forall a.\upsilon}\ \textsc{TyAll}
$$

Rules TyVar, TyArr and TyAll are straightforward. Rule TyQual elaborates a qualified type into a System F arrow type: the constraint C is translated into the dictionary type υ_1, via relation $\models_{\mathsf{ct}} C \rightsquigarrow \upsilon$ which performs elaboration of constraints:

$$
\frac{\models_{\mathsf{ty}} \tau \rightsquigarrow \upsilon}{\models_{\mathsf{ct}} TC\ \tau \rightsquigarrow T_{TC}\ \upsilon}\ (\mathrm{C}Q)
\qquad
\frac{\models_{\mathsf{ct}} C \rightsquigarrow \upsilon}{\models_{\mathsf{ct}} \forall a.C \rightsquigarrow \forall a.\upsilon}\ (\mathrm{C}\forall)
$$

$$
\frac{\models_{\mathsf{ct}} C_1 \rightsquigarrow \upsilon_1 \qquad \models_{\mathsf{ct}} C_2 \rightsquigarrow \upsilon_2}{\models_{\mathsf{ct}} C_1 \Rightarrow C_2 \rightsquigarrow \upsilon_1 \to \upsilon_2}\ (\mathrm{C}{\Rightarrow})
$$

Rule (CQ) elaborates a class constraint ($TC\ \tau$) into a type constructor application ($T_{TC}\ \upsilon$), which corresponds to the type of dictionaries that witness ($TC\ \tau$). Rule (C$\forall$) is straightforward. Rule (C$\Rightarrow$) elaborates implication constraints of the form ($C_1 \Rightarrow C_2$) into System F arrow types ($\upsilon_1 \to \upsilon_2$), that is, types of *dictionary transformers*. As a concrete example, the constraint corresponding to the *Show* instance for type *HPerf* (Section 2.2):

$$\forall f\ a.Show\ a \Rightarrow (\forall x.Show\ x \Rightarrow Show\ (f\ x)) \Rightarrow Show\ (HPerf\ f\ a)$$

is elaborated into the type

$$\forall f\ a.T_{Show}\ a \to (\forall x.T_{Show}\ x \to T_{Show}\ (f\ x)) \to T_{Show}\ (HPerf\ f\ a)$$

5.3 Elaboration of Terms

Term elaboration is straightforward. Rule TmVar handles term variables. The instantiation of the type scheme $\forall \overline{a}.\overline{C} \Rightarrow \tau$ to $[\overline{b}/\overline{a}]\tau$ becomes explicit in the System F representation, by the application of x to type variables $\overline{b}$, as well as the fresh dictionary variables $\overline{d}$, corresponding one-to-one to the implicit constraints $\overline{C}$. Rule TmAbs elaborates λ-abstractions. Since in System F all bindings are explicitly typed, in the elaborated term we annotate the binding of x with its type a. Similarly, Rule TmLet elaborates let bindings, again explicitly annotating x with its type υ_1 in the elaborated term. Rule TmApp is straightforward.

5.4 Dictionary Construction

The entailment algorithm of Figure 6 constructs explicit witness proofs (in the form of dictionary substitutions) while entailing a constraint.

Simplification The evidence substitution η in the simplification relation shows how to construct a witness for the wanted constraint C from the simpler constraints $\mathcal{A}'$ and program theory $\mathcal{P}$.

The goal of Rule ($\Rightarrow$R) is to build an evidence substitution η', which constructs a proof for $(d_0 : C_1 \Rightarrow C_2)$ from the proofs $\overline{d'}$ for the simpler constraints $\overline{C_1 \Rightarrow C}$. It is instructive to consider the generated evidence substitution in parts, also taking the types into account:

1. η illustrates how to generate a proof for $(d_2 : C_2)$, from the local assumption $(d_1 : C_1)$ and local residual constraints $(\overline{d : C})$.
2. $[\overline{d'\ d_1/d}]$ generates proofs for the (local) residual constraints $(\overline{d : C})$, by applying the residual constraints $(\overline{d' : C_1 \Rightarrow C})$ to the local assumption $(d_1 : C_1)$.
3. $([\overline{d'\ d_1/d}] \cdot \eta)(d_2)$ is a proof for C_2, under assumptions $(d_1 : C_1)$ and $(\overline{d' : C_1 \Rightarrow C})$.
4. Finally, we construct the proof for $(d_0 : C_1 \Rightarrow C_2)$ by explicitly abstracting over d_1: $\lambda(d_1 : \upsilon_1).[\overline{d'\ d_1/d}](\eta(d_2))$

Rule ($\forall$R) proceeds similarly. Finally, Rule (QR) generates the evidence substitution via constraint matching, which we discuss next.

Matching Similarly, the evidence substitution η in the matching relation shows how to construct a witness for the wanted constraint Q from the simpler constraints $\mathcal{A}$ and program theory $\mathcal{P}$.

Rule ($\Rightarrow$L) generates two fresh dictionary variables, d_1 for the residual constraint $\theta(C_1)$, and d_2 for the local assumption C_2. Finally, dictionary d_2 is replaced by the application of the dictionary transformer d to the residual dictionary d_1. Rule ($\forall$L) behaves similarly. The instantiation of the axiom d becomes explicit, by applying it to the chosen type $\theta(b)$. Finally, Rule (QL) is straightforward: since the wanted and the given constraints are identical (given that they unify), the wanted dictionary d is replaced by the given d'.

5.5 Declaration Elaboration

Figure 7 presents the elaboration of both class and instance declarations into System F.

Elaboration of Class Declarations A declaration for a class TC is encoded in System F as a dictionary type T_{TC}, with a single data constructor K_{TC} and $n + 1$ arguments: n arguments for the superclass dictionaries (of type $\overline{v}^n$) and one more for the method implementation (of type v). For example, the *Trans* declaration of Section 2.1 gives rise to the following dictionary type:

$$\begin{aligned} \textbf{data}\ T_{Trans}\ t = K_{Trans}\ & (\forall m.T_{Monad}\ m \rightarrow T_{Monad}\ (t\ m)) \\ & (\forall m\ a.T_{Monad}\ m \rightarrow m\ a \rightarrow (t\ m)\ a) \end{aligned}$$

Accordingly, we generate $n + 1$ projection functions that extract each of the arguments (d_i extracts the i-th superclass dictionary and f the method implementation). We use $proj^i_{TC}(d)$ to denote pattern matching against d and extracting the i-th argument:

$$proj^i_{TC}(d) \equiv \textbf{case}\ d\ \textbf{of}\ K_{TC}\ \overline{x}^k \rightarrow x_i \qquad ,\overline{x}^k\ \text{fresh}$$

where k denotes the arity of data constructor K_{TC}. E.g., the superclass projection function for class *Trans* takes the form:

$$\begin{aligned} d_{sc} &: \forall t.T_{Trans}\ t \rightarrow (\forall m.T_{Monad}\ m \rightarrow T_{Monad}\ (t\ m)) \\ d_{sc} &= \Lambda t.\lambda(d : T_{Trans}\ t).\ \textbf{case}\ d\ \textbf{of}\ \{\ K_{Trans}\ d'\ \_ \rightarrow d'\ \} \end{aligned}$$

Elaboration of Class Instances A class instance is elaborated into a System F dictionary transformer d_I:

$$\textbf{let}\ d_I : (\forall \overline{b}.\overline{v} \rightarrow T_{TC}\ \tau) = \Lambda\overline{b}.\overline{\lambda(d : v)}.K_{TC}\ \tau\ \overline{\eta(d')}\ t$$

Given dictionaries $\overline{d}$ – corresponding to the given context constraints – we need to provide all arguments of the data constructor K_{TC}: (a) the instantiation of the class type parameter, (b) the superclass dictionaries, and (c) the method implementation. The first argument is trivial. We obtain the superclass dictionaries by applying the evidence substitution η on the dictionary variables $\overline{d}'$ that abstract over the required superclass constraints. The method implementation t is elaborated via premise

$$\overline{b}; \mathcal{P}_I; \Gamma_I \vdash_{\mathtt{tm}} e : [\tau/a]\sigma \rightsquigarrow t$$

which elaborates type subsumption in a similar manner.

6 Termination of Resolution

Termination of resolution is the cornerstone of the overall termination of type inference. This section discusses how to enforce termination by means of syntactic conditions on the axioms. These conditions are adapted from those of Cochis [32] and generalize the earlier conditions for Haskell by Sulzmann et al. [35].

Overall Strategy We show termination by characterising the resolution process as a (resolution) tree with goals in the nodes and axioms on the (multi-)edges. The initial goal sits at the root of the tree. A multi-edge from a parent node to its children presents an axiom that matches the parent node's goal and its children are the residual goals. Resolution terminates iff the tree is finite. Hence, if it does not terminate, there is an infinite path from the root in the tree, that denotes an infinite sequence of axiom applications.

To show that there cannot be such an infinite path, we use a norm $\|\cdot\|$ that maps the head [6] of every goal C to a natural number, its size. The size of a class constraint $TC\ \tau$ is the size of its type parameter τ, which is given by the following equations:

$$\begin{aligned} \|a\| &= 1 \\ \|\tau_1 \rightarrow \tau_2\| &= 1 + \|\tau_1\| + \|\tau_2\| \end{aligned}$$

If we can show that this size strictly decreases from any parent goal to its children, then we know that, because the order on the natural numbers is well-founded, on any path from the root there is evantually a goal that has no children.

Termination Condition It is trivial to show that the size strictly decreases, if we require that every axiom makes it so. This requirement is formalised as the termination condition of axioms $term(C)$:

$$\frac{}{term(Q)}\ (Q\text{T}) \qquad \frac{term(C)}{term(\forall a.C)}\ (\forall\text{T})$$

$$\frac{\begin{array}{c} term(C_1) \qquad term(C_2) \\ Q_1 = head(C_1) \qquad Q_2 = head(C_2) \qquad \|Q_1\| < \|Q_2\| \\ \forall a \in fv(C_1) \cup fv(C_2):\quad occ_a(Q_1) \leq occ_a(Q_2) \end{array}}{term(C_1 \Rightarrow C_2)}\ (\Rightarrow\text{T})$$

Rule ($\Rightarrow$T) for $C_1 \Rightarrow C_2$ enforces the main condition, that the size of the residual constraint's head Q_1 is strictly smaller than the head Q_2 of C_2. In addition, the rule ensures that this property is stable under type substitution. Consider for instance the axiom $\forall a.C\ (a \rightarrow a) \Rightarrow C\ (a \rightarrow Int \rightarrow Int)$. The head's size 5 is strictly greater than the context constraint's size 3. Yet, if we instantiate a to $(Int \rightarrow Int \rightarrow Int)$, then the head's size becomes 10 while the context constraint's size becomes 11. Declaratively, we can formulate stability as:

$$\forall\theta.dom(\theta) \subseteq fv(C_1) \cup fv(C_2) \Rightarrow \|\theta(Q_1)\| < \|\theta(Q_2)\|$$

The rule uses instead an equivalent algorithmic formulation which states that the number of occurrences of any free type variable a may not be larger in Q_1 than in Q_2. Here the number of occurrences of a type variable a in a class constraint $TC\ \tau$ (denoted as $occ_a(TC\ \tau)$) is the same as the number of free occurrences of a in the parameter τ, where function $occ_a(\tau)$ is defined as:

$$\begin{aligned} occ_a(b) &= \begin{cases} 1 & \text{, if } a = b \\ 0 & \text{, if } a \neq b \end{cases} \\ occ_a(\tau_1 \rightarrow \tau_2) &= occ_a(\tau_1) + occ_a(\tau_2) \end{aligned}$$

Finally, as the constraints have a recursive structure whereby their components are themselves used as axioms, the rules also enforce the termination condition recursively on the components.

[6] The head of a constraint is defined as: $head(Q) = Q$; $head(\forall a.C) = head(C)$; and $head(C_1 \Rightarrow C_2) = head(C_2)$.

Superclass Condition If we could impose the termination condition above on all axioms in the theory P, we would be set. Unfortunately, this condition is too strong for the superclass axioms. Consider the superclass axiom $\forall a. Ord\ a \Rightarrow Eq\ a$ of the standard Haskell'98 *Ord* type class. Here both $Ord\ a$ and $Eq\ a$ have size 1; in other words, the size does not strictly decrease and so the axiom does not satisfy the termination condition.

To accommodate this and other examples, we impose an alternative condition for superclass axioms. This superclass condition relaxes the strict size decrease to a non-strict size decrease and makes up for it by requiring that the superclass relation forms a *directed acyclic graph* (DAG). The superclass relation is defined as follows on type classes.

Definition 6.1 (Superclass Relation)**.** Given a class declaration

$$\mathbf{class}\ (C_1, \ldots, C_n) \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \}$$

each type class TC_i is a superclass of TC, where $head(C_i) = TC_i\ \tau_i$.

Observe that the DAG induces a well-founded partial order on type classes. Hence, on any path in the resolution tree, any uninterrupted sequence of superclass axiom applications has to be finite. For the length of such a sequence, the size of the goal does not increase (but might not decrease either). Yet, after a finite number of steps the sequence has to come to an end. If the path still goes on at that point, it must be due to the application of an instance or local axiom, which strictly decreases the goal size. Hence, overall we have preserved the variant that the goal size decreases after a bounded number[7] of steps.

7 Related Work

This section discusses related work, focusing mostly on comparing our approach with existing encodings/workarounds in Haskell. The history of quantified class constraints and their demand in previous research was already discussed in Section 1.

The Coq Proof Assistant Coq provides very flexible support for type classes [33] and allows for arbitrary formulas in class and instance contexts – actually the contexts are just parameters. For instance, we can model the *Trans* class as:

```
Class Trans (T : (Type -> Type) -> Type -> Type)
  `{forall M, `{Monad M} -> Monad (T M)} :=
  { lift : forall A M, `{Monad M} -> M A -> (T M) A }.
```

The downside of Coq's flexibility is that resolution can be ambiguous and non-terminating. The accepted workaround is for the programmer to perform resolution manually when necessary. This is acceptable in the context of Coq's interactive approach to proving, but would mean a great departure from Haskell's non-interactive type inference.

Trifonov's Workaround and Monatron Trifonov [36] gives an encoding of quantified class constraints in terms of regular class constraints. The encoding introduces a new type class that encapsulates the quantified constraint, e.g. $Monad\_t\ t$ for $\forall m. Monad\ m \Rightarrow Monad\ (t\ m)$, and that provides the implied methods under a new name. This expresses the *Trans* problem as follows:

$$\begin{aligned}
&\mathbf{class}\ Monad_t\ t\ \mathbf{where}\\
&\quad treturn :: Monad\ m \Rightarrow a \rightarrow t\ m\ a\\
&\quad tbind\ \ :: Monad\ m \Rightarrow t\ m\ a \rightarrow (a \rightarrow t\ m\ b) \rightarrow t\ m\ b\\
&\mathbf{class}\ Monad_t\ t \Rightarrow Trans\ t\ \mathbf{where}\\
&\quad lift :: Monad\ m \Rightarrow m\ a \rightarrow t\ m\ a
\end{aligned}$$

While this approach captures the intention of the quantified constraint, it does not enable the type checker to see that $Monad\ (t\ m)$ holds for any transformer t and monad m. While the monad methods are available for $t\ m$, they do not have the usual name.

For this reason, Trifonov presents a further (non-Haskell'98) refinement of the encoding, which was adopted by the Monatron [13] library[8] among others. A non-essential difference is that Monatron merges the above *Monad_t* and *Trans* into a single class:

$$\begin{aligned}
&\mathbf{class}\ MonadT\ t\ \mathbf{where}\\
&\quad lift\ \ \ \ \ :: Monad\ m \Rightarrow m\ a \rightarrow t\ m\ a\\
&\quad treturn :: Monad\ m \Rightarrow a \rightarrow t\ m\ a\\
&\quad tbind\ \ \ :: Monad\ m \Rightarrow t\ m\ a \rightarrow (a \rightarrow t\ m\ b) \rightarrow t\ m\ b
\end{aligned}$$

The key novelty is that it also makes the methods *treturn* and *tbind* available under their usual name with a single *Monad* instance for all monad transformers.

$$\begin{aligned}
&\mathbf{instance}\ (Monad\ m, MonadT\ t) \Rightarrow Monad\ (t\ m)\ \mathbf{where}\\
&\quad return = treturn\\
&\quad (>\!\!>\!\!=)\ = tbind
\end{aligned}$$

With these definitions the monad transformer composition does type check. Unfortunately, the head of the $Monad\ (t\ m)$ instance is highly generic and easily overlaps with other instances.

The MonadZipper Because they found Monatron's overlapping instances untenable, Schrijvers and Oliveira [31] presented a different workaround for this problem in the context of their monad zipper datatype, which is an extended form of transformer composition. Their solution adds a method *mw* to the *Trans* type class:

$$\begin{aligned}
&\mathbf{class}\ Trans\ t\ \mathbf{where}\\
&\quad lift\ :: Monad\ m \Rightarrow m\ a \rightarrow t\ m\ a\\
&\quad mw :: Monad\ m \Rightarrow MonadWitness\ t\ m
\end{aligned}$$

For any monad m this method returns a GADT [29] witness for the fact that $t\ m$ is a monad. This is possible because with GADTs, type class instances can be stored in the data constructors.

$$\begin{aligned}
&\mathbf{data}\ MonadWitness\ (t :: (* \rightarrow *) \rightarrow (* \rightarrow *))\ m\ \mathbf{where}\\
&\quad MW :: Monad\ (t\ m) \Rightarrow MonadWitness\ t\ m
\end{aligned}$$

By pattern matching on the witness of the appropriate type the programmer can bring the required $Monad\ (t_2\ m)$ constraint into scope to satisfy the type checker.

$$\begin{aligned}
&\mathbf{instance}\ (Trans\ t_1, Trans\ t_2) \Rightarrow Trans\ (t_1 * t_2)\ \mathbf{where}\\
&\quad lift\ :: \forall m\ a. Monad\ m \Rightarrow m\ a \rightarrow (t_1 * t_2)\ m\ a\\
&\quad lift\ = \mathbf{case}\ (mw :: MonadWitness\ t_2\ m)\ \mathbf{of}\\
&\qquad MW \rightarrow C \cdot lift \cdot lift\\
&\quad mw = \ldots
\end{aligned}$$

The downside of this approach is that it offloads part of the type checker's work on the programmer. As a consequence the code becomes cluttered with witness manipulation.

[7] bounded by the height of the superclass DAG

[8] For the implementation see https://hackage.haskell.org/package/Monatron

The constraint Library Kmett's constraint library [20] provides generic infrastructure for reifying quantified constraints in terms of GADTs, not unlike in the MonadZipper solution above. While not impossible, encoding the *Trans* problem with this library is a daunting task indeed.

Corecursive Resolution Fu et al. [7] address the divergence problem that arises for generic nested datatypes. They turn the diverging resolution with user-supplied instances into a terminating resolution in terms of automatically derived instances. These auxiliary instances are derived specifically to deal with the query at hand; they shift the pattern of divergence to the term-level in the form of co-recursively defined dictionaries. The authors do point out that the class of divergent cases they support is limited and that deriving quantified instances would be beneficial.

Cochis The calculus of coherent implicits, Cochis [32], and its focusing-based resolution in particular, have been a major inspiration of this work. Just like this work, Cochis supports recursive resolution of quantified constraints. Yet, there are a number of significant differences. Firstly, Cochis does not feature a separate syntactic sort for type classes, but implicitly resolves regular terms in the Scala tradition. As a consequence, it does not distinguish between instance and superclass axioms, e.g., for the sake of enforcing termination and coherence. Perhaps more significantly, Cochis features local "instances" as opposed to our globally scoped instances. Local instances may overlap with one another and coherence is obtained by prioritizing those instances that are introduced in the innermost scope. This way Cochis's resolution is entirely deterministic, while ours is non- deterministic (yet coherent) due to overlapping local and superclass axioms.

8 Conclusion

This paper has presented a fully fledged design of quantified class constraints. We have shown that this feature significantly increases the modelling power of type classes, while at the same enables a terminating type class resolution for a larger class of applications. Interesting future work we aim to pursue includes (a) establishing the metatheory, (b) extending the system with quantification over predicates[9], raising the power of type classes to (a fragment of) second-order logic, and (c) studying the interaction of quantified class constraints with commonly used type-level features like *functional dependencies* [18] or *associated type families* [4], allowing us to integrate the new feature in Haskell's ecosystem.

[9] See GHC feature request #5927.

References

[1] Jean-marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2 (1992), 297–347.

[2] H. Barendregt. 1981. *The Lambda Calculus: its Syntax and Semantics, volume 103 of Studies in Logic and the Foundations of Mathematics.* North-Holland.

[3] Richard S. Bird and Lambert G. L. T. Meertens. 1998. Nested Datatypes. In *Proceedings of the Mathematics of Program Construction (MPC '98).* Springer-Verlag, London, UK, 52–67.

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

[5] Satvik Chauhan, Piyush P. Kurur, and Brent A. Yorgey. 2016. How to Twist Pointers Without Breaking Them. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016).* ACM, New York, NY, USA, 51–61.

[6] Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82).* ACM, New York, NY, USA, 207–212.

[7] Peng Fu, Ekaterina Komendantskaya, Tom Schrijvers, and Andrew Pond. 2016. Proof Relevant Corecursive Resolution. In *Functional and Logic Programming: 13th International Symposium, Proceedings (FLOPS 2016)*, Oleg Kiselyov and Andy King (Eds.). Springer International Publishing, 126–143.

[8] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types.* Cambridge University Press, New York, NY, USA.

[9] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (March 1996), 109–138.

[10] Ralf Hinze. 2000. Perfect trees and bit-reversal permutations. *J. Funct. Program.* 10, 3 (2000), 305–317.

[11] Ralf Hinze. 2010. Adjoint Folds and Unfolds: Or: Scything Through the Thicket of Morphisms. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC'10).* Springer-Verlag, Berlin, Heidelberg, 195–228.

[12] Ralf Hinze and Simon Peyton Jones. 2000. Derivable Type Classes. In *Proceedings of the Fourth Haskell Workshop.* Elsevier Science, 227–236.

[13] Mauro Jaskelioff. 2011. Monatron: an extensible monad transformer library. In *Proceedings of the 20th international conference on Implementation and application of functional languages (IFL'08).* Springer-Verlag, Berlin, Heidelberg, 233–248.

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

[15] Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text.* Springer-Verlag, London, UK, UK, 97–136.

[16] Mark P. Jones. 1995. *Qualified Types: Theory and Practice.* Cambridge University Press, New York, NY, USA.

[17] Mark P. Jones. 1995. Simplifying and Improving Qualified Types. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95).* ACM, New York, NY, USA, 160–169.

[18] Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Programming Languages and Systems*, Gert Smolka (Ed.). LNCS, Vol. 1782. Springer Berlin Heidelberg, 230–244.

[19] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Proceedings of the 1997 Haskell Workshop.* ACM.

[20] Edward A. Kmett. 2017. The constraint package. (2017). https://hackage.haskell.org/package/constraints-0.9.1.

[21] Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. *SIGPLAN Not.* 38, 3 (Jan. 2003), 26–37.

[22] Ralf Lämmel and Simon Peyton Jones. 2005. Scrap Your Boilerplate with Class: Extensible Generic Functions. *SIGPLAN Not.* 40, 9 (Sept. 2005), 204–215.

[23] Chuck Liang and Dale Miller. 2009. Focusing and Polarization in Linear, Intuitionistic, and Classical Logics. *Theor. Comput. Sci.* 410, 46 (Nov. 2009), 4747–4768.

[24] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. 1989. *Uniform Proofs As a Foundation for Logic Programming.* Technical Report. Durham, NC, USA.

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

[26] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes As Objects and Implicits. *SIGPLAN Not.* 45, 10 (Oct. 2010), 341–360.

[27] Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. 2012. The Implicit Calculus: A New Foundation for Generic Programming. *SIGPLAN Not.* 47, 6 (June 2012), 35–44.

[28] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-rank Types. *J. Funct. Program.* 17, 1 (Jan. 2007).

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

[30] Frank Pfenning. 2010. Lecture Notes on Focusing. (2010). https://www.cs.cmu.edu/~fp/courses/oregon-m10/04-focusing.pdf.

[31] Tom Schrijvers and Bruno C.d.S. Oliveira. 2011. Monads, Zippers and Views: Virtualizing the Monad Stack. *SIGPLAN Not.* 46, 9 (Sept. 2011), 32–44.

[32] Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. *Cochis: Deterministic and Coherent Implicits.* Report CW 705. KU Leuven, Department of Computer Science.

[33] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008. Proceedings (LNCS)*, Vol. 5170. Springer, 278–293.

[34] Mike Spivey. 2017. Faster Coroutine Pipelines. In *International Conference on Functional Programming (ICFP).* accepted.

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

[36] Valery Trifonov. 2003. Simulating Quantified Class Constraints. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03).* ACM, New York, NY, USA, 98–102.

[37] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let Should Not Be Generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '10).* ACM, NY, USA, 39–50.

[38] P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89).* ACM, New York, NY, USA, 60–76.

A Additional Judgments

A.1 Well-formedness of Types & Constraints

Well-formedness of types takes the form $\Gamma \vdash_{\mathsf{ty}} \sigma$ and is given by the following rules:

$$\frac{a \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} a}\ \textsc{TyVar} \qquad \frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \to \tau_2}\ \textsc{TyArr}$$

$$\frac{\Gamma \vdash_{\mathsf{ct}} C \qquad \Gamma \vdash_{\mathsf{ty}} \rho}{\Gamma \vdash_{\mathsf{ty}} C \Rightarrow \rho}\ \textsc{TyQual} \qquad \frac{\Gamma, a \vdash_{\mathsf{ty}} \sigma}{\Gamma \vdash_{\mathsf{ty}} \forall a.\sigma}\ \textsc{TyAll}$$

It is entirely straightforward and ensures that type terms are well-scoped. Rule TyQual requires checking the well-formedness of our new form of constraints C, via relation $\Gamma \vdash_{\mathsf{ct}} C$, given by the following rules:

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau}{\Gamma \vdash_{\mathsf{ct}} TC\ \tau}\ (\text{CQ}) \qquad \frac{\Gamma \vdash_{\mathsf{ct}} C_1 \qquad \Gamma \vdash_{\mathsf{ct}} C_2}{\Gamma \vdash_{\mathsf{ct}} C_1 \Rightarrow C_2}\ (\text{C}\Rightarrow) \qquad \frac{\Gamma, a \vdash_{\mathsf{ct}} C}{\Gamma \vdash_{\mathsf{ct}} \forall a.C}\ (\text{C}\forall)$$

Finally, an axiom set A is well-formed if all constraints it contains are well-formed:

$$\frac{}{\Gamma \vdash_{\mathsf{ax}} \bullet}\ \textsc{AxNil} \qquad \frac{\Gamma \vdash_{\mathsf{ax}} A \qquad \Gamma \vdash_{\mathsf{ct}} C}{\Gamma \vdash_{\mathsf{ax}} A, C}\ \textsc{AxCons}$$

A.2 Program Typing

The judgment for program typing takes the form $P; \Gamma \vdash_{\mathsf{pgm}} pgm : \sigma$ and is given by the following rules:

$$\frac{\Gamma \vdash_{\mathsf{cls}} cls : A_S; \Gamma_c \qquad P,_{\mathsf{S}} A_S; \Gamma, \Gamma_c \vdash_{\mathsf{pgm}} pgm : \sigma}{P; \Gamma \vdash_{\mathsf{pgm}} (cls; pgm) : \sigma}\ \textsc{PgmCls}$$

$$\frac{P; \Gamma \vdash_{\mathsf{inst}} inst : A_I \qquad P,_{\mathsf{I}} A_I; \Gamma \vdash_{\mathsf{pgm}} pgm : \sigma}{P; \Gamma \vdash_{\mathsf{pgm}} (inst; pgm) : \sigma}\ \textsc{PgmInst}$$

$$\frac{P; \Gamma \vdash_{\mathsf{tm}} e : \sigma}{P; \Gamma \vdash_{\mathsf{pgm}} e : \sigma}\ \textsc{PgmExpr}$$

For brevity, if $P = \bullet$ and $\Gamma = \bullet$ we denote program typing as $\vdash_{\mathsf{pgm}} pgm : \sigma$.

A.3 Unification Algorithm

The unification algorithm takes the form $unify(\bar{a}; E) = \theta_\perp$ and is given by the following equations:

$$\begin{array}{lll}
unify(\bar{a}; \bullet) & = & \bullet \\
unify(\bar{a}; E, b \sim b) & = & unify(\bar{a}; E) \\
unify(\bar{a}; E, b \sim \tau) & = & unify(\bar{a}; \theta(E)) \cdot \theta \\
\quad \text{where } b \notin \bar{a} \wedge b \notin fv(\tau) \wedge \theta = [\tau/b] & & \\
unify(\bar{a}; E, \tau \sim b) & = & unify(\bar{a}; \theta(E)) \cdot \theta \\
\quad \text{where } b \notin \bar{a} \wedge b \notin fv(\tau) \wedge \theta = [\tau/b] & & \\
unify(\bar{a}; E, (\tau_1 \to \tau_2) \sim (\tau_3 \to \tau_4)) & = & unify(\bar{a}; E, \tau_1 \sim \tau_3, \tau_2 \sim \tau_4)
\end{array}$$

Function *unify* is a straightforward extension of the standard first-order unification algorithm [6]. The only difference between the two lies in the additional argument: the *untouchable* variables $\bar{a}$. These variables are treated by the algorithm as skolem constants and therefore can not be substituted (they can be unified with themselves though).

A.4 Elaboration of Programs

Elaboration of programs is given by judgment $\mathcal{P}; \Gamma \vdash_{\mathsf{pgm}} pgm : \sigma \rightsquigarrow fpgm$:

$\boxed{\mathcal{P}; \Gamma \vdash_{\mathsf{pgm}} pgm : \sigma \rightsquigarrow fpgm}$ Program Elaboration

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{cls}} cls : \mathcal{A}_S; \Gamma_c \rightsquigarrow fdata; \overline{fval} \\ \mathcal{P},_{\mathsf{S}} \mathcal{A}_S; \Gamma, \Gamma_c \vdash_{\mathsf{pgm}} pgm : \sigma \rightsquigarrow fpgm\end{array}}{\mathcal{P}; \Gamma \vdash_{\mathsf{pgm}} (cls; pgm) : \sigma \rightsquigarrow fdata; \overline{fval}; fpgm}\ \textsc{PCls}$$

$$\frac{\begin{array}{c}\mathcal{P}; \Gamma \vdash_{\mathsf{inst}} inst : \mathcal{A}_I \rightsquigarrow fval \\ \mathcal{P},_{\mathsf{I}} \mathcal{A}_I; \Gamma \vdash_{\mathsf{pgm}} pgm : \sigma \rightsquigarrow fpgm\end{array}}{\mathcal{P}; \Gamma \vdash_{\mathsf{pgm}} (inst; pgm) : \sigma \rightsquigarrow fval; fpgm}\ \textsc{PIns}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{tm}} e : \tau \rightsquigarrow t \mid \mathcal{A}; E \\ \theta = unify(\bullet; E) \qquad \bar{a} = fv(\theta(\mathcal{A})) \cup fv(\theta(\tau)) \\ \bar{a}; \langle \bullet, \mathcal{A}_I, \mathcal{A}_L \rangle \models \theta(\mathcal{A}) \rightsquigarrow \overline{d : C}; \eta \qquad \overline{\vdash_{\mathsf{ct}} C_i \rightsquigarrow v_i}\end{array}}{\langle \mathcal{A}_S, \mathcal{A}_I, \mathcal{A}_L \rangle; \Gamma \vdash_{\mathsf{pgm}} e : \forall \bar{a}.\overline{C} \Rightarrow \theta(\tau) \rightsquigarrow \Lambda \bar{a}.\lambda(\overline{d : v}).\eta(\theta(t))}\ \textsc{PExp}$$

Rules PCls and PIns handle class and instance declarations, respectively, and they are entirely standard. Rule PExp performs standard type-inference, simplification [17] and generalization for a top-level expression e. For simplicity, we do not utilize *interaction rules* (e.g. we do not simplify the constraints $\{Eq\ a, Ord\ a\}$ to $\{Ord\ a\}$), but it is straightforward to do so. Finally, observe that superclass axioms $\mathcal{A}_S$ are not used for the simplification of wanted constraints. This is standard practice for Haskell but our distinction between the axioms within the program theory allows us to express this explicitly.

B System F Semantics

Both the typing rules and call-by-name operational semantics for System F are entirely standard and can be found elsewhere, we include them here to keep the presentation self-contained. In the following, we denote System F typing environments by Δ:

$$\Delta \ ::= \ \bullet \mid \Delta, T \mid \Delta, K : v \mid \Delta, a \mid \Delta, x : v \qquad \textit{typing environment}$$

B.1 Term Typing

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} t : v}$ Term Typing

$$\frac{(x : v) \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} x : v}\ \textsc{TmVar} \qquad \frac{\Delta, x : v_1 \vdash^{\mathsf{F}}_{\mathsf{tm}} t : v_2 \qquad \Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} v_1}{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} \lambda(x : v_1).t : v_1 \to v_2}\ (\to\text{I})$$

$$\frac{(K : v) \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} K : v}\ \textsc{TmCon} \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} t_1 : v_1 \to v_2 \qquad \Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} t_2 : v_1}{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} t_1\ t_2 : v_2}\ (\to\text{E})$$

$$\frac{\Delta, a \vdash^{\mathsf{F}}_{\mathsf{tm}} t : v}{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} \Lambda a.t : \forall a.v}\ (\forall\text{I}) \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} t : \forall a.v \qquad \Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} v_1}{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} t\ v_1 : [v_1/a]v}\ (\forall\text{E})$$

$$\frac{\Delta, x : v_1 \vdash^{\mathsf{F}}_{\mathsf{tm}} t_1 : v_1 \qquad \Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} v_1 \qquad \Delta, x : v_1 \vdash^{\mathsf{F}}_{\mathsf{tm}} t_2 : v_2}{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} (\mathbf{let}\ x : v_1 = t_1\ \mathbf{in}\ t_2) : v_2}\ \textsc{TmLet}$$

$$\frac{\begin{array}{c}\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} t_1 : T\ v \\ (K : \forall a.\overline{v} \to T\ a) \in \Delta \qquad \Delta, \overline{x : [v/a]v} \vdash^{\mathsf{F}}_{\mathsf{tm}} t_2 : v_2\end{array}}{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} (\mathbf{case}\ t_1\ \mathbf{of}\ K\ \overline{x} \to t_2) : v_2}\ \textsc{TmCase}$$

B.2 Well-formedness of Types

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} v}$ Type Well-formedness

$$\frac{a \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} a}\ \textsc{TyVar} \qquad \frac{T \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} T}\ \textsc{TyCon} \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} \upsilon_1 \qquad \Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} \upsilon_2}{\Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} \upsilon_1 \to \upsilon_2}\ \textsc{TyArr}$$

$$\frac{\Delta, a \vdash^{\mathsf{F}}_{\mathsf{ty}} \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} \forall a.\upsilon}\ \textsc{TyAll} \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} \upsilon_1 \qquad \Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} \upsilon_2}{\Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} \upsilon_1\ \upsilon_2}\ \textsc{TyApp}$$

B.3 Value Binding Typing

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathsf{val}} \mathit{fval} : \Delta_{\mathit{fval}}}$ Value Binding Typing

$$\frac{\Delta, x : \upsilon \vdash^{\mathsf{F}}_{\mathsf{tm}} t : \upsilon \qquad \Delta \vdash^{\mathsf{F}}_{\mathsf{ty}} \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathsf{val}} (\mathbf{let}\ x : \upsilon = t) : [x : \upsilon]}\ \textsc{Val}$$

B.4 Datatype Declaration Typing

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathsf{data}} \mathit{fdata} : \Delta_{\mathit{fdata}}}$ Datatype Declaration Typing

$$\frac{\overline{\Delta, a \vdash^{\mathsf{F}}_{\mathsf{ty}} \upsilon}}{\Delta \vdash^{\mathsf{F}}_{\mathsf{val}} (\mathbf{data}\ T\ a = K\ \overline{\upsilon}) : [T, K : \forall a.\overline{\upsilon} \to T\ a]}\ \textsc{Data}$$

B.5 Program Typing

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathsf{pgm}} \mathit{fpgm} : \upsilon}$ Program Typing

$$\frac{\Delta \vdash^{\mathsf{F}}_{\mathsf{tm}} t : \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathsf{pgm}} t : \upsilon}\ \textsc{PgmExpr}$$

$$\frac{\Delta \vdash^{\mathsf{F}}_{\mathsf{val}} \mathit{fval} : \Delta_{\upsilon} \qquad \Delta, \Delta_{\upsilon} \vdash^{\mathsf{F}}_{\mathsf{pgm}} \mathit{fpgm} : \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathsf{pgm}} (\mathit{fval}; \mathit{fpgm}) : \upsilon}\ \textsc{PgmVal}$$

$$\frac{\Delta \vdash^{\mathsf{F}}_{\mathsf{data}} \mathit{fdata} : \Delta_{d} \qquad \Delta, \Delta_{d} \vdash^{\mathsf{F}}_{\mathsf{pgm}} \mathit{fpgm} : \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathsf{pgm}} (\mathit{fdata}; \mathit{fpgm}) : \upsilon}\ \textsc{PgmData}$$

For brevity, if $\Delta = \bullet$ we denote System F program typing as $\vdash^{\mathsf{F}}_{\mathsf{pgm}} \mathit{fpgm} : \upsilon$.

B.6 Call-by-name Operational Semantics

The small-step, call-by-name operational semantics of System F are presented below:

$\boxed{t \longrightarrow t'}$ Operational Semantics (Small-step)

$$\frac{}{(\Lambda a.t)\ \upsilon \longrightarrow [\upsilon/a]t}\ \textsc{TyBeta} \qquad \frac{}{(\lambda(x : \upsilon).t)\ t' \longrightarrow [t'/x]t}\ \textsc{TmBeta}$$

$$\frac{t_1 \longrightarrow t_1'}{(\mathbf{case}\ t_1\ \mathbf{of}\ K\ \overline{x} \to t_2) \longrightarrow (\mathbf{case}\ t_1'\ \mathbf{of}\ K\ \overline{x} \to t_2)}\ \textsc{CaseStep}$$

$$\frac{}{(\mathbf{case}\ K\ \overline{t}\ \mathbf{of}\ K\ \overline{x} \to t) \longrightarrow [\overline{t}/\overline{x}]t}\ \textsc{CaseBeta}$$

$$\frac{}{(\mathbf{let}\ x : \upsilon = t_1\ \mathbf{in}\ t_2) \longrightarrow [\mathbf{let}\ x : \upsilon = t_1\ \mathbf{in}\ t_1/x]t_2}\ \textsc{LetBeta}$$