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:
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-kindred 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, Lammel 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, requesting for quantified class 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] rules 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 CoqHs [32], a calculus for Scala-style implicit [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-kindred datatypes and accepts all examples in this paper, at https://github.com/gkaracha/quantcs-impl.

1https://ghc.haskell.org/trac/ghc/ticket/2893
2except 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 in-
stances 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]:

\[
\text{class } Trans \ t \ where
\]
\[
lift :: \text{Monad } m \Rightarrow m a \rightarrow (t m) a
\]
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 \text{Monad} instance of the form:

\[
\text{instance } \text{Monad } m \Rightarrow \text{Monad } (T m) \ where \ 
\]
Because the type checker is not told about this requirement, it will
not accept the following definition of monad transformer composi-
tion:

\[
\text{newtype } (t_1 \ast t_2) m a = C \{ \text{runC} :: t_1 (t_2 m) a \} \]
\[
\text{instance } (Trans \ t_1, Trans \ t_2) \Rightarrow Trans \ (t_1 \ast t_2) \ where
\]
\[
lift = C \cdot \text{lift} \cdot \text{lift}
\]
The idea of this code is to \( \text{lift} \) from monad \( m \) to \( (t_2 m) \) and then to
\( \text{lift} \) from \( (t_2 m) \) to \( t_1 (t_2 m) \). However, the second \( \text{lift} \) 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:

\[
\text{class } (\forall m. \text{Monad } m \Rightarrow \text{Monad } (t m)) \Rightarrow Trans \ t \ where
\]
\[
lift :: \text{Monad } m \Rightarrow m a \rightarrow (t m) a
\]
The instance for transformer composition \( t_1 \ast t_2 \) 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 of \text{Mu} of a second-order
functor:

\[
\text{data } \text{Mu } h a = \text{In } \{ \text{out} :: h (\text{Mu } h) a \}
\]
\[
\text{data } \text{List}_2 f a = \text{Nil } | \text{Cons } a (f a)
\]
\[
\text{type } \text{List} = \text{Mu } \text{List}_2
\]
A second-order functor \( h \) is a type constructor that sends functors
to functors. This can be concisely expressed with the quantified
class constraint \( \forall f. \text{Functor } f \Rightarrow \text{Functor } (h f) \), for example in the
\text{Functor} instance of \( \text{Mu} \):

\[
\text{instance } (\forall f. \text{Functor } f \Rightarrow \text{Functor } (h f)) \Rightarrow \text{Functor } (\text{Mu } h)
\]
\[
\text{where } \text{fmap } f \text{ (In } x) = \text{In } (\text{fmap } f x)
\]
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 represen-
tation, 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:

\[
\text{data } \text{Perfect } a = \text{Zero } a | \text{Succ } (\text{Perfect } (a, a))
\]
This can be expressed with the generic representation as \( \text{Mu } h \text{Perf} \),
the fixpoint of the second-order functor \( h \text{Perf} \), defined as

\[
\text{data } h \text{Perf } f a = h \text{Zero } a | h \text{Succ } (f (a, a))
\]
Johann and Ghani’s notion of second-order functor differs slightly
from Hinze’s.3 Ideally, their notion would be captured by the fol-
lowing class declaration:

\[
\text{class } (\forall f. \text{Functor } f \Rightarrow \text{Functor } (h f)) \Rightarrow h \text{Functor } h \text{ where}
\]
\[
\text{hfmap } :: (\text{Functor } f, \text{Functor } g)
\]
\[
\Rightarrow (\forall x. f x \rightarrow g x) \rightarrow (\forall x. h f x \rightarrow h g x)
\]
Like in Hinze’s case, the quantified class constraint expresses that
a second-order functor takes first-order functors to first-order func-
tors. Additionally, second-order functors provide a second-order
\( \text{fmap} \), called \( \text{hfmap} \), which 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 follow-
ing declaration instead:

\[
\text{class } h \text{Functor } h \text{ where}
\]
\[
\text{hfmap } :: (\text{Functor } f, \text{Functor } g)
\]
\[
\Rightarrow (\forall x. f x \rightarrow g x) \rightarrow (\forall x. h f x \rightarrow h g x)
\]
In essence, they inline the \( \text{fmap} \) method provided by the quantified
class constraint in the \( h \text{Functor} \) class. This is unfortunate because it
duplicates the \( \text{Functor} \) class’s functionality.

2.2 Terminating Corecursive Resolution
Quantified class constraints were first proposed by Hinze and Pey-
ton Jones [12] as a solution to a problem of diverging type class
resolution. Consider their generalized rose tree datatype:

\[
\text{data } \text{GRoot } f a = \text{GBranch } a (f (\text{GRoot } f a))
\]
and its \text{Show} instance:

\[
\text{instance } \text{Show } a, \text{Show } (f (\text{GRoot } f a)) \Rightarrow \text{Show } (\text{GRoot } f a)
\]
\[
\text{where } \text{show } (\text{GBranch } x s) = \text{unwords } (\text{show } x, \text{”-“}, \text{show } s)
\]
Notice the two constraints in the instance context which are due to
the two \text{show} invocations in the method definition. Standard
recursive type class resolution would diverge when faced with the
constraint \( (\text{Show } (\text{GRoot } [] \text{ Bool}) \). Indeed, it would recursively solve
the instance context: \text{Show} \text{ Bool} is easily dismissed, but \text{Show} \text{ [GRoot } [] \text{ Bool} \] requires solving \text{Show} \text{ [GRoot } [] \text{ Bool} \] again. Clearly this process loops.

To solve this problem, Hinze and Peyton Jones proposed to write the
\text{GRoot} instance with a quantified type class constraint as:

\[
\text{instance } (\forall a, \text{Show } x \Rightarrow \text{Show } (f x)) \Rightarrow \text{Show } (\text{GRoot } f a)
\]
\[
\text{where } (\text{Show } (\text{GBranch } x s)) = \text{unwords } (\text{show } x, \text{”-“}, \text{show } s)
\]
This would avoid the diverging loop in the type system extension
they sketch, because the two recursive resolvents, \text{Show} \text{ Bool} and \text{Show} \text{ x} are readily discharged with the available
\text{Bool} and \( a \) instances.

When faced with the same loops in their \text{Scrap Your Boilerplate} work, Lambl 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 (HSuc x) = "(S " ++ show x ++ ")"

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

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

instance (Show a, ∀ 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 (h f x)) ⇒ Show (HPerf f a)
where show (HZero a) = "(Z " ++ show a ++ ")"
  show (HSuc x) = "(S " ++ show x ++ ")"

2.3 Summary

In summary, quantified type class constraints enable 1) expressing
more of a type class's specification in a natural and succinct man-
er, 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 provided in Section 4 and Section 5 pro-
vides 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 lan-
guage. 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.

\[ x, y, z, f := \langle \text{term variable name} \rangle \]
\[ a, b, c := \langle \text{type variable name} \rangle \]
\[ TC := \langle \text{class name} \rangle \]
\[ pgm := e | cls; pgm | \text{inst; pgm} \]
\[ cls := \text{class } A \Rightarrow TC \text{ a where } \{ f :: \sigma \} \]
\[ \text{inst := instance } A \Rightarrow TC \text{ r where } \{ f = e \} \]
\[ e := x | \lambda x.e | e_1 \ e_2 | \text{let } x = e_1 \text{ in } e_2 \]
\[ \tau := a | \tau_1 \rightarrow \tau_2 \]
\[ \rho := \tau | C \Rightarrow \rho \]
\[ \sigma := \rho | \forall a.\sigma \]
\[ A := \bullet \mid A, C \]
\[ C := Q | C_1 \Rightarrow C_2 | \forall a. C \]
\[ Q := TC \tau \]
\[ \Gamma := \bullet \mid \Gamma, x : \sigma | \Gamma, a \]
\[ P := \langle A_S, A_L \rangle \]

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 be-
tween monotypes \( \tau \), qualified types \( \rho \) and type schemes \( \sigma \). 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 gen-
eralizes 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:
\[ \forall a. (Q_1 \land \ldots \land Q_n) \Rightarrow Q_0 \]

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/instantiation contexts. Consequently, the syntactic
sort \( C \) of constraints and axioms is one and the same.

Note that constraint schemes of the form \( \forall a. (Q_1 \land \ldots \land Q_n) \Rightarrow Q_0 \),
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:
\[ \forall a. (Q_1 \land \ldots \land Q_n) \Rightarrow Q_0 \equiv \forall a. (Q_1 \Rightarrow (\ldots (Q_n \Rightarrow Q_0) \ldots)) \]

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 \( \Gamma \), 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 \( P_0, C \) to denote that
we extend the local component of the triple, and similar notation
### Figure 2. Declarative Type System (Selected Rules)

Term Typing

\[
P; \Gamma, e : \sigma \\
\frac{\Gamma, e : \tau \in \Gamma}{P; \Gamma, x : \tau \Gamma, e : \sigma} \quad \text{Var}
\]

\[
P; \Gamma, x : \tau \Gamma, e : \sigma \\
\frac{\text{let } x = e_1 \text{ in } e_2 : \sigma}{P; \Gamma, e_2 : \sigma} \quad \text{Let}
\]

Class Declaration Typing

\[
\Gamma \vdash x : A ; \Gamma
\]

\[
\Gamma, a : \sigma \vdash C \quad \text{Class}
\]

Class Instance Typing

\[
P; \Gamma, \text{instance } A \Rightarrow \text{TC } \tau \quad \text{Instance}
\]

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-scopedness on types, constraints and axiom sets are denoted \( \Gamma \vdash_1 \sigma \), \( \Gamma \vdash_1 C \) and \( \Gamma \vdash_1 A \), respectively. Their definitions are straightforward and can be found in Appendix A.

#### Term Typing

Term typing takes the form \( P; \Gamma, e : \sigma \) and can be read as "under program theory \( P \) and typing environment \( \Gamma \), expression \( e \) has type \( \sigma \)." 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 \( P; \Gamma \vdash_1 C \), which is used in the constraint elimination Rule \((=)\). The definition of this auxiliary judgment is discussed in detail in Section 3.3.

#### Class Declaration Typing

Typing for class declarations takes the form \( \Gamma \vdash_1 \text{cls } : A ; \Gamma \) 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 \( f(a) \subseteq \{a\} \), 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:

\[
\forall t. \text{Trans } t \Rightarrow (\forall m. \text{Monad } m \Rightarrow \text{Monad } (t m))
\]

#### Class Instance Typing

Instance typing takes the form \( P; \Gamma \vdash_1 \text{inst } : A \) 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 \( P; \Gamma \vdash_1 \text{pgm } : \sigma \). 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 Coчис 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:

\[
\begin{align*}
\text{SPEC} & : \quad C \in P \quad \Rightarrow \quad \Gamma \vdash C \\
\text{VC} & : \quad P, \Gamma \vdash \forall a.C \\
\text{EC} & : \quad P, \Gamma \vdash [\tau/a]C \\
\text{IC} & : \quad P, \Gamma \vdash \forall C_1 \Rightarrow C_2 \\
\text{VEC} & : \quad P, \Gamma \vdash C_1 \Rightarrow C_2 \\
\text{R} & : \quad P, \Gamma \vdash C_1 \Rightarrow C_2 \\
\text{L} & : \quad \Gamma, C_1 \vdash Q \Rightarrow A \\
\text{E} & : \quad \Gamma, \forall \beta.C \vdash Q \Rightarrow A \\
\text{Tractable Constraint Entailment} & : \quad \Gamma, [\forall \beta.C] \vdash Q \Rightarrow \bullet \\
\end{align*}
\]

If we interpret constraints \( C \) as logical formulas, the above rules are nothing more than the rules of first-order predicate logic. Rule (\text{SPEC}) is the standard axiom rule. Rules (\text{VC}) and (\text{EC}) correspond to implication introduction and elimination, respectively. Similarly, Rules (\text{VIC}) and (\text{VEC}) 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 \( P = (\bullet, \bullet, Eqa) \), here are only two of the infinitely many proofs of \( P, \Gamma \vdash Eqa \):

\[
\text{SPEC} : \quad Eqa \in P \\
\text{IC} : \quad \Gamma \vdash Eqa \\
\text{EC} : \quad P, \Gamma \vdash Eqa
\]

versus

\[
\text{SPEC} : \quad Eqa \in P' \\
\text{IC} : \quad P', \Gamma \vdash Eqa \\
\text{EC} : \quad P, \Gamma \vdash Eqa
\]

where \( P' = P_a, Eqa \). 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 CoCns 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 \( P, \Gamma \vdash C \) is defined in terms of two auxiliary judgments, \( P, \Gamma \vdash [C] \) and \( \Gamma, [C] \vdash Q \Rightarrow A \), 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 \( P, \Gamma \vdash [C] \). 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 (\text{=R}) and (\text{VR}) decompose the goal by applying implication and quantifier introductions respectively. Once the goal is stripped down to a simple class constraint \( Q \), Rule (\text{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 of \( A \) of new (and hopefully simpler) goals whose entailment is checked recursively.

The second auxiliary judgment \( \Gamma, [C] \vdash Q \Rightarrow A \) 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 (\text{QL}) expresses the base case where the axiom is identical to the goal and there are no new goals. Rule (=R) 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 (\text{VL}) 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:

\[
\text{QL} : \quad \Gamma, [Eqa] \vdash Eqa \Rightarrow \bullet \\
\text{QR} : \quad P, \Gamma \vdash Eqa
\]

**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:

\[
\text{class Default a where } [\text{ default } :: a ] \\
\text{instance Default Bool where } [\text{ default } = \text{ True } ] \\
\text{instance Default Bool where } [\text{ default } = \text{ False } ]
\]

The two instances give rise to two different proofs for Default Bool, with distinct computational content (\text{True} vs. \text{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 \( E \).

\[
\begin{align*}
\text{class } & C a \ 	ext{where } \{ \ldots \} \\
\text{class } & D a \ 	ext{where } \{ \ldots \} \\
\text{class } & E a \ 	ext{where } \{ \ldots \}
\end{align*}
\]

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 \( D a \wedge \text{E} \) and \( E a \wedge \text{E} \), we have two ways to establish \( C \). 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 \( \tau \).

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

**Guessing Polymorphic Instantiation** A second source of ambiguity is that Rule (VI) requires guessing an appropriate type \( \tau \) 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.

\[
\begin{align*}
\text{instance } & C \text{ Char where } \{ \ldots \} \\
\text{instance } & C \text{ Bool where } \{ \ldots \} \\
\text{instance } & C a \Rightarrow D \text{ Int where } \{ \ldots \}
\end{align*}
\]

The third instance gives rise to the axiom \( \forall a. C a \Rightarrow D \text{ Int} \). When resolving \( D \text{ Int} \) with this axiom we can choose \( a \) to be either \( \text{Char} \) or \( \text{Bool} \) 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 \( \text{unamb}(C) \) in Figure 4 formalizes the requirement in terms of the auxiliary judgment \( \overline{a} \text{ unamb } C \), where \( \overline{a} \) are type variables that need to be determined by the head of \( C \). Rule (\( \text{Q} \)) constitutes the base case where \( Q \) is the head and contains the determinable type variables \( \overline{a} \). Rule (\( \text{VU} \)) processes a quantifier by adding the new type variable to the list of determinable type variables \( \overline{a} \). Finally, Rule (\( \Rightarrow \text{U} \)) 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 (\( \Rightarrow \text{B} \)); hence they must be well-behaved on their own.

The predicate \( \text{unamb}(C) \) must be imposed on all constraints that are added to the theory. This happens in four places: the instance axioms added in Rule \( \text{INSTANCE} \), the superclass axioms added in Rule \( \text{CLASS} \), the local axioms added when checking against a given signature in Rule (\( \Rightarrow \text{A} \)) and the local axioms added during constraint entailment checking in Rule (\( \Rightarrow \text{R} \)). 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 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:

\[
\begin{align*}
\mathcal{P} & \mapssto \langle \mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_L \rangle & \text{variable-annotated theory} \\
\mathcal{A} & \mapssto d \mid \mathcal{A}_1, \mathcal{A}_C & \text{variable-annotated axiom set} \\
\mathcal{C} & \mapssto d \mid \mathcal{C} & \text{variable-annotated constraint} \\
\mathcal{Q} & \mapssto d \mid \mathcal{Q} & \text{variable-annotated class constraint}
\end{align*}
\]

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 \( E \):

\[
\begin{align*}
E & \mapssto d \mid E, \tau_1 \sim \tau_2 & \text{type equalities}
\end{align*}
\]

**Type & Evidence Substitutions** Furthermore, we introduce two kinds of substitutions: type substitutions \( \theta \) and dictionary substitutions \( \eta \):

\[
\begin{align*}
\theta & \mapssto d \mid \theta \cdot [\tau / a] & \text{type substitution} \\
\eta & \mapssto d \mid \eta \cdot [\tau / d] & \text{evidence substitution}
\end{align*}
\]

A type substitution \( \theta \) maps type variables to monotypes, while an evidence substitution \( \eta \) 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 \( \Gamma \vdash \text{e} : \tau \Rightarrow d \mid \mathcal{A} \mid \mathcal{E} \). Given a typing environment \( \Gamma \) and a term \( e \) we infer (1) a monotype \( \tau \), (2) a set of wanted constraints \( \mathcal{A} \), and (3) a set of wanted equalities \( \mathcal{E} \). Its definition is standard.

Rule \( \text{TmVar} \) handles variables. We instantiate the polymorphic type \( \forall x . \mathcal{T} \Rightarrow \tau \) of a term variable \( x \) with fresh unification variables \( \overline{f} \), introducing \( \overline{c} \) as wanted constraints, instantiated likewise.

Rule \( \text{TmAbs} \) assigns a fresh unification variable to the abstracted
Figure 5. Constraint Generation for Terms with Elaboration

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash a : \tau</td>
<td>\mathcal{A}; E$</td>
</tr>
<tr>
<td>$\Gamma, \sigma \vdash a \rightarrow \tau</td>
<td>\mathcal{A}; E$</td>
</tr>
</tbody>
</table>

Figure 6. Constraint Entailment with Dictionary Construction

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{P}, C \models \mathcal{A} \Rightarrow \mathcal{A} : \eta$</td>
<td>Constraint Solving Algorithm</td>
</tr>
<tr>
<td>$\mathcal{P} \models \mathcal{A}, \mathcal{B} : \eta \Rightarrow \eta$</td>
<td>Constraint Simplification</td>
</tr>
<tr>
<td>$\mathcal{P} \models \mathcal{A} \Rightarrow \mathcal{A} : \eta$</td>
<td>Constraint Matching</td>
</tr>
</tbody>
</table>

4.3 Constraint Solving

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

Solving Equality Constraints

Solving Type Class Constraints

Figure 6 defines the judgment for solving type class constraints; it takes the form $\mathcal{P}, \mathcal{A} \models \mathcal{A} : \eta$.

Simplification

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

Rules $(\Rightarrow \mathcal{R})$ and $(\Rightarrow \mathcal{L})$ recursively simplify the head of the goal.

Observe that we add the bound variable $b$ to the untouched variables $\mathcal{C}$ when going under a binder in Rule $(\Rightarrow \mathcal{V})$. Once the goal is stripped down to a simple class constraint $\mathcal{C}$, Rule $(\Rightarrow \mathcal{Q})$ selects an axiom $\mathcal{C}$ 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 $\mathcal{P}, \mathcal{C} \models \mathcal{A} : \eta$, discussed below.

Matching

Auxiliary judgment $\mathcal{P}, \mathcal{A} \models \mathcal{A} : \eta$ focuses on the axiom $\mathcal{C}$ and checks whether it matches the simple goal $\mathcal{Q}$. The main difference between this algorithmic relation and its declarative specification in Figure 3 lies in the type substitution $\theta$. Instead of guessing a type for instantiating a polymorphic axiom in Rule $(\Rightarrow \mathcal{V})$ (top-down), we defer the choice until the head of the axiom is met, in Rule $(\Rightarrow \mathcal{Q})$ (bottom-up). Observe that Rule $(\Rightarrow \mathcal{V})$ does not record $b$ as untouched, effectively turning it into a unification variable. Thus, by unifying the head of the axiom with the goal we
can determine without guessing an instantiation for all top-level quantifiers, captured by the type substitution $\theta$.

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

\begin{align*}
\text{unify}(b; a \cdot b) = \theta &= \beta = \beta/b[a] \quad \text{(QL)} \\
\begin{array}{c} 
\forall a.\, Eq\, a \Rightarrow Eq\, [a] \Rightarrow Eq\, [b] \quad \text{(VR)} \\
\end{array} \quad \text{let} \quad d = f(v) \Rightarrow d \cdot d' \quad \begin{array}{c} 
\begin{array}{c} 
\forall a.\, Eq\, a \Rightarrow Eq\, [a] \Rightarrow Eq\, [b] \\
\forall b.\, Eq\, b \Rightarrow Eq\, [b] \\
\end{array} \quad \text{(VR)} \\
\end{array}
\end{align*}

\textbf{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 superclasses 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 class simple 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{align*}
\text{class} \ (E \ a \Rightarrow C \ a) \Rightarrow D \ a \\
\text{class} \ (G \ a \Rightarrow C \ a) \Rightarrow F \ a
\end{align*}

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
Quantified Class Constraints

Without loss of generality, we simplify $ma$...and elaboration for programs is straightforward. Rule TyQual elaborates a qualified type into a System F arrow type: the constraint $C$ is translated into the dictionary type $v_1$, via relation $\beta_1: C \mapsto v$ which performs elaboration of constraints:

$$\begin{align*}
\beta_1: C &\mapsto v \\
\beta_1: C_1 &\mapsto v_1 \\
\beta_1: C_2 &\mapsto v_2 \\
\beta_1: C &\mapsto v_1 \mapsto v_2
\end{align*}$$

Rule (CQ) elaborates a class constraint ($TC\tau$) into a type constructor application ($T_{TC}\tau$), which corresponds to the type of dictionaries that witness ($TC\tau$). Rule (CY) is straightforward. Rule (C⇒) elaborates implication constraints of the form ($C_1 \Rightarrow C_2$) into System F arrow types ($v_1 \mapsto v_2$), that is, types of dictionary transformers. As a concrete example, the constraint corresponding to the Show instance for type HPerf (Section 2.2):

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

is elaborated into the type

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

5.3 Elaboration of Terms

Term elaboration is straightforward. Rule TmVar handles term variables. The instantiation of the type scheme $\forall a. C \Rightarrow \tau \mapsto \exists a.\exists \tau \mapsto \tau$ becomes explicit in the System F representation, by the application of $x$ to type variables $\mathcal{E}$, as well as the fresh dictionary variables $\mathcal{D}$, corresponding one-to-one to the implicit constraints $\mathcal{C}$. Rule TmAbs elaborates $\lambda$-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 $v_1$ 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 $\eta$ in the simplification relation shows how to construct a witness for the wanted constraint $C$ from the simpler constraints $A'$ and program theory $P'$. The goal of Rule (⇒R) is to build an evidence substitution $\eta'$, which constructs a proof for ($d_0 : C_1 \Rightarrow C_2$) from the proofs $\widehat{d}'$ for the simpler constraint $C_1 \Rightarrow C$. It is instructive to consider the generated evidence substitution in parts, also taking the types into account:

1. $\eta$ illustrates how to generate a proof for ($d_2 : C_2$), from the local assumption ($d_1 : C_1$) and local residual constraints ($d : C$).

2. $\widehat{d'}d_1d_2[\eta]$ generates proofs for the (local) residual constraints ($d : C$), by applying the residual constraints ($d' : C_1 \Rightarrow C_1$) to the local assumption ($d_1 : C_1$).

3. ($\widehat{d'}d_1d_2[\eta](d_2)$) is a proof for $C_2$, under assumptions ($d_1 : C_1$) and ($\widehat{d'} : C_1 \Rightarrow C$).

4. Finally, we construct the proof for ($d_0 : C_1 \Rightarrow C_2$) by explicitly abstracting over $d_1 : \lambda (d_1 : v_1). [\widehat{d'}d_1d_2[\eta](d_2)]$ Rule (VR) proceeds similarly. Finally, Rule (QR) generates the evidence substitution via constraint matching, which we discuss next.
Accordingly, we generate
larly.

Section 2.1 gives rise to the following dictionary type:

Elaboration of Class Declarations A declaration for a class TC
is encoded in System F as a dictionary type
Elaboration of Class Declarations. This is elaborated via premise

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

Elaboration of Class Instances. This is elaborated via premise

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):
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 \( \forall a. \text{Ord} \ a \Rightarrow \text{Eq} \ a \) of the standard Haskell’98 \( \text{Ord} \) type class. Here both \( \text{Ord} \ a \) and \( \text{Eq} \ a \) 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

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

each type class \( TC_i \) is a superclass of \( TC \), where \( \text{head}(C_i) = TC_i \).

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 \( \text{Trans} \) class as:

```plaintext
Class \( \text{Trans} \) (\( \text{T} : \text{(Type -> Type)} \to \text{Type} \to \text{Type} \))
\( \exists(\forall m, (\forall \text{Monad} \ M) 
\Rightarrow \text{Monad} \ (\text{T} M)) \equiv
\{ \text{lift} : \forall A M, (\forall \text{Monad} \ M) 
\Rightarrow A M \Rightarrow (\text{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. \( \text{Monad}_t \ t \) for \( \text{forall} \ M. \text{Monad} \ m \Rightarrow \text{Monad} \ (t \ m) \), and that provides the implied methods under a new name. This expresses the \( \text{Trans} \) problem as follows:

```plaintext
class \( \text{Monad}_t \ t \) where
\text{tret} :: \text{Monad} \ m \Rightarrow a \to t \ m \ a
\text{tbind} :: \text{Monad} \ m \Rightarrow t \ m \ a 
\Rightarrow (a \to t \ m \ b) \to t \ m \ b
```

While this approach captures the intention of the quantified constraint, it does not enable the type checker to see that \( \text{Monad} \ (t \ m) \) 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 \( \text{Monad}_t \ t \) and \( \text{Trans} \) into a single class:

```plaintext
class \( \text{Monad}\_\text{T} \ t \) where
\text{tret} :: \text{Monad} \ m \Rightarrow a \to t \ m \ a
\text{ttret} :: \text{Monad} \ m \Rightarrow a \to t \ m \ a
\text{tbind} :: \text{Monad} \ m \Rightarrow t \ m \ a 
\Rightarrow (a \to t \ m \ b) \to t \ m \ b
```

The key novelty is that it also makes the methods \( \text{ttret} \) and \( \text{tbind} \) available under their usual name with a single \( \text{Monad} \) instance for all monad transformers.

```plaintext
instance \( \text{Monad} \ m, \text{Monad}\_\text{T} \ t \) \Rightarrow \text{Monad} \ (t \ m) \ where
\text{return} = \text{ttret}
(>\_>) = \text{tbind}
```

With these definitions the monad transformer composition does type check. Unfortunately, the head of the \( \text{Monad} \ (t \ m) \) 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 \( \text{mw} \) to the \( \text{Trans} \) type class:

```plaintext
class \( \text{Trans} \ t \) where
\text{tlift} :: \text{Monad} \ m \Rightarrow a \to t \ m \ a
\text{tlift} :: \text{Monad} \ m \Rightarrow \text{MonadWitness} \ m t a
```

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.

```plaintext
\text{data} \text{MonadWitness} \ (t :: (* \rightarrow *) \rightarrow (* \rightarrow *)) \ m \ where
\text{MW} :: \text{Monad} \ (t \ m) \Rightarrow \text{MonadWitness} t m
```

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

```plaintext
instance \( \text{Trans} t_1, \text{Trans} t_2) \Rightarrow \text{Trans} (t_1 * t_2) \ where
\text{tlift} :: \forall m. \text{Monad} m \Rightarrow a \to (t_1 * t_2) \ m \ a
\text{tlift} = \text{case} (\text{mw} :: \text{MonadWitness} t_2 m) \ of
\text{MW} \Rightarrow C \cdot \text{tlift} \cdot \text{tlift}
```

\( \text{mw} = \ldots \)

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.

COCHS

The calculus of coherent implicits, COCHS [32], and its focusing-based resolution in particular, have been a major inspiration of this work. Just like this work, COCHS supports recursive resolution of quantified constraints. Yet, there are a number of significant differences. Firstly, COCHS 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, COCHS 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 COCHS’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 predicates⁹, 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.

References


See GHC feature request #5927.

Haskell’17, September 7 – 8, 2017, Oxford, UK Bottu et al.
A Additional Judgments

A.1 Well-formedness of Types & Constraints

Well-formedness of types takes the form $\Gamma \vdash \tau$ and is given by the following rules:

$\tau \in \Gamma \vdash \tau \quad \text{TVar}$

$\Gamma \vdash \tau \quad \text{TyArr}$

$\Gamma \vdash \tau, \rho \quad \text{TyQual}$

$\Gamma \vdash A \quad \text{AxNil}$

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 $\Gamma \vdash C$, given by the following rules:

$\Gamma \vdash \tau \quad \text{CQ}$

$\Gamma \vdash C_1 \quad \text{C1}$

$\Gamma \vdash C_2 \quad \text{C2}$

$\Gamma \vdash \tau \quad \text{C\&}$

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

$\Gamma \vdash A \quad \text{AxCons}$

A.2 Program Typing

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

$\Gamma, \tau \vdash \text{CS} \quad \text{PGMCS}$

$\Gamma, \text{inst} : A \vdash \text{pgm} : \sigma \quad \text{PGMInst}$

$\Gamma, \text{inst} : \text{pgm} : \sigma \quad \text{PGMExpr}$

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

A.3 Unification Algorithm

The unification algorithm takes the form $\text{unify}(\alpha; \theta) = \theta_\alpha$ and is given by the following equations:

$\text{unify}(\alpha; \bullet) = \bullet$

$\text{unify}(\alpha, b \cdot b) = \text{unify}(\alpha, E)$

where $b \notin \alpha \land b \notin \theta$.

$\text{unify}(\alpha, t \sim \rho) = \text{unify}(\alpha, \theta(E)) \cdot \theta$

where $b \notin \alpha \land b \notin \theta$.

$\text{unify}(\alpha, E, \tau \sim \rho) = \text{unify}(\alpha, \theta(E)) \cdot \theta$

where $b \notin \alpha \land b \notin \theta$.

$\text{unify}(\alpha, E, \tau \sim \rho) = \text{unify}(\alpha, \theta(E)) \cdot \theta$

where $b \notin \alpha \land b \notin \theta$.

Function $\text{unify}$ 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 $\bar{V}$. 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 $P; \Gamma \vdash \text{pgm} : \sigma \Rightarrow \text{fpGM}$.

### Program Elaboration

$P; \Gamma \vdash \text{pgm} : \sigma \Rightarrow \text{fpGM}$

$\text{PCLS}$

$P; \Gamma \vdash \text{inst} : A \vdash \text{pgm} : \sigma \Rightarrow \text{fpGM}$

$\text{PLNS}$

Rules PCLS and PLNS handle class and instance declarations, respectively, and 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 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 $\Delta$:

$\Delta ::= \bullet \mid \Delta, \tau \mid \Delta, K : \text{v} \mid \Delta, a \mid \Delta, x : \text{v}$

B.1 Term Typing

$\Delta \vdash t : \text{v}$

$\text{Term Typing}$

$\Delta \vdash t : \text{v}$

$\Delta \vdash \text{v} \vdash \alpha : \text{v}$

$\Delta, \text{let} x : \text{v} = t_1 \text{ in } t_2 : \text{v}$

$\Delta \vdash \text{case } t : \text{of } \text{K} \rightarrow t_2 : \text{v}$

$\Delta \vdash t : \text{v}$

$\Delta \vdash \text{Type Well-formedness}$
B.3 Value Binding Typing

Value Binding Typing

\[ \Delta \vdash \text{let } x : v \mapsto t : \Delta \vdash v \mapsto t : \Delta. \]

B.4 Datatype Declaration Typing

Datatype Declaration Typing

\[ \Delta \vdash \text{data } a: T \mapsto \Delta. \]

B.5 Program Typing

Program Typing

\[ \Delta \vdash \text{pgm } \mapsto v \mapsto \Delta. \]

B.6 Call-by-name Operational Semantics

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

Operational Semantics (Small-step)

\[ (Aa.t) v \rightarrow [v/a]t \mapsto \Delta(x : v).t \rightarrow [t'/x]t \]

\[ t \rightarrow t' \]

\[ (\text{let } x : v = t_1 \text{ in } t_2) \rightarrow [\text{let } x : v = t_1 \text{ in } t_1/x]t_2 \]