KU LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN

Quantified Class Constraints in Haskell

Gert-Jan Bottu

Thesis submitted for the degree of
Master of Science in Engineering:
Computer Science, specialisation
Artificial Intelligence

Thesis supervisor:
Prof. dr. ir. Tom Schrijvers

Assessor:
Ir. George Karachalias,
Dr. ir. Dominique Devriese

Mentor:
Ir. George Karachalias

Academic year 2016 – 2017

Preface

For the past year, I've had the pleasure of writing a master thesis in the research group of Tom Schrijvers at the KU Leuven, as well as actually conducting interesting theoretical research in the field of Haskell. The work hasn't always been easy, and this text took more long days of work than I care to admit. But in the end, the thought of actually being able to do this fascinating research, as well as the unrelenting support of colleagues, friends and family enabled me to write this text.

More specifically, this text would not have been possible without the incredible enthousiasm, help and mentorship of George Karachalias, who stood by my side to help tackle even the most difficult of challenges, even outside the scope of this thesis. The same holds for my supervisor, Tom Schrijvers who was always ready to answer any questions and without whose advice none of this would have been possible. Furthermore, my sincerest gratitute goes out to my friends and family, whose support was absolutely crucial in helping me find the courage for writing this text.

And finally, thanks to you, the reader for reading this thesis. I sincerely hope this text proves to be as fruitful for you, as it was for me.

Gert-Jan Bottu

Contents

Abstract

Type classes have become one of the corner stones of the Haskell language, alongside several other (functional) programming languages. However, despite it having become omnipresent in functional programming, almost three decades of intensive research having gone into it, and countless extensions having been build on top of it, they are not without their limitations. While working on their Derivable Type Classes paper, Hinze and Peyton Jones encountered a set of, albeit perfectly reasonable instance declarations, which surprisingly could not be implemented in Haskell at the time. They proposed to extend the language with polymorphic constraints, which they called "quantified class constraints". These constraints would essentially boost the expressive power of type classes to first-order logic on types. They provided only a sketch of what the extension would look like, as well as a specification for typing.

Several workarounds and alternative encodings for adding the expressiveness, offered by quantified class constraints, without actually extending the Haskell type system, have been investigated. Unfortunately, none of them proved to be a worthy replacement of the actual language extension. Despite the extension being a much requested feature, no-one has thusfar successfully taken on the challenge of continuing upon their work, to the best of our knowledge.

Fifteen years after date, this thesis has elaborated on the idea of Hinze and Peyton Jones. We describe a formal specification of the extension and provide an algorithm for both type inference and elaboration into System F. The main challenge proved to be designing a coherent constraint entailment algorithm. For this, we borrowed the idea of focussing from the calculus of coherent implicits (Cochis), which describes a very similar language, supporting recursive resolution of quantified constraints in a Scala setting, by Schrijvers et al. [25] Furthermore, we investigated the meta-theory behind this extension and provide a prototype implementation of a Haskell compiler, extended with quantified class constraints.

List of Figures

Chapter 1

Introduction

Type classes were first introduced by Wadler and Blott [31], as a way of modeling ad-hoc polymorphism. Since then, type classes, or a derived form thereof, have been introduced in several mainstream programming languages. Examples of these include Haskell [31], Coq [27] and, in a more limited form, Rust [26, 6]. However, despite being widely used, both in academic and industrial applications, and despite years of intensive research surrounding type classes and extensions, type classes are still not without their limitations.

While working on their Derivable Type Classes paper [11], which discusses a way of introducing generic programming in Haskell, back in 2001, Hinze and Peyton Jones noted a specific type classes limitation. They encountered a set of, albeit perfectly reasonable, instance declarations, which simply could not be expressed in Haskell at the time, since they caused the constraint entailment algorithm to diverge. Their example is expressed in detail in Section 5.2. As a possible solution, they proposed to extend the Haskell type system with polymorphic constraints, which they called "quantified class constraints". These constraints would essentially boost the expressive power of type classes to first-order logic on types. They provided a sketch of what the extension would look like, as well as a specification for typing.

Unfortunately, neither ended up continuing their work on quantified class constraints. GHC feature request #2893 [1] was opened in 2008, to request these more flexible constraints in the language, but still isn't resolved today. A workaround for the derivable classes was found [17], without quantified class constraints. Several general alternative encodings [28] were investigated to simulate the behaviour and extra expressiveness of quantified class constraints, as explained in Chapter 5. Neither of these however, proved to be a full replacement for the actual language extension with quantified class constraints.

This thesis finally elaborates on the idea of Hinze and Peyton Jones for extending the Haskell type system with quantified class constraints. More specifically, the main contributions of this thesis are as following:

- The thesis text discusses all required background knowledge, useful for the reader to understand all important aspects of quantified class constraints. This

[1] https://ghc.haskell.org/trac/ghc/ticket/2893

includes the simply typed lambda calculus and System F in Chapter 2, the Hindley-Milner type system in Chapter 3 and type classes in Chapter 4.

- We investigate what extra expressivity quantified class constraints can offer, and look at their main benefits. We consider several different examples where quantified class constraints are useful, and divide them in two main classes. This is done is Chapter 5.

- We formalize the high level intuition from Hinze and Peyton Jones. This includes formalizing the type inference aspect, both into a specification and a type inference algorithm (Chapter 6). One of the main challenges from this thesis was solving these extended, flexible constraints. This constraint entailment aspect is explained in detail in Chapter 7.

- Furthermore, a dictionary based translation of programs into System F was designed for the extended language. This is discussed in Chapter 8.

- We elaborate on the meta-theoretical properties of the extended system. This includes, among others, termination of the type inference algorithm and soundness of the algorithm with respect to the specification. The meta-theory is described in Chapter 9 and proofs for a selection of these lemmas can be found in Appendix A.

- Finally, we provide a prototype implementation of a Haskell compiler, extended with quantified class constraints. This is done mainly for testing purposes and can be found at `https://github.com/gkaracha/quantcs-impl`.

Furthermore, Chapter 10 discusses the related work regarding quantified class constraints. This mainly includes: a) Discussing alternative environments in which quantified class constraints have already been investigated or even utilized in a production language. b) Discussing workarounds and alternative encodings for quantified class constraints, which offer very similar advantages and expressiveness.

Our main findings from this work, and a summary of the most interesting remaining future work is available in Chapter 11.

Based on this work, we submitted a paper [3] to Haskell Symposium 2017, in cooperation with Oliveira and Wadler. This paper can be found in Appendix C, including a Dutch translation in Appendix D.

Chapter 2

System F

Compilers often translate source code into an intermediate language, instead of immediately compiling into assembly language (x86, AMD64, etc.) or even machine code. These intermediary languages are generally more suitable (smaller) for applying optimizations and to reason about, compared to the complex original source language. Furthermore, the translation itself can be dramatically simplified by breaking it up into two simpler parts: source-to-mid (translation from source language to intermediate language) and mid-to-target (translation from intermediate language to assembly / bytecode), as to avoid the massive gap between the source and target language. This process of translating a program from one language into an equivalent program in another language, is known as transcompiling or transpiling. Examples of this include: a) The TypeScript transcompiler [1], which transpiles TypeScript into JavaScript. b) SmartEiffel [2] and EiffelStudio [3] (the two most commonly used Eiffel compilers), which compile Eiffel into (among others) C code. c) The Oracle Java compiler, which translates Java source code into Java bytecode to be run in the Java Virtual Machine. d) etc... System F [7] is the intermediate language of choice for functional languages, such as ML or Haskell.

The translation of Haskell expressions into System F expressions actually extends these terms with additional information. System F terms contain explicit type annotations, whereas those are not required in Haskell terms. A translation which extends expressions with additional information is known as an elaboration.

The elaboration of programs into System F, is a major aspect of our language extension. Because of this, this chapter provides the necessary background on System F. Before doing so, we first provide some information about the simply typed lambda calculus, since both System F and our language extension continue building on this basic calculus. Most of the information from this chapter originates from the Types and Programming Languages book, by Pierce [23].

[1] `https://www.typescriptlang.org/`, `https://en.wikipedia.org/wiki/TypeScript`
[2] `http://smarteiffel.loria.fr/`
[3] `https://www.eiffel.com/eiffelstudio/`, `https://en.wikipedia.org/wiki/EiffelStudio`

Parametric Polymorphism System F is an extension of the simply typed lambda calculus with parametric polymorphism. Polymorphism is an important concept in the field of computer science, allowing functions to be defined for multiple types of arguments. Different forms of polymorphism exist, but this thesis focusses on two of them: parametric and ad-hoc polymorphism. Parametric polymorphism allows for overloading functions, for different types of arguments, where the implementation does not depend on the type of argument. This as opposed to ad hoc polymorphism, in which several distinct implementations may exist, depending on the argument types. As an example of parametric polymorphism, consider the type signature for the `append` function:

```
append :: a -> [a] -> [a]
```

Since Haskell implicitly quantifies type variables, the full type is:

```
append :: forall a. a -> [a] -> [a]
```

The `a` type variable is abstracted over and the append function acts identically for any type `a`. This limits the implementation in that it can never make any assumptions regarding `a`. For instance pattern matching against the argument is not possible, since this would imply specifying the type `a`. This is opposed to ad hoc polymorphism, as explained in Chapter 4.

2.1 The Simply Typed Lambda Calculus

The lambda calculus is a great way of modeling programs, and formalizing properties regarding these computations. Unfortunately, the lambda calculus simply accepts any expression the user tries to model, including nonsensical ones. Consider for instance the following expression:

```
True 0
```

It is a valid term in the lambda calculus, but since `True` is not a function, it makes no sense applying an argument `0` to it. The term is not a value, but it cannot be evaluated any further. This is called a non-evaluating or stuck term.

The simply typed lambda calculus (STLC) extends the untyped lambda calculus with types, thus allowing to rule out non-evaluating terms. STLC offers type safety, which means that any well-typed term can continue evaluating, until it becomes a value. Differently put, a well-typed term can never reach a stuck state.

2.1.1 Syntax

The syntax for STLC is provided in Figure 2.1. A term can either be:

- A variable x.
- Term abstraction $\lambda(x : \tau).e$ abstracts over the variable x in term e.

- Term application $e_1\ e_2$ applies e_2 on the e_1 term.
- Let statement $\textbf{let}\ x : \tau = e_1\ \textbf{in}\ e_2$. A let statement differs from normal term abstraction in that it allows for recursively using x in e_1.

Using the definition from the figure, values can only be lambda expressions. However, in a real world example, more terms, such as `true`, `false` or natural numbers are often defined as being values as well. Types in STLC can either be function types or a type T, from a predefined set of base types (such as *Bool* or *Nat*). Lastly, the typing environment is a list of bindings from variables to their corresponding types.

As an example of a valid STLC expression, consider the following definition of a `map` function for natural numbers:

$$\begin{aligned}
&\textbf{let}\ \mathit{map} : (\mathit{Nat} \to \mathit{Nat}) \to \mathit{NatList} \to \mathit{NatList} \\
&\quad = \lambda(f : \mathit{Nat} \to \mathit{Nat}).\lambda(\mathit{ys} : \mathit{NatList}).\textbf{case}\ \mathit{ys}\ \textbf{of} \\
&\qquad\qquad \mathit{Nil} \to \mathit{Nil} \\
&\qquad\qquad \mathit{Cons}\ x\ \mathit{xs} \to \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs}) \\
&\quad \textbf{in}\ \mathit{map}\ (\lambda(x : \mathit{Nat}).x + 1)\ (\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil})))
\end{aligned} \tag{2.1}$$

The example assumes some additional syntax, such as pattern matching (explained in Section 2.3) and lists (conslists). `map` is a recursive function, which takes a function `f` and a list of natural numbers `ys`. It pattern matches on the latter and applies the function `f` on each of the elements, by recursively calling itself on the tail `xs`. `map` is then called on the successor function and the list [1, 2, 3].

$$\begin{array}{rcll}
e & ::= & x \mid \lambda(x : \tau).e \mid e_1\ e_2 \mid \textbf{let}\ x : \tau = e_1\ \textbf{in}\ e_2 & \textit{term} \\[1em]
v & ::= & \lambda(x : \tau).e & \textit{value} \\[1em]
\tau & ::= & T \mid \tau_1 \to \tau_2 & \textit{type} \\[1em]
\Gamma & ::= & \bullet \mid \Gamma, x : \tau & \textit{typing environment}
\end{array}$$

FIGURE 2.1: Simply typed lambda calculus syntax

2.1.2 Typing

Figure 2.2 shows the typing rules for STLC. Term typing $\Gamma \vdash_{\mathtt{tm}} e : \tau$ can be read as "*under the typing environment* Γ*, expression* e *has type* τ". Note that STLC uses explicit type annotations to fix the type of arguments for lambda abstractions and let bindings. The variable case is handled by the TmVar rule, which locates the type of `x` in the typing environment Γ. The ($\to$I) rule handles the term abstraction case. It assigns the appropriate function type to lambda abstractions. The result type corresponds to the type of the e term, in the extended environment with x (as declared by the explicit type annotation). Term application eliminates the function type by checking that the argument term has the correct type and assigning the result type of the function type to the entire expression, as decribed in the ($\rightarrow$E) rule.

Let Expression In this calculus we opted for a recursive let expression. Both the e_1 and e_2 expressions have access to the x variable. The TmLet rule thus type checks both the e_1 and e_2 terms in the extended environment, containing x. The type for e_2 is also the type for the global expression. Alternatively, another option would have been to not make the let expression recursive. Regular let expressions do not require separate typing or evaluation rules, since they can actually be decomposed into a term application and abstraction as follows:

$$\textbf{let } x : \tau = e_1 \textbf{ in } e_2 \equiv (\lambda(x : \tau).e_2)\ e_1$$

Unfortunately, since the language has no other methods of creating loops, besides recursion in let expressions, opting for regular let bindings, would mean a significant loss in expressivity.

As explained in Let Should Not Be Generalized [29], generalizing the let expression adds very little in expressiveness, but renders the system much more complex. In this text, we thus opt for using a monomorphic let expression, which means that `x` in a let binding can not have a polymorphic type.

Secondly, we adopt the Barendregt convention [1], which states that all variables in binders are unique. We follow both of these conventions throughout the remainder of this thesis text.

Example Typing Derivation Figure 2.3 shows the full typing derivation for the following STLC expression:

$$\textbf{let } f : Nat \rightarrow Nat = \lambda y : Nat.y + 1 \textbf{ in } f\ 2$$

In order to type check this example, we assume the language contains natural numbers, the corresponding Nat type, as well as typing rules for sum terms (TmSum) and natural numbers (TmNat):

$$\frac{\Gamma \vdash_{\texttt{tm}} x : Nat \qquad \Gamma \vdash_{\texttt{tm}} y : Nat}{\Gamma \vdash_{\texttt{tm}} x + y : Nat}\ \textsc{TmSum} \qquad \frac{}{\Gamma \vdash_{\texttt{tm}} n : Nat}\ \textsc{TmNat}$$

In order to type check a sum, both terms have to have type Nat. A natural number is always assigned the type Nat.

We discuss the derivation in a bottom-up fashion. The given term is type checked in the initial empty environment. The TmLet rule is applied first. f is checked not to be present in the (empty) environment, which trivially holds. The e_1 term $\lambda(y : Nat).y + 1$ is checked under the extended environment $\bullet, f : Nat \rightarrow Nat$, from now on named Γ_1. Next up, by applying ($\rightarrow$I), the inner part of the lambda expression e_1 is type checked in the extended environment $\bullet, f : Nat \rightarrow Nat, y : Nat$. This environment will be called Γ_2. The sum term is type checked using the TmSum rule. The left term y gets type Nat, through the TmVar rule which checks that y is present in the typing environment Γ_2. The right term 1 gets type Nat because of the TmNat rule. The sum term is thus well-typed in the Γ_2 environment and gets type Nat. This means the lambda term is well-typed as well with type $Nat \to Nat$.

In order for the let expression to be well-typed, the e_2 term $f\ 2$ has to be type checked as well in the Γ_1 environment. The $(\to\mathrm{E})$ rule is applied. f is well-typed under the Γ_1 environment since it is present in the typing environment with type $Nat \to Nat$, as described in the TmVar rule. Lastly, the TmNat rule is applied again to check that 2 is well-typed with type Nat. The term application and the initial let expression are thus well-typed with type Nat.

$\boxed{\Gamma \vdash_{\mathtt{tm}} e : \tau}$ Term Typing

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash_{\mathtt{tm}} x : \tau}\ \textsc{TmVar}$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{tm}} e : \tau_2}{\Gamma \vdash_{\mathtt{tm}} \lambda(x : \tau_1).e : \tau_1 \to \tau_2}\ (\to\mathrm{I}) \qquad \frac{\Gamma \vdash_{\mathtt{tm}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathtt{tm}} e_2 : \tau_1}{\Gamma \vdash_{\mathtt{tm}} e_1\ e_2 : \tau_2}\ (\to\mathrm{E})$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{tm}} e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{tm}} e_2 : \tau_2}{\Gamma \vdash_{\mathtt{tm}} (\mathbf{let}\ x : \tau_1 = e_1\ \mathbf{in}\ e_2) : \tau_2}\ \textsc{TmLet}$$

Figure 2.2: Simply typed lambda calculus typing

2.1.3 Operational Semantics

The operational semantics for STLC are provided in Figure 2.4. The first part describes small-step evaluation for a term. These rules describe a single step in the evaluation from a term to a value. The second part of the figure illustrates big-step evaluation of a term, which means applying the small-step evaluation rules iteratively until the term can not evaluate any further (since it no longer matches any one-step evaluation rule head). This text focusses on call-by-name semantics since they are very close to Haskell's lazy / call-by-need (call-by-name with memoization) operational semantics. The E-App rule describes how the left hand side of term applications is evaluated for a single step. When they reach a lambda function state, the whole term application is replaced by substituting x for the right hand side e_2 in e_1, as explained by rule E-AppAbs. E-Let descibes the evaluation of let bindings. They are evaluated similarly, but since they are recursive in the e_1 term, x is substituted for a new let binding $\mathbf{let}\ x : \tau_1 = e_1\ \mathbf{in}\ e_1$.

Example Term Evaluation Figure 2.5 shows the step-by-step evaluation of the example term from Section 2.1.2. The E-Let rule is applied first and replaces all

$$\dfrac{f \notin dom(\bullet) \quad \dfrac{y \notin dom(\Gamma_1) \quad \dfrac{\dfrac{}{y : Nat \in \Gamma_2}\,\textsc{TmVar} \quad \dfrac{}{\Gamma_2 \vdash_{\text{tm}} 1 : Nat}\,\textsc{TmNat}}{\Gamma_2 \vdash_{\text{tm}} y + 1 : Nat}\,\textsc{TmSum}}{\Gamma_1 \vdash_{\text{tm}} \lambda(y : Nat).y + 1 : Nat \to Nat}\,\to I \quad \dfrac{\dfrac{f : Nat \to Nat \in \Gamma_1}{\Gamma_1 \vdash_{\text{tm}} f : Nat \to Nat}\,\textsc{TmVar} \quad \dfrac{}{\Gamma_1 \vdash_{\text{tm}} 2 : Nat}\,\textsc{TmNat}}{\Gamma_1 \vdash_{\text{tm}} f\ 2 : Nat}\,\to E}{\bullet \vdash_{\text{tm}} \textbf{let } f : Nat \to Nat = \lambda(y : Nat).y\ 2 \textbf{ in } f\ 2 : Nat}$$

Figure 2.3: Simply typed lambda calculus example typing derivation

occurrences of f with the new let expression. Next up, E-App is applied to evaluate the left hand side of the term application. The actual evaluation happens by applying the E-Let rule again. Since there are no more occurrences of f in the term, the let expression is dropped and replaced by the inner lambda term. This term is then further evaluated using the E-AppAbs rule, which results in the term $2 + 1$, which reduces to 3.

$\boxed{e \to e'}$ Small-Step Term Evaluation

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \textsc{E-App} \qquad \frac{}{(\lambda(x:\tau).e_1)\ e_2 \to [e_2/x]e_1}\ \textsc{E-AppAbs}$$

$$\frac{}{\textbf{let}\ x:\tau_1 = e_1\ \textbf{in}\ e_2 \to [\textbf{let}\ x:\tau_1 = e_1\ \textbf{in}\ e_1/x]e_2}\ \textsc{E-Let}$$

$\boxed{e \to e'}$ Big-Step Term Evaluation

$$\frac{e_1 \to e_2 \qquad e_2 \to^* e_3}{e_1 \to^* e_3}\ \textsc{E-Step} \qquad \frac{\nexists e_2 : e_1 \to e_2}{e_1 \to^* e_1}\ \textsc{E-Stop}$$

Figure 2.4: Simply typed lambda calculus operational semantics

$$\begin{gathered}\textbf{let}\ f : Nat \to Nat = \lambda(y:Nat).y+1\ \textbf{in}\ f\ 2\\ \to_{\text{E-Let}} (\textbf{let}\ f : Nat \to Nat = \lambda(y:Nat).y+1\ \textbf{in}\ \lambda(y:Nat).y+1)\ 2\\ \to_{\text{E-App \& E-Let}} (\lambda(y:Nat).y+1)\ 2\\ \to_{\text{E-AppAbs}} 2+1 \to 3\end{gathered}$$

Figure 2.5: Simply typed lambda calculus example evaluation

2.1.4 Properties

The simply typed lambda calculus meets several interesting, desirable properties. The main ones are:

Progress The progress theorem states that all well-typed terms are either values or are able to evaluate further, using the single step operational semantics. A well typed term is thus never stuck, which would mean that it is not yet a value, but no more evaluation rules apply to evaluate it any further.

Theorem 1 *If* $\Gamma \vdash_{\mathtt{tm}} e : \tau$
then either e *is a value or* $\exists e' : e \to e'$

Preservation The preservation theorem means that when a well-typed term takes an evaluation step, the new resulting term is always well-typed as well.

Theorem 2 *If* $\Gamma \vdash_{\mathtt{tm}} e : \tau$ *and* $e \to e'$
then $\Gamma \vdash_{\mathtt{tm}} e' : \tau$

Type Safety The previous two theorems can be combined into the type safety theorem. This states that no well-typed term can ever get stuck. It will always evaluate into a value or keep evaluating forever.

Theorem 3 *If* $\Gamma \vdash_{\mathtt{tm}} e : \tau$ *and* $e \to^* e'$
then either e' *is a value or* $\exists e'' : e' \to e''$

Uniqueness of Types The uniqueness of types theorem states that in any given typing context, a term has at most one type. So if the term is well-typed, it has exactly one unique type.

Theorem 4 *If* $\Gamma \vdash_{\mathtt{tm}} e : \tau_1$ *and* $\Gamma \vdash_{\mathtt{tm}} e : \tau_2$
then $\tau_1 = \tau_2$

2.2 System F

System F builds on STLC by adding parametric polymorphism, as explained in the introduction.

2.2.1 Syntax

Figure 2.6 shows the syntax for System F. The reader can safely ignore the parts which are highlighted in grey, as data types will be explained in detail in Section 2.3. Note that the variable, term abstraction, term application and recursive let case are identical to those in STLC. However, since System F has added support for parametric type polymorphism, it contains some additional syntax for terms:

- Type abstraction $\Lambda a.t$ is very similar to term abstraction, except for the fact that a is now a type variable instead of a term variable. It thus abstracts over types, whereas term abstraction abstracts over terms.
- Type application is analogous to term application, except that it now applies a type to a term, instead of applying a term to a term.

System F types may take one of the following forms: type variables, function types, type abstraction and type application. Lastly, we chose to add programs as well in this representation of System F, for it to be more realistic and closer to a real world programming language. Programs consist of a list of global value bindings (and datatype declarations, which we explain in Section 2.3) and a single term, which has access to all previously defined value bindings.

$$
\begin{array}{lcl}
x, y, z, f & ::= & \langle \textit{term variable name} \rangle \\
a, b, c & ::= & \langle \textit{type variable name} \rangle
\end{array}
$$

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

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

$$
\upsilon ::= a \mid \upsilon_1 \to \upsilon_2 \mid \forall a.\upsilon \mid \colorbox{lightgray}{$T\ \upsilon$} \qquad \textit{type}
$$

$$
\Delta ::= \bullet \mid \Delta, x : \upsilon \mid \Delta, a \mid \colorbox{lightgray}{$\Delta, K : \upsilon$} \mid \colorbox{lightgray}{Δ, T} \qquad \textit{typing environment}
$$

FIGURE 2.6: System F syntax

As an example of a valid System F expression, consider again the `map` function. But instead of making the definition specific for natural numbers, System F allows us to make the function more general, using parametric polymorphism. This is possible since `map` never has to make any assumptions regarding its argument types. The definition is the following:

$$
\begin{aligned}
&\textbf{let } \mathit{map} : \forall a.\forall b.(a \to b) \to [a] \to [b] \\
&\quad = \Lambda a.\Lambda b.\lambda(f : a \to b).\lambda(\mathit{ys} : [a]).\textbf{case } \mathit{ys} \textbf{ of} \\
&\qquad\qquad\qquad \mathit{Nil} \to \mathit{Nil} \\
&\qquad\qquad\qquad \mathit{Cons}\ x\ \mathit{xs} \to \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs}) \\
&\qquad \textbf{in } \mathit{map}\ (\lambda(x : \mathit{Nat}).x + 1)\ (\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil})))
\end{aligned}
\tag{2.2}
$$

The `map` function can now be applied to lists of any type a, as long as it matches the argument type of the provided function $f : a \to b$, resulting in a list of type b. This is reflected in the type of `map`, which applies to any types a and b. Theses types are then abstracted over by the $\Lambda a.\Lambda b.e$ term, placing them in scope for the e expression.

2.2.2 Typing

Similarly to STLC, System F has explicit type annotations as well. Its typing rules are thus very similar. However, since System F allows for more complex type syntax, the notion of well-formed types is introduced and defined in Figure 2.7. In this system, types are well-formed if all type variables are declared using the $\forall$-operator. A more advanced representation would introduce kinds, as a way of checking the well-formedness and validity of types. Kinds operate very similarly to the way types are used to check and constraint terms. However, the inclusion of kinds would render this representation significantly more complex. Furthermore, they would bring us beyond the relevant scope of this thesis.

Term Typing The typing rules for System F are shown in Figure 2.8. The rules for variables (TmVar) and term application ($\rightarrow$E) are identical to those of STLC. Term abstraction ($\rightarrow$I) and let bindings (TmLet) only differ in that the provided type υ_1 for x is now checked for well-formedness. The ($\forall$I) rule handles type abstraction, by assigning it a $\forall a$ type and checking the type of t in the extended environment. Type application substitutes the type variable for the right hand side type υ, after checking the well-formedness for this type.

Program Typing Programs consist of a list of value bindings, and finally a single top-level expression. The second half of Figure 2.8 shows how type checking for programs works. The Val rule works similarly to the TmLet rule. The t term is type checked under the extended environment, since the let bindings are allowed to be recursive. The program typing relation will type-check the rest of the program in this new extended environment.

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

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

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

Figure 2.7: System F type well-formedness

2.2.3 Operational Semantics

Figure 2.9 shows the operational semantics for System F. The E-App, E-AppAbs and E-Let rules are identical to those for STLC. The rules for type application (E-TyApp & E-TyAppAbs) are analogous to those for term application.

2.3 System F with Data Types

This chapter introduces data types in System F. Data types make the language more convenient for the programmer (and make it look like a more realistic language). They ensure that we do not have to add basic types (such as Nat or Bool) to the language syntax and typing / evaluation rules, since they can be encoded using

$\boxed{\Delta \vdash^{F}_{\texttt{tm}} t : \upsilon}$ Term Typing

$$\frac{(x : \upsilon) \in \Delta}{\Delta \vdash^{F}_{\texttt{tm}} x : \upsilon}\ \textsc{TmVar} \qquad \frac{(K : \upsilon) \in \Delta}{\Delta \vdash^{F}_{\texttt{tm}} K : \upsilon}\ \textsc{TmCon}$$

$$\frac{\begin{array}{c}\Delta, x : \upsilon_1 \vdash^{F}_{\texttt{tm}} t : \upsilon_2 \\ x \notin \mathit{dom}(\Delta) \qquad \Delta \vdash^{F}_{\texttt{ty}} \upsilon_1\end{array}}{\Delta \vdash^{F}_{\texttt{tm}} \lambda(x : \upsilon_1).t : \upsilon_1 \to \upsilon_2}\ (\to\mathrm{I}) \qquad \frac{\begin{array}{c}\Delta \vdash^{F}_{\texttt{tm}} t_1 : \upsilon_1 \to \upsilon_2 \\ \Delta \vdash^{F}_{\texttt{tm}} t_2 : \upsilon_1\end{array}}{\Delta \vdash^{F}_{\texttt{tm}} t_1\ t_2 : \upsilon_2}\ (\to\mathrm{E})$$

$$\frac{\Delta, a \vdash^{F}_{\texttt{tm}} t : \upsilon \qquad a \notin \Delta}{\Delta \vdash^{F}_{\texttt{tm}} \Lambda a.t : \forall a.\upsilon}\ (\forall\mathrm{I}) \qquad \frac{\Delta \vdash^{F}_{\texttt{tm}} t : \forall a.\upsilon \qquad \Delta \vdash^{F}_{\texttt{ty}} \upsilon_1}{\Delta \vdash^{F}_{\texttt{tm}} t\ \upsilon_1 : [\upsilon_1/a]\upsilon}\ (\forall\mathrm{E})$$

$$\frac{\Delta, x : \upsilon_1 \vdash^{F}_{\texttt{tm}} t_1 : \upsilon_1 \qquad x \notin \mathit{dom}(\Delta) \qquad \Delta \vdash^{F}_{\texttt{ty}} \upsilon_1 \qquad \Delta, x : \upsilon_1 \vdash^{F}_{\texttt{tm}} t_2 : \upsilon_2}{\Delta \vdash^{F}_{\texttt{tm}} (\mathbf{let}\ x : \upsilon_1 = t_1\ \mathbf{in}\ t_2) : \upsilon_2}\ \textsc{TmLet}$$

$$\frac{\begin{array}{c}\Delta \vdash^{F}_{\texttt{tm}} t_1 : T\ \upsilon \\ (K : \forall a.\overline{\upsilon} \to T\ a) \in \Delta \qquad \Delta, \overline{x : [\upsilon/a]\upsilon} \vdash^{F}_{\texttt{tm}} t_2 : \upsilon_2 \qquad x \notin \mathit{dom}(\Delta)\end{array}}{\Delta \vdash^{F}_{\texttt{tm}} (\mathbf{case}\ t_1\ \mathbf{of}\ K\ \overline{x} \to t_2) : \upsilon_2}\ \textsc{TmCase}$$

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

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

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

$$\frac{\upsilon' = \forall a.\overline{\upsilon} \to T\ a \qquad T \notin \Delta \qquad K \notin \mathit{dom}(\Delta) \qquad \Delta, a \vdash^{F}_{\texttt{ty}} \overline{\upsilon}}{\Delta \vdash^{F}_{\texttt{val}} (\mathbf{data}\ T\ a\ \mathbf{where}\ K\ \overline{\upsilon}) : [T, K : \upsilon']}\ \textsc{Data}$$

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

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

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

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

FIGURE 2.8: System F typing

$\boxed{t \rightarrow^F t'}$ Term Evaluation

$$\frac{t_1 \rightarrow^F t_1'}{t_1\ t_2 \rightarrow^F t_1'\ t_2}\ \text{E-App} \qquad \frac{}{(\lambda x : \upsilon . t_1)\ t_2 \rightarrow^F [t_2/x]t_1}\ \text{E-AppAbs}$$

$$\frac{}{\mathbf{let}\ x : \upsilon_1 = t_1\ \mathbf{in}\ t_2 \rightarrow^F [\mathbf{let}\ x : \upsilon_1 = t_1\ \mathbf{in}\ t_1/x]t_2}\ \text{E-Let}$$

$$\frac{t \rightarrow^F t'}{t\ \upsilon \rightarrow^F t'\ \upsilon}\ \text{E-TyApp} \qquad \frac{}{(\Lambda a.t)\ \upsilon \rightarrow^F [\upsilon/a]t}\ \text{E-TyAppAbs}$$

$$\frac{t_1 \rightarrow^F t_1'}{\mathbf{case}\ t_1\ \mathbf{of}\ K\ \overline{x} \rightarrow t_2 \rightarrow^F \mathbf{case}\ t_1'\ \mathbf{of}\ K\ \overline{x} \rightarrow t_2}\ \text{E-Case}$$

$$\frac{}{\mathbf{case}\ (K\ \overline{t})\ \mathbf{of}\ K\ \overline{x} \rightarrow t_2 \rightarrow^F [\overline{t}/\overline{x}]t_2}\ \text{E-CaseCon}$$

Figure 2.9: System F operational semantics

data types and preloaded into the language. Since data types have been shown to be encodable using polymorphism, they do not add any extra expressive power. The syntax and typing / evaluation rules for data types were highlighted in grey in figures 2.6, 2.7, 2.8 and 2.9.

2.3.1 Syntax

As shown in Figure 2.6, terms may now also contain:

- Data constructors K, which are actually functions which construct a term of the corresponding type T.
- Case expressions, which allow for pattern matching against terms. Our case expressions only allow for a single branch, for brevity and simplicity reasons, but are easily extended to support multiple distinct branches.

A program may now consist of a series of both value bindings, as well as data type declarations, before its final expression. A data type declaration declares a new type and the corresponding data constructor (which is the only means of constructing terms of this type). The typing environment Δ can also bind data constructors to types, as well as storing type constructors. It can thus contain entries of the form $\Delta, K : \upsilon$ and Δ, T as well.

2.3.2 Typing

Figure 2.7 shows how the type well-formedness rules are extended with a single rule TyCon, which checks that a type constructor T is bound in the typing environment.

Figure 2.8 is extended with the TmCon rule, which works identically to the TmVar rule, for data constructors. The TmCase rule requires that all branches of the case statement have the same type υ_2. Branches t_2 are type checked in the extended environment where x is bound to the type υ, in which all type variables $\overline{a}$ from the T data type are filled in with the concrete types $\overline{\upsilon}$ from t_1.

Declarations Besides global value bindings and a final term, System F programs can also contain datatype declarations. A declaration **data** $T\ a$ **where** $K\ \overline{\upsilon}$ declares the new type $T\ a$. Furthermore, a data constructor K, which can potentially take a number of type arguments $\overline{\upsilon}$, is created. In order to construct a term of type $T\ \upsilon'$, for some υ', firstly a type υ' is applied to the data constructor K. Furthermore, a series of terms, with the corresponding types $[\upsilon'/a]\overline{\upsilon}$ have to be applied. The type of K thus is $\forall a.\overline{\upsilon} \rightarrow T\ a$. Both the type constructor T and the data constructor K are placed in the typing environment.

2.3.3 Operational Semantics

The operational semantics from Figure 2.9 are extended with two new rules for case expressions. The E-Case rule evaluates the term, which is pattern matched against. When it reaches weak head normal form (thus consisting of a data constructor, possibly with a series of terms applied to it), the E-CaseCon rule applies. The result is the branch term, in which all term variables from the pattern are instantiated with the corresponding terms from the term which is being pattern matched against.

2.4 Properties

Most important properties for STLC still hold in System F, despite for the fact that the language is much more powerful. Progress and preservation (and thus type safety as well) all continue to hold.

Type Inference System F is a very powerful language. However, writing explict type annotations becomes very tedious in a real world scenario. Unfortunately, automatically infering the argument types (instead of manually annotating them) is known to be undecibable for System F [32]. This undecidability means that no algorithm can be constructed which can infer the type of any (well-formed) expression. One of the factors that complicate matters for type inference, is the fact that System F allows for higer-rank polymorphic types [20]. Consider for instance the following example, using System F with tuples, booleans and natural numbers:

$$\lambda(id : \forall a.a \rightarrow a).(id\ True, id\ 0)$$

This term takes the identity function (which just takes any argument and returns it unchanged) and applies both a boolean value and natural number to it. The type of this term would be:

$$(\forall a.a \rightarrow a) \rightarrow Bool \times Nat$$

This is a higher-rank type, since a type containing the universal operator is nested within another larger type. Inference is known not to be decidable for higher rank polymorphic types. Another factor which makes type inference undecidable is the fact that System F is impredicative. Impredicativity means that type substitutions allow type variables to be instantiated with types, containing type abstraction. This way, first-order types can easily be made higer-rank.

Chapter 3

The Hindley-Milner Type System

System F is an incredibly powerful and expressive language. However, writing the explicit type annotations in a realistic coding scenario, becomes very cumbersome very fast. Unfortunately, as explained in Section 2.4, type inference is not decibable for a language as extensive as System F. Hindley, Milner and Damas introduced the Hindley-Milner system (HM) [5], which is a proper subset of System F. The language strikes a sweet spot between expressiveness and desirable properties. Despite being a powerful language, it still manages to offer both decidable and complete type inference. Because of this, type annotations are no longer required.

The language extension presented in this thesis indirectly extends upon HM. Furthermore, the type inference algorithm for quantified class constraints, which builds upon Algorithm W (presented in Section 3.3), is one of the main contributions of this thesis. This chapter thus presents the required background knowledge regarding HM, type inference and Algorithm W.

3.1 Syntax

The syntax for HM is provided in Figure 3.1. Terms are defined identically to STLC. HM still supports parametric type polymorphism, but since it does not require explicit type annotations, there is no need to explictly introduce type variables. There are thus no terms for explict type abstraction and application. Unlike STLC, types are now split in monotypes and polytypes. Monotypes are either type variables or function types, whereas a polytype is either a monotype or type abstraction. This way the syntax guarantees that no type abstractions appear in nested positions. This is the main difference between HM and System F.

Typing Environment Lastly, the typing environment is similar to that of System F. It contains both type variables and term variables bound to their corresponding types, as well as data constructors and type constructors. Note that data constructors are bound to a System F type, since the argument types correspond to methods, which may contain type abstraction, thus resulting in nested type abstractions. The language syntax does not allow the user to actually define new datatypes however. All datatypes in the language have to be predefined and preloaded into the environment. Extending the langage with user-defined datatypes would be straightforward to do, but would bring us outside of the scope for this thesis.

$$
\begin{array}{lll}
x, y, z, f & ::= & \langle \textit{term variable name} \rangle \\
a, b, c & ::= & \langle \textit{type variable name} \rangle
\end{array}
$$

$$
\begin{array}{lll@{\qquad}r}
e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \textit{term} \\
\\
\tau & ::= & a \mid \tau_1 \to \tau_2 & \textit{monotype} \\
\sigma & ::= & \tau \mid \forall a.\sigma & \textit{polytype} \\
\\
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, K : v \mid \Gamma, T & \textit{typing environment}
\end{array}
$$

Figure 3.1: Hindley-Milner syntax

3.2 Typing

Figure 3.2 shows the type well-formedness rules for HM types. Types are well-formed if all type variables are declared. Unlike STLC, HM does not require explicit type annotations. Because of this, the typing rules are not deterministic. They are shown in Figure 3.3. The rules are very similar to those for STLC. The main differences are:

- The term typing rules now assign polytypes to terms. Term variables are also bound to polytypes in the typing environment.
- For both term abstraction and let-binding typing, the argument type τ_1 is no longer explicitly provided within the term.
- The ($\forall$I) and ($\forall$E) rules are added, for introducing and eliminating type abstraction respectively. They are identical to the rules from System F.

$\boxed{\Gamma \vdash_{\mathsf{ty}} \sigma}$ Type Well-formedness

$$
\frac{a \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} a}\ \textsc{TyVar}
\qquad
\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \to \tau_2}\ \textsc{TyArr}
\qquad
\frac{\Gamma, a \vdash_{\mathsf{ty}} \sigma \qquad a \notin \Gamma}{\Gamma \vdash_{\mathsf{ty}} \forall a.\sigma}\ \textsc{TyAll}
$$

Figure 3.2: Hindley-Milner type well-formedness

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

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash_{\mathtt{tm}} x : \sigma}\ \textsc{TmVar}$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma \vdash_{\mathtt{ty}} \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{tm}} e : \tau_2}{\Gamma \vdash_{\mathtt{tm}} \lambda x.e : \tau_1 \to \tau_2}\ (\to\mathrm{I}) \qquad \frac{\Gamma \vdash_{\mathtt{tm}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathtt{tm}} e_2 : \tau_1}{\Gamma \vdash_{\mathtt{tm}} e_1\ e_2 : \tau_2}\ (\to\mathrm{E})$$

$$\frac{\Gamma, a \vdash_{\mathtt{tm}} e : \sigma \qquad a \notin \Gamma}{\Gamma \vdash_{\mathtt{tm}} e : \forall a.\sigma}\ (\forall\mathrm{I}) \qquad \frac{\Gamma \vdash_{\mathtt{tm}} e : \forall a.\sigma \qquad \Gamma \vdash_{\mathtt{ty}} \tau}{\Gamma \vdash_{\mathtt{tm}} e : [\tau/a]\sigma}\ (\forall\mathrm{E})$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{tm}} e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{tm}} e_2 : \sigma_2}{\Gamma \vdash_{\mathtt{tm}} (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) : \sigma_2}\ \textsc{TmLet}$$

FIGURE 3.3: Hindley-Milner typing

3.3 Inference

In a real world programming context, writing explicit type annotations quickly becomes very tedious. Type inference is the process where the compiler automatically derives the types of terms, thus freeing the programmer of this burden. Hindley-Milner is more restrictive, compared to System F, in that it does not accept higher rank polymorphic types (as explained in Section 2.4). Because of this limitation type inference becomes viable. HM supports both decidable and complete type inference. Decidability means that an algorithm exists, which for any give term (without type annotations), can determine its corresponding type or fail if the term is not well typed, in a finite number of steps. It can never go into an infinite loop. Furthermore, the algorithm can assign a type to any well typed term (completeness).

Algorithm W The type inference algorithm for Hindley-Milner is called Algorithm W. Algorithm W infers the type of terms in two distinct phases. The first part is described in the typing rules in Figure 3.4, which generate a set of equality constraints (of the form $\tau_1 \sim \tau_2$), as well as a type, containing unresolved type variables. The second phase then solves these constraints, to generate a substitution, which instantiates the type variables in the generated type.

For now, the reader can safely ignore the elaboration related parts (highlighted in grey), as they are not relevant for type inference and will be explained in detail later on (Section 3.5). The $\Gamma \vdash_{\mathtt{tm}} e : \tau \mid E$ relation, for the first phase, can be read as follows: "Under the given typing environment Γ, the given term e gets assigned the monotype τ, with the remaining constraints E." The type τ thus still contains unification variables which will be instantiated by solving the constraints E.

The TmVar rule looks up the variable x in the typing environment Γ. All free type variables of the corresponding type have to be declared, for the type to be well formed. The figure abuses notation in that it declares the full list of type variables at once. For all free variables, new unification variables $\overline{b}$ are created and all occurrences in the monotype τ are replaced by the new unification variables. The reason for this is that when the variable x is used in several locations throughout the code, they may be applied to multiple different types. When all occurrences of x would contain the same type variables, solving the constraints E might fail since the different applied types are likely not unifiable.

The TmLet rule gathers the types and constraints for both e_1 and e_2. Since let is recursive, both of these terms need to be typed in the extended environment where x is bound. However, when inferring the type of e_1, no type for x is available yet, since no explicit annotations exist and this is the type being infered right now. A new unification variable a is thus created and x is assigned to the type a. While inferring the type of e_2, the τ_1 is available and can be placed in the type environment. Later, a needs to be unified with the newly found type τ_1, so the $a \sim \tau_1$ equality is added to the list of constraints.
The TmAbs rule handles the lambda function case. It explains how a new unification variable a is created as the type for the argument x. The inner term e is typed under the extended environment where x is bound to type a. The resulting type for the lambda expression thus becomes the function type $a \to \tau$, where τ is the type of e. Lastly, the TmApp rule handles term application. The types of both terms are infered to be τ_1 and τ_2 respectively. τ_1 has to be a function type, where the argument type is τ_2 and the result type is yet unknown. A new type variable a is thus created to represent the result type and the constraint $\tau_1 \sim \tau_2 \to a$ is added.

$\boxed{\Gamma \vdash_{\texttt{tm}} e : \tau \rightsquigarrow t \mid E}$ Term Typing

$$\frac{\overline{b}\ \text{fresh} \qquad (x : \forall \overline{a}.\tau) \in \Gamma}{\Gamma \vdash_{\texttt{tm}} x : [\overline{b}/\overline{a}]\tau \rightsquigarrow x\ \overline{b} \mid \bullet}\ \textsc{TmVar}$$

$$\frac{\begin{array}{c} a\ \text{fresh} \qquad \Gamma, x : a \vdash_{\texttt{tm}} e_1 : \tau_1 \rightsquigarrow t_1 \mid E_1 \\ \Gamma, x : \tau_1 \vdash_{\texttt{tm}} e_2 : \tau_2 \rightsquigarrow t_2 \mid E_2 \end{array}}{\Gamma \vdash_{\texttt{tm}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2 \rightsquigarrow \textbf{let}\ x : \tau_1 = t_1\ \textbf{in}\ t_2 \mid (E_1, E_2, a \sim \tau_1)}\ \textsc{TmLet}$$

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

$$\frac{a\ \text{fresh} \qquad \Gamma \vdash_{\texttt{tm}} e_1 : \tau_1 \rightsquigarrow t_1 \mid E_1 \qquad \Gamma \vdash_{\texttt{tm}} e_2 : \tau_2 \rightsquigarrow t_2 \mid E_2}{\Gamma \vdash_{\texttt{tm}} e_1\ e_2 : a \rightsquigarrow t_1\ t_2 \mid E_1, E_2, \tau_1 \sim \tau_2 \to a}\ \textsc{TmApp}$$

Figure 3.4: Hindley-Milner type inference with elaboration

Equalities Solving Figure 3.5 describes the unification algorithm for solving equality constraints. The unfication algorithm takes a list of untoucheables $\overline{a}$ (the type variables which are not allowed to be unified) and a list of equality constraints E of the form $\tau_1 \sim \tau_2$, to produce a type substitution θ. However, as noted by the $\perp$ annotation, the algorithm might fail if the equality constraints prove to be unsolveable. The second phase of the type inference algorithm thus consists of creating the type substitution θ, and applying it to the infered type τ.

The type substitution is gradually created by recursively solving each equaltity constraint. Solving the $b \sim b$ constraint is trivial. Since both variables are already equal, no substitution has to be created. The $b \sim \tau$ constraint is solved by substituting all occurrences of b with τ. However, this constraint is only solveable if b is not in the untoucheables $\overline{a}$, otherwise it is not allowed to be unified with anything. Furthermore, in order to avoid infinite types, b is not allowed to occur in τ, since otherwise b could be replaced infinitely many times in τ, resulting in an infinitely long type, and b would never be completely removed from the type by applying the substitution. The $\tau \sim b$ constraint is solved identically to the previous one. Lastly, an equality constraint between two function types, can be split into two seperate equality constraints and solved one after the other. Since each other rule applies the resulting type substitution to the remaining constraints, the unified type variables from $\tau_2 \sim \tau_4$ are already instantiated in τ_1 and τ_3, before solving the second constraint.

$\boxed{\mathit{unify}(\overline{a}; E) = \theta_\perp}$ Type Unification Algorithm

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

FIGURE 3.5: Hindley-Milner equality constraint solving

3.4 Operational Semantics

Since the term syntax is completely identical to those for STLC, the operational semantics are identical as well. The reader can review these rules in Section 2.1.3.

3.5 Elaboration

As explained in Chapter 2, programming languages are often translated into an intermediate language before compiling into assembly or machine code. These intermediate languages are generally more suitable for applying optimizations. For functional languages, System F is the intermediate lanugage of choice. The process of translating the source language into an intermediate language, such as System F, is called transpiling. In the case of transpiling an HM expression into System F, this is called elaboration, since the process adds additional information to the terms, in the form of explicit type annotations.

The type inference rules from Figure 3.4 contain the elaboration to System F as well, highlighted in grey. The full $\Gamma \vdash_{\texttt{tm}} e : \tau \rightsquigarrow t \mid E$ relation now reads "Under the typing environment Γ, the given term e gets assigned the monotype τ, under constraints E. t is the translation of the term e in System F."

The TmVar rule looks up the type of x in the typing environment Γ. Unification variables $\overline{b}$ are created for each of the free variables $\overline{a}$ of the type of x. The variables are replaced by the unfication variables in the monotype τ. System F has explicit type abstraction. The term x is thus applied to the new types $\overline{b}$, to replace the type variables $\overline{a}$. The TmLet rule translates the let term, by using the elaborated terms t_1 and t_2. Since System F requires explicit type annotations, the x variable in the translated term is annotated with the τ_1 type. In fact this is the elaborated System F type, not the source language τ_1 type. However, since monotypes correspond one-on-one between the source and target language, this elaboration is not made explicit in the figure. Lambda terms are translated similarly, by the TmAbs rule. However, since no argument type is available there, the unification variable a is used as the type annotation. Lastly, the TmApp rule translates term application by using the elaborated terms t_1 and t_2.

Chapter 4

Type Classes

Ad-hoc Polymorphism As opposed to parametric polymorphism (as explained in Chapter 2), ad-hoc polymorphism allows a function to behave differently, depending on the type of arguments provided. In ad-hoc polymorphism, the function name is overloaded with several different implementations, which are selected based on the argument type.

As an example consider the sum function. The `+` operator is overloaded for different types, such as natural numbers, floating point numbers or even strings. The function behaves differently, based on the type of the argument. For instance summing a natural number and a floating point value results in a new floating point value. Whereas, summing two strings concatenates the two arguments.

Type Classes Haskell first introduced type classes, as a way of extending the Hindley-Milner type system with ad-hoc polymorphism [31]. Since Haskell does not keep track of typing information at runtime, simply pattern matching against a parametric argument type (type reflection) is not possible. However, type classes allow for a restricted, more static form of ad-hoc polymorphism. A type class typically consists of a type class name and a single type variable. Overloading is made possible by defining the overloaded function as a method of a type class. All types which instantiate this class, have to provide an implementation for all methods of this type class in their instance declaration.

A typical example for type classes is the `Eq` class, which models all types whose values can be compared for equality:

```
class Eq a where
  (==) :: a -> a -> Bool
```

The `(==)` method takes two expressions of type `a`, returns `True` if they are equal and `False` otherwise. A possible instance for this class is the following:

```
instance Eq Bool where
  (==) True  True  = True
  (==) False False = True
  (==) _     _     = False
```

This example declares the `Bool` type an instance of the `Eq` class and thus provides the accompanying method implementation.

Despite type classes having become omnipresent in many (functional) programming languages, and years of research having been performed on them, type classes are not without their limitations. This thesis introduces quantified class constraints as an extension for the type classes system in Haskell. This chapter first provides some overview and background knowledge on type classes.

4.1 Syntax

This chapter provides an overview of a small subset of Haskell, including type classes, which will be extended with the extension introduced by this thesis in Chapter 6. The syntax for the language is provided in Figure 4.1. Programs consist of a list of class and instance declarations, followed by a single term. The syntax for terms is identical to that for HM, as explained in Chapter 3.

A class declaration defines a new type class. It consists of a list of superclass constraints (as explained in Section 4.5), a type class name, a single type variable and the name and type of a single method f. This thesis opted for a single variable and single method representation of type classes, for brevity of both the syntax and typing rules. This could however easily be extended into a more flixble approach with multiple argument type classes and classes containing multiple different methods.

Every instance corresponds to an axiom, for this type class. An instance declaration contains the class name and type τ for which the instance holds. An implementation for the class method f, which matches the σ type from the class declaration, in which a is instantiated with τ, is provided. Lastly, a series of constraints C can be contained, under which the instance holds. They are called logical assumptions. If the instance context is empty, the resulting axiom is a simple axiom. Consider for instance the following declaration: `instance Eq Bool where ...`, which corresponds to the `Eq Bool` axiom. If not, the resulting axiom corresponds to a logical implication, or deduction rule. For example, the following declaration: `instance Eq a => Eq [a] where ...` corresponds to the following logical implication axiom: `forall a . Eq a => Eq [a]`

The next sections will assume there are no superclasses. All superclass related parts will be explained in Section 4.5.

As with HM, types are split into different categories. Besides monotypes and type schemes (polytypes), qualified types [13] are now added as well. Qualified types may contain several required constraints and finally a monotype. Consider for instance the equality type, constructed from the `Eq` class:

```
class Eq a where
    (==) :: a -> a -> Bool
```

The corresponding type in the typing environment is: `forall a . Eq a => a -> a -> Bool`. The full type is a polytype, consisting of type abstraction and a qualified type `Eq a => a -> a -> Bool`. The latter consists of a single implication, and a monotype `a -> a -> Bool`.

Constraints are defined to only be class constraints $TC\ \tau$. Whereas an axiom set is defined to be a list of constraints. The typing environment is identical to that of HM. New however, is the theory P. The program theory is a triple of constraint scheme lists: superclass constraints, instance constraints and local constraints. The reason for this is explained in Section 4.5. Appending an axiom to the program theory is annotated as $A,_{\mathrm{S}}\ C$, $A,_{\mathrm{I}}\ C$ and $A,_{\mathrm{L}}\ C$ for appending C to the superclass constraints, instance constraints or local constraints respectively.

$$\begin{array}{lcl}
x, y, z, f & ::= & \langle \textit{term variable name} \rangle \\
d & ::= & \langle \textit{term variable name for dictionary} \rangle \\
a, b, c & ::= & \langle \textit{type variable name} \rangle \\
TC & ::= & \langle \textit{class name} \rangle
\end{array}$$

$$\begin{array}{lcll}
pgm & ::= & e \mid cls; pgm \mid inst; pgm & \textit{program} \\
cls & ::= & \mathbf{class}\ A \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \} & \textit{class decl.} \\
inst & ::= & \mathbf{instance}\ A \Rightarrow TC\ \tau\ \mathbf{where}\ \{\ f = e\ \} & \textit{instance decl.}
\end{array}$$

$$\begin{array}{lcll}
e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 & \textit{term}
\end{array}$$

$$\begin{array}{lcll}
\tau & ::= & a \mid \tau_1 \to \tau_2 & \textit{monotype} \\
\rho & ::= & \tau \mid C \Rightarrow \rho & \textit{qualified type} \\
\sigma & ::= & \rho \mid \forall a.\sigma & \textit{type scheme}
\end{array}$$

$$\begin{array}{lcll}
A & ::= & \bullet \mid A, C & \textit{axiom set} \\
C & ::= & Q & \textit{constraint} \\
Q & ::= & TC\ \tau & \textit{class constraint}
\end{array}$$

$$\begin{array}{lcll}
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, K : \upsilon \mid \Gamma, T & \textit{typing environment} \\
P & ::= & \langle \overline{R_S}, \overline{R_I}, \overline{R_L} \rangle & \textit{program theory} \\
R & ::= & \forall \overline{a}.A \Rightarrow C & \textit{constraint scheme}
\end{array}$$

FIGURE 4.1: Type classes syntax

4.2 Typing

Figure 4.2 presents the term typing, type well-formedness and constraint entailment rules. The TMVAR, TMLET, ($\to$I), ($\to$E), ($\forall$I) and ($\forall$E) rules are identical to those for HM. ($\Rightarrow$I) introduces the constraint implication type. The e term is further type checked under the exteded program theory, containing the C constraint. The ($\Rightarrow$E) rule does the exact opposite by dropping the C constraint after checking that the constraint is entailed by the program theory P. The fact that C is already entailed, means that it can be derived by using available axioms and implications from the program theory P and can thus be safely dropped.

Type well-formedness is mostly identical to HM. The only new rule is TyQual, which checks the well-formedness of qualified types. A qualified type $C \Rightarrow \rho$ is well-formed if ρ is well formed. The C constraint is simply a class constraint, which means that checking the well-formedness of a contraint simply means checking the well-formedness of the contained monotype under the current environment.

As illustrated in the third part of the figure, constraints are well formed under the given typing enviroment Γ when the contained monotype is well-formed under this environment.

The fourth and last part of the figure are the constraint entailment rules. Axiom sets are entailed by the given program theory P and the typing environment Γ, when each of their contained constraints are checked to be entailed. When checking the entailment of a constraint, a matching constraint scheme is searched in the program theory. A matching constraint scheme is defined as having an identical constraint head, except for the fact that in the wanted constraint the free variables are instantiated with (well-formed) types.

Declarations The typing rules for class and instance declarations are provided in Figure 4.3. The Class rule checks the well-formedness of the method type σ, with the type variable a in scope. The rule creates a new typing environment for class declarations, containing the method name and corresponding type. In order to call the method, the argument type has to be provided, as well as a proof that $TC\ a$ holds. The type reflects this. The rule abuses notation in that in realitiy for the type to be well-formed, all type abstractions in σ should be placed before the $TC\ a$ constraint, but it's straightforward to transform it accordingly.

The Instance rule checks that the type of the method implementation matches the type provided in the class declaration, where type variable a is substituted with type τ. The rule creates a new program theory as well, containing the $TC\ \tau$ axiom, given that proofs for all constraints in C are provided.

Lastly, the PgmCls and PgmInst rules type check programs by checking class and instance declarations respectively. The new typing environments and program theories are then used to recursively check the rest of the program. Finally, the PgmExpr rule checks the final term under the program theory and typing environment which was build up by the previous rules.

4.3 Type Inference & Elaboration

Similarly to the HM system, type inference consists of two distinct phases. The first part of the type inference algorithm, the constraint generation aspect, is presented in Figure 4.4. The elaboration parts are highlighted in grey. Type classes are elaborated into System F using a dictionary based translation. A dictionary is a data type containing the implementation of the class method. A class declaration is thus translated into the declaration of a new dictionary data type and a global value

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

$$\frac{(x : \sigma) \in \Gamma}{P;\Gamma \vdash_{\texttt{tm}} x : \sigma}\ \textsc{TmVar} \qquad \frac{P;\Gamma, x : \tau \vdash_{\texttt{tm}} e_1 : \tau \qquad P;\Gamma, x : \tau \vdash_{\texttt{tm}} e_2 : \sigma}{P;\Gamma \vdash_{\texttt{tm}} (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : \sigma}\ \textsc{TmLet}$$

$$\frac{P;\Gamma, a \vdash_{\texttt{tm}} e : \sigma}{P;\Gamma \vdash_{\texttt{tm}} e : \forall a.\sigma}\ (\forall\mathrm{I}) \qquad \frac{\begin{array}{c}P;\Gamma \vdash_{\texttt{tm}} e : \forall a.\sigma \\ \Gamma \vdash_{\texttt{ty}} \tau\end{array}}{P;\Gamma \vdash_{\texttt{tm}} e : [a \mapsto \tau]\sigma}\ (\forall\mathrm{E})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\texttt{ty}} \tau_1 \\ P;\Gamma, x : \tau_1 \vdash_{\texttt{tm}} e : \tau_2\end{array}}{P;\Gamma \vdash_{\texttt{tm}} \lambda x.e : \tau_1 \to \tau_2}\ (\to\mathrm{I}) \qquad \frac{\begin{array}{c}P;\Gamma \vdash_{\texttt{tm}} e_1 : \tau_1 \to \tau_2 \\ P;\Gamma \vdash_{\texttt{tm}} e_2 : \tau_1\end{array}}{P;\Gamma \vdash_{\texttt{tm}} e_1\ e_2 : \tau_2}\ (\to\mathrm{E})$$

$$\frac{P,_{\mathrm{L}}\ C;\Gamma \vdash_{\texttt{tm}} e : \rho}{P;\Gamma \vdash_{\texttt{tm}} e : C \Rightarrow \rho}\ (\Rightarrow\mathrm{I}) \qquad \frac{\begin{array}{c}P;\Gamma \vdash_{\texttt{tm}} e : C \Rightarrow \rho \\ P \models C\end{array}}{P;\Gamma \vdash_{\texttt{tm}} e : \rho}\ (\Rightarrow\mathrm{E})$$

$\boxed{\Gamma \vdash_{\texttt{ty}} \sigma}$ Type Well-formedness

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

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

$\boxed{\Gamma \vdash_{\texttt{ct}} C}$ Constraint Well-formedness

$$\frac{\Gamma \vdash_{\texttt{ty}} \tau}{\Gamma \vdash_{\texttt{ct}} TC\ \tau}\ \mathrm{C}Q$$

$\boxed{P;\Gamma \models A}$ Axiom Set Entailment

$$\frac{\overline{P;\Gamma \models C_i}^{C_i \in A}}{P;\Gamma \models A}\ \mathrm{STEP}$$

$\boxed{P;\Gamma \models A \rightsquigarrow A'}$ Constraint Entailment

$$\frac{(\forall \overline{a}.A' \Rightarrow TC\ \tau_1) \in P \qquad \overline{\Gamma \vdash_{\texttt{ty}} \tau}}{P;\Gamma \models TC\ [\overline{\tau}/\overline{a}]\tau_1}\ \mathrm{QC}$$

FIGURE 4.2: Type classes typing

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

$$\frac{\Gamma, a \vdash_{\texttt{ax}} A \qquad \Gamma, a \vdash_{\texttt{ty}} \sigma}{\Gamma \vdash_{\texttt{cls}} \textbf{class}\ A \Rightarrow TC\ a\ \textbf{where}\ \{\ f :: \sigma\ \} : \overline{\forall a. TC\ a \Rightarrow C_i}^{C_i \in A} ; f : \forall a. TC\ a \Rightarrow \sigma}\ \textsc{Class}$$

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

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

$\boxed{P_1 ; \Gamma_1 \vdash_{\texttt{pgm}} \mathit{pgm} : P_2 ; \Gamma_2 ; \sigma}$ Program Typing

$$\frac{\Gamma_1 \vdash_{\texttt{cls}} \mathit{cls} : A_S ; \Gamma_{\mathit{cls}} \qquad P_1,_{\mathrm{S}} A_S ; \Gamma_1, \Gamma_{\mathit{cls}} \vdash_{\texttt{pgm}} \mathit{pgm} : P_2 ; \Gamma_2 ; \sigma}{P_1 ; \Gamma_1 \vdash_{\texttt{pgm}} (\mathit{cls} ; \mathit{pgm}) : P_2 ; \Gamma_2 ; \sigma}\ \textsc{PgmCls}$$

$$\frac{P_1 ; \Gamma_1 \vdash_{\texttt{inst}} \mathit{inst} : A_I \qquad P_1,_{\mathrm{I}} A_I ; \Gamma_1 \vdash_{\texttt{pgm}} \mathit{pgm} : P_2 ; \Gamma_2 ; \sigma}{P_1 ; \Gamma_1 \vdash_{\texttt{pgm}} (\mathit{inst} ; \mathit{pgm}) : P_2 ; \Gamma_2 ; \sigma}\ \textsc{PgmInst}$$

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

Figure 4.3: Type classes typing continued

binding for the class method. An instance declaration is then elaborated into a new dictionary for this class and type.

Since type inference and elaboration to System F are performed simultaneously, constraints are annotated with their corresponding System F dictionary variable d. As not to introduce new symbols, the same letters from before are reused, yet in a calligraphic font, as follows:

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

The well-formedness of types is identical to that in Figure 4.2. The added elaboration aspect works similarly to that of HM, as explained in Section 3.2. The only new part is the translation of qualified types, which are translated into function types in System F. A constraint has to be satisfied by explicitly passing in a dictionary of the appropriate type.

As illustrated in the third part of the figure, a constraint is translated into the corresponding dictionary data type.

The full term typing relation now becomes $\Gamma \vdash_{\texttt{tm}} e : \tau \rightsquigarrow t \mid \mathcal{P} ; E$, which is read as follows: "Under the typing environment Γ, the term e gets assigned the monotype

τ and is translated into the System F term t. In order for the term to be well typed, the constraints in $\mathcal{P}$ need to hold. The unification variables can be substituted by solving the equality constraints E." The TmVar rule handles the term variable case. It looks up the type scheme corresponding to x in the typing environment. New type variables $\overline{b}$ are generated and substituted for the free variables $\overline{a}$ in the monotype τ. Because of this, the variable x can be used multiple times throughout the code, applied to different types, without interfering with each other. New dictionary variables $\overline{d}$ (System F term variables representing dictionaries) are created as well, to be able to provide the dictionaries for the $\overline{C}$ constraints. The $\overline{d}$ variables are then placed in the wanted constraints list with the constraints they are supposed to witness.

Constraint Solving The second phase of type inference is the constraint solving aspect. Unlike regular HM however, two distinct sorts of constraints emerge here. Firstly, the equality constraints E are solved identically to those in HM. The reader can review this aspect in Section 3.3. Secondly, the generated type substitution can be applied to the set of type class constraints P, before solving these. Figure 4.8 explains the constraint solving algorithm. Solving a single constraint takes the form: $\overline{a}; P \models C \rightsquigarrow A'; \eta$, which may be read as "Under the program theory P, the constraint C is simplified into A'".

Firstly, a matching constraint in searched in the program theory P. A matching constraint is defined to be a constraint scheme, where the constraint head matches the wanted constraint, in that their contained types can be unified.

A substitution θ is then created by unifying the two constraints. The free variables of the wanted constraint $TC\ \tau_1$ are in the provided untoucheable type variables $\overline{a}$. Because of this, the type variables in τ_2 can be instantiated with concrete variables from τ_1, but not the other way around. For instance in the program theory:

$$P = \langle \bullet, \bullet, \texttt{Eq (a , Bool)} \rangle$$

the constraint `Eq (Nat , Bool)` would be entailed, but not the constraint `Eq (Nat , b)`.

The remaining constraints A' are returned, after instantiating the free variables using θ. Together with it, a dictionary substitution η is provided. This substitution shows how to construct a proof for $(d : TC\ \tau_1)$ from the program theory P and the simpler constraints A'.

In the basic version of this system, without superclasses, no backtracking is required since no overlapping constraints may exist in the program theory.

The constraint entailment algorithm for axiom sets actually recursively simplifies the given axiom set A as far as possible. It does not necessarily completely solve the set. A constraint C is selected from the given set, and simplified into A_2, as well as a dictionary substitution η, which maps the proof for C onto proofs for the constraints in A_2. The substitution is applied to the remaining constraints, before recursively solving those as far as possible. This way, a proof for the original constraints will be constructed by solving increasingly simpler constraints.

Because we allow constraints to be partially entailed, simplification of top-level signatures can be performed. This is standard practice when infering types in Haskell. Consider for instance:

```
f x = [x] == [x]
```

The derived constraint `Eq [a]` will be simplified to `Eq a` in Haskell, thus yielding the signature `forall a . Eq a => a -> Bool`.

Declarations The inference rules for class and instance declarations are presented in Figure 4.5. The Class rule works identically to the one in Figure 4.3, except that it now elaborates the class declaration as well.

A class declaration is elaborated into three System F declarations. The first one is the declaration of a new data type for the class dictionaries. The second one binds the f method. It takes a type for the type variable a and explicitly takes the corresponding dictionary, from which it takes the actual method implementation by projecting on it. The projection function $\mathit{proj}^i_{TC}(d)$ pattern matches against d and takes out the i-th argument out of the dictionary, as follows:

$$\mathit{proj}^i_{TC}(d) = \textbf{case}\ d\ \textbf{of}\ K_{TC}\ \overline{x} \rightarrow x_i$$

Because of this, $\mathit{proj}^{n+1}_{TC}(d)$ takes out the method implementation, since n is the number of superclass constraints in A_S and thus the number of superclass dictionaries in d. The third and last one is explained in Section 4.5.

The Instance rule looks up the class declaration for the given instance. The type of the method implementation is then matched against the method signature, where the type variable a is instantiated with τ. This matching is done through the subsumption rule, as explained below. This is performed under the extended program theory, where the context constraints from the instance declaration are added as local axioms, and the instance axiom itself is added as an instance axiom. Futhermore, the instance declaration is transformed into an axiom, abstracting over the free variables of τ and containing an implication over all context constraints. This axiom is returned in the axiom set $\mathcal{A}_I$. An instance declaration is translated into a single System F value binding, creating the corresponding dictionary. The dictionary takes types to substitute each of the free variables of τ, as well as dictionaries for each of the constraints $\overline{C}$.

The elaboration of programs is illustrated in Figure 4.7. Class and instance declarations are handled iteratively by applying the rules from Figure 4.5 and building up the available program theory and environment. Elaborating the final expression requires typing the expressing under the accumulated typing environment. The equality constraints are resolved, resulting in a type substitution θ. This substitution is applied to generated class constraints, which are then simplifed as far as possible, resulting in the remaining constraints $\overline{(d : C)}$ and a dictionary substitution η, mapping the proofs for $\theta(\mathcal{A})$ onto those for the simplified constraints. The type of the final expression (and thus of the entire program) is thus $\forall \overline{a}.\overline{(C)} \Rightarrow \theta(\tau)$. The elaborated term abstracts over the remaining free variables in $\theta(\mathcal{A})$ and $\theta(\tau)$ and explicitly takes dictionaries for each of the remaining constraints. Note thus, that when the set of remaining constraints is not empty, the resulting System F expression cannot be executed, before providing dictionaries for each of these axioms.

Subsumption Figure 4.6 explains how a term can be checked with an explicit polytype annotation (originating from the class method signature). Firstly, the type τ_1 of e is infered. The equality constraints are extended with the $\tau_1 \sim \tau_2$ constraint, to create a type substitution θ. Under this substitution, τ_1 should thus match the head τ_2 of the assigned type signature. The class constraints have to be completely entailed, with the context constraints $\overline{C}$ from the type signature in store. We say the type τ_1 is subsumed by the given type signature polytype, meaning that the given type signature is a more general version of the τ_1 type. The term is elaborated into a System F term, by applying both the type and dictionary substitutions, abstracting over the free variables of τ_1 and taking explicit dictionaries for each of the context constraints of the type signature.

4.4 Operational Semantics

The language with type classes does not itself offer operational semantics. Their operational semantics can only be provided with respect to the elaborated program. The execution of System F terms is explained in Section 2.3.3. The source language program itself can thus not be executed.

4.5 Superclasses

Superclasses are a widespread extension of the common type classes system, as presented above. Adding superclasses means that no instance can be given for this class unless the corresponding superclass constraints are satisfied. A list of superclass constraints is thus added to the class declaration. Furthermore, the final expression in a program is typed by simplifying the constraints, using only instance and local constraints in the theory. Because of this, the program theory is split in a triple of constraints, to differentiate between them.

The Class rule in Figure 4.3 now also checks the well-formedness of superclass constraints and returns a list of superclass schemes. The Instance rule needs to check whether all constraints from the superclass requirements are satisfied, instantiated for the given instance parameter.

Figure 4.4 presents the algorithm typing rules. The elaboration for class and instance declarations is shown in figure 4.5. The former now also returns an extra set of System F declarations: a global value binding for each of the superclass constraints, which can take out the contained superclass dictionary out of another dictionary for this class. This way, an extra way of solving entailments arises: a constraint may be satisfied by its subclass. An example of this is provided below.

For the instance declaration to make sense, all superclass constraints have to be completely entailed by the current program theory, the local constraints from

$\boxed{\Gamma \vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon}$ Type Well-formedness

$$\frac{a \in \Gamma}{\Gamma \vdash_{\mathtt{ty}} a \rightsquigarrow a}\ \textsc{TyVar} \qquad \frac{\Gamma \vdash_{\mathtt{ty}} \tau_1 \rightsquigarrow \upsilon_1 \qquad \Gamma \vdash_{\mathtt{ty}} \tau_2 \rightsquigarrow \upsilon_2}{\Gamma \vdash_{\mathtt{ty}} \tau_1 \to \tau_2 \rightsquigarrow \upsilon_1 \to \upsilon_2}\ \textsc{TyArr}$$

$$\frac{\Gamma \vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon_1 \qquad \Gamma \vdash_{\mathtt{ty}} \rho \rightsquigarrow \upsilon_2}{\Gamma \vdash_{\mathtt{ty}} C \Rightarrow \rho \rightsquigarrow \upsilon_1 \to \upsilon_2}\ \textsc{TyQual} \qquad \frac{a \notin \Gamma \qquad \Gamma, a \vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon}{\Gamma \vdash_{\mathtt{ty}} \forall a.\sigma \rightsquigarrow \forall a.\upsilon}\ \textsc{TyAll}$$

$\boxed{\Gamma \vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon}$ Constraint Well-formedness

$$\frac{\Gamma \vdash_{\mathtt{ty}} \tau \rightsquigarrow \upsilon}{\Gamma \vdash_{\mathtt{ct}} TC\ \tau \rightsquigarrow T_{TC}\ \upsilon}\ QC$$

$\boxed{\Gamma \vdash_{\mathtt{ax}} A \rightsquigarrow \overline{\upsilon}}$ Axiom Well-formedness

$$\frac{\Gamma \vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon \qquad \Gamma \vdash_{\mathtt{ax}} A \rightsquigarrow \overline{\upsilon}}{\Gamma \vdash_{\mathtt{ax}} A, C \rightsquigarrow \overline{\upsilon}, \upsilon}\ AC$$

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

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

$$\frac{\begin{array}{c} a\ \text{fresh} \qquad \Gamma, x : a \vdash_{\mathtt{tm}} e_1 : \tau_1 \rightsquigarrow t_1 \mid \mathcal{P}_1; E_1 \\ \Gamma, x : \tau_1 \vdash_{\mathtt{tm}} e_2 : \tau_2 \rightsquigarrow t_2 \mid \mathcal{P}_2; E_2 \end{array}}{\Gamma \vdash_{\mathtt{tm}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \rightsquigarrow \mathbf{let}\ x : \tau_1 = t_1\ \mathbf{in}\ t_2 \mid (\mathcal{P}_1, \mathcal{P}_2); (E_1, E_2, a \sim \tau_1)}\ \textsc{TmLet}$$

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

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

Figure 4.4: Type classes type inference

$\boxed{\Gamma_1 \vdash_{\texttt{cls}} cls : \mathcal{A}_S; \Gamma_2 \rightsquigarrow fdata; \overline{fval}}$ Class Declaration Typing

$$\dfrac{\begin{array}{c} \Gamma, a \vdash_{\texttt{ty}} \sigma \rightsquigarrow \upsilon \qquad \Gamma, a \vdash_{\texttt{ax}} A_S \rightsquigarrow \overline{\upsilon_S} \qquad d, \overline{d_i}\ \text{fresh} \\ \mathcal{A}'_S = \overline{d_i : \forall a. TC\ a \Rightarrow C_i}^{C_i \in A_S} \qquad fdata = \mathbf{data}\ T_{TC}\ a = K_{TC}\ \overline{\upsilon}^n\ \upsilon \\ fval = \mathbf{let}\ f : (\forall a. T_{TC}\ a \to \upsilon) = \Lambda a.\lambda(d : T_{TC}\ a).proj^{n+1}_{TC}(d) \\ \overline{fval_i = \mathbf{let}\ d_i : (\forall a. T_{TC}\ a \to \upsilon_S) = \Lambda a.\lambda(d : T_{TC}\ a).proj^{i}_{TC}(d)} \end{array}}{\Gamma \vdash_{\texttt{cls}} (\mathbf{class}\ A_S \Rightarrow TC\ a\ \mathbf{where}\ \{\ f : \sigma\ \}) : \mathcal{A}'_S; [f : \forall a. TC\ a \Rightarrow \sigma] \rightsquigarrow fdata; fval, \overline{fval_i}}\ \textsc{Class}$$

$\boxed{\mathcal{P}_1; \Gamma \vdash_{\texttt{inst}} inst : \mathcal{A}_I \rightsquigarrow fval}$ Class Instance Typing

$$\dfrac{\begin{array}{c} \mathbf{class}\ A_S \Rightarrow TC\ a\ \mathbf{where}\ \{\ f : \sigma\ \} \qquad \overline{b} = fv(\tau) \qquad \overline{d}, \overline{d_i}, d_I\ \text{fresh} \\ \overline{b}; \mathcal{P},_{\text{L}} (\overline{d : A}),_{\text{I}} (d_I : \forall \overline{b}. A \Rightarrow TC\ \tau); \Gamma, \overline{b} \vdash_{\texttt{tm}} e : [\tau/a]\sigma \rightsquigarrow t \\ \Gamma, \overline{b} \vdash_{\texttt{ax}} A \rightsquigarrow \overline{\upsilon_A} \\ \overline{b}; \mathcal{P},_{\text{L}} (\overline{d : A}),_{\text{I}} (d_I : \forall \overline{b}. A \Rightarrow TC\ \tau) \models (\overline{d_i} : [\tau/a]A_S) \rightsquigarrow \bullet; \eta \\ fval = \mathbf{let}\ d_I : (\forall \overline{b}. \overline{\upsilon_A} \to T_{TC}\ \tau) = \Lambda \overline{b}.\lambda \overline{(d : \upsilon_A)}.K_{TC}\ \tau\ \overline{\eta(d_i)}\ t \end{array}}{\mathcal{P}; \Gamma \vdash_{\texttt{inst}} (\mathbf{instance}\ A \Rightarrow TC\ \tau\ \mathbf{where}\ \{\ f = e\ \}) : [d_I : \forall \overline{b}. A \Rightarrow TC\ \tau] \rightsquigarrow fval}\ \textsc{Instance}$$

FIGURE 4.5: Type classes type inference continued

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

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

FIGURE 4.6: Type classes subsumption rule

the instance and the instance axiom itself. We can apply this generated dictionary substitution η to the dictionary variables for the data constructor, to construct the required proofs for $\overline{d_i}$ using axioms from the previously mentioned extended program theory.

As is common practice for Haskell, superclass axioms are not used to simplify wanted constraints for simplicity reasons, as can be seen in the PGMEXPR rule from Figure 4.7. For instance the constraint set `{Eq a, Ord a}`, could be simplified to `{Ord a}`, using the fact that `Ord` is a subclass of `Eq`. This would however render the inference algorithm significantly more complex and could result in problems for determinicy, since a single class can have multiple distinct subclasses.

Lastly, one of the main changes occurs in the rules of Figure 4.8, which presents

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

$$\frac{\Gamma \vdash_{\texttt{cls}} \mathit{cls} : \mathcal{A}_S; \Gamma_c \rightsquigarrow \mathit{fdata}; \overline{\mathit{fval}} \qquad \mathcal{P},_{\mathrm{S}} \mathcal{A}_S; \Gamma, \Gamma_c \vdash_{\texttt{pgm}} \mathit{pgm} : \sigma \rightsquigarrow \mathit{fpgm}}{\mathcal{P}; \Gamma \vdash_{\texttt{pgm}} (\mathit{cls}; \mathit{pgm}) : \sigma \rightsquigarrow \mathit{fdata}; \overline{\mathit{fval}}; \mathit{fpgm}} \textsc{PgmCls}$$

$$\frac{\mathcal{P}; \Gamma \vdash_{\texttt{inst}} \mathit{inst} : \mathcal{A}_I \rightsquigarrow \mathit{fval} \qquad \mathcal{P},_{\mathrm{I}} \mathcal{A}_I; \Gamma \vdash_{\texttt{pgm}} \mathit{pgm} : \sigma \rightsquigarrow \mathit{fpgm}}{\mathcal{P}; \Gamma \vdash_{\texttt{pgm}} (\mathit{inst}; \mathit{pgm}) : \sigma \rightsquigarrow \mathit{fval}; \mathit{fpgm}} \textsc{PgmInst}$$

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

Figure 4.7: Type classes type inference for programs

$\boxed{\overline{a}; P \models A_1 \rightsquigarrow A_2; \eta}$ Recursive Constraint Solving

$$\frac{\overline{a}; P \models C \rightsquigarrow A_2; \eta_1 \qquad \overline{a}; P \models \eta(A_1, A_2) \rightsquigarrow A_3; \eta_2}{\overline{a}; P \models A_1, C \rightsquigarrow A_3; \eta_2 \cdot \eta_1} \text{STEP}$$

$$\frac{\nexists C \in A_1 : \overline{a}; P \models C \rightsquigarrow A_2; \eta}{\overline{a}; P \models A_1 \rightsquigarrow A_1; \bullet} \text{STOP}$$

$\boxed{\overline{a}; P \models C \rightsquigarrow A'; \eta}$ Constraint Solving

$$\frac{\begin{array}{c}(d' : \forall \overline{b}. A' \Rightarrow TC\ \tau_2) \in P \\ \overline{d''} \text{ fresh} \qquad \theta = \mathit{unify}(\overline{a}; \tau_1 \sim \tau_2) \qquad \eta = [d'\ \theta(\overline{b})\ \overline{d''}/d]\end{array}}{\overline{a}; P \models (d : TC\ \tau_1) \rightsquigarrow \theta(\overline{d'' : A'}); \eta} \text{QC}$$

Figure 4.8: Type classes constraint solving algorithm

the constraint solving algorithm. Before, no overlapping constraints could exist in the program theory, so there was no need for backtracking while selecting a matching constraint scheme from P, since at most one could match the wanted constraint. However, when superclasses are available this property no longer holds. Consider for instance the example:

```
class Eq a where
  (==) :: a -> a -> Bool
class Eq a => Ord a where
  (<=) :: a -> a -> Bool

instance Eq Bool where
  (==) = ...
instance Ord Bool where
```

```
  (<=) = ...
```

This code fragment would result in the following program theory:

$$P = \langle (\forall \mathtt{a}.\mathtt{Ord\ a} \Rightarrow \mathtt{Eq\ a}); (\mathtt{Eq\ Bool}, \mathtt{Ord\ Bool}); \bullet \rangle$$

Since there is now overlap for the `Eq Bool` constraint, it could now be entailed in two separate ways: either directly by the instance rule or it could be simplified to a subclass of `Eq`, the `Ord Bool` contraint, using the superclass relation. In this scenario both approaches would work, but this is not always the case (consider for instance the case where no instance declaration for `Ord Bool` exists). Backtracking has to be introduced in case the first matching constraint would fail later on, while recursively checking its required constraints. Furthermore, the entailment rule should remain deterministic. To achieve this, we define the rule to always prefer the instance constraints when possible.

Chapter 5

Motivation for Quantified Class Constraints

This chapter illustrates the added expressive power, provided by quantified class constraints. We separate the discussed examples into two distinct groups: a) The extra expressivity allows to express several instance requirements much more exactly. This provides extra information for the type system, but also makes the code much more resembling to the system which the programmer has in mind, thus making the program more intuitive. b) The more flexible constraints and the new entailment algorithm, provide termination for several code examples where the constraint solver would diverge before.

5.1 Precise Specifications

Monad Transformer Firstly, consider the well-known monad transformer class [14]:

```
class Trans t where
    lift :: Monad m => m a -> t m a
```

A monad transformer `t` takes any monadic value `m a` and lifts `a` into a new monad `t m`. However, the requirement that `t m` is a monad, for any monad `m` is not expressible in current Haskell. This requirement thus has to remain implicit or be expressed in comments. In either case, the type system does not have access to this knowledge.

Now consider the monad zipper datatype [24], as introduced by Schrijvers et al.:

```
newtype (t1 * t2) m a = C { runC :: t1 (t2 m) a }

instance (Trans t1, Trans t2) => Trans (t1 * t2) where
    lift = C . lift . lift
```

The zipper datatype is thus an extensive form of monad transformer composition. The `lift` implementation first lifts `m a` into `(t2 m) a`, and then into `t1 (t2 m) a`. The first `lift` application is valid, since we know `Monad m` from the constraint on the `lift` implementation we are currently defining. However, in order for the second `lift` to be valid, we need to know that `t2 m` is a monad. Unfortunately, the type checker has no way of verifying this.

This problem can easily be solved using quantified class constraints, since they allow to make the initial property of the transformer `t` explicit, as follows:

```
class (forall m . Monad m => Monad (t m)) Trans t where
    lift :: Monad m => m a -> t m a
```

The axiom `Monad (t2 m)` is now available and the instance declaration for the monad zipper typechecks. Alternative workarounds for this example exist as well (as shown in Chapter 10), but they render the code significantly longer and more cluttered.

Second-Order Functors In his Adjoint Folds and Unfolds [10] work, Hinze represents datatypes as the fixpoint `Mu` of a second-order functor. The `Mu` fixpoint is defined as follows:

```
data Mu h a = In { out :: h (Mu h) a }
```

It takes a second-order functor `h` and applies it to `Mu` of itself. A second-order functor takes functors into another functor. `Mu` can be used to represent datatypes as follows. Consider for instance the `List` datatype:

```
data List a = Nil | Cons a (List a)
```

Alternatively, `List` can be represented by abstracting over the recursive collection `f` and then taking the fixpoint of itself:

```
data List' f a = Nil | Cons a (f a)
type List = Mu List'
```

This representation is extremely useful for representing nested datatypes, as described by Johann and Ghani [2]. Nested datatypes are a form of irregular datatypes where, unlike `List`, the contained datatype is not homogeneous. For each node deeper in the nested datatype, the tail is 'nested' deeper. As an example, consider the perfect binary tree datatype [9]:

```
data Perfect a = Zero a | Succ (Perfect (a , a))
```

At each layer in the tree, the tail of the datatype is nested in a one layer deeper tuple, compared to the parent. This can again be represented by abstracting over the recursive collection and then taking the fixpoint of itself:

```
data HPerf f a = HZero a | HSucc (f (a , a))
type Perfect = Mu HPerf
```

The constraint that `h` is a second-order functor, and thus lifts functors into functors, can easily be expressed using quantified class constraints. Consider for instance the `Functor` instance of `Mu`:

```
instance (forall f . Functor f => Functor (h f)) => Functor (Mu h) where
    fmap f (In x) = In (fmap f x)
```

However, as Hinze notes:

> *Unfortunately, the extension has not been implemented yet. It can be simulated within Haskell 98 [28], but the resulting code is somewhat clumsy.*

5.2 Terminating Corecursive Resolution

Recursive Resolution The notion of Quantified class constraints was first introduced by Hinze and Peyton Jones [11]. They discovered a set of, seemininlgy reasonable, instance declarations which would cause the type checker to diverge. Consider their example, the `GRose` datatype:

```
data GRose f a = GBranch a (f (GRose f a))

instance (Show a, Show (f (GRose f a))) => Show (GRose f a) where
    show (GBranch x xs) = show x ++ " - " ++ show xs
```

The datatype abstracts over the collection `f`. Each branch thus consists of a single value of type `a` and an `f` collection of `GRose` terms. The `Show` instance for the `GRose` datatype requires two constraints to hold. `Show a` is required for `show x`, and `Show (f (GRose f a))` is required for `show xs`. Unfortunately, even though this instance declaration is correct, the current constraint entailment algorithm would diverge while trying to check, for instance, the constraint `Show (GRose [] Bool)`:

```
    Show (GRose [] Bool)
|-> Show Bool, Show [GRose [] Bool]
|-> Show Bool, Show (GRose [] Bool)
|-> Show Bool, Show Bool, Show [GRose [] Bool]
|-> ...
```

Solving this constraint means recursively checking the instance context. Firstly, `Show Bool` is easily satisfied. Checking `Show [GRose [] Bool]` requires the top level `Show` instance for lists:

```
instance Show a => Show [a] where
    show = ...
```

The resulting constraint to be checked is again exactly the constraint we were trying to prove in the first place. The classic constraint entailment thus loops.

They introduced the concept of quantified class constraints to circumvent this loop, and encode the context constraints as follows:

```
instance (Show a, forall x . Show x => Show (f x)) => Show (GRose f a) where
    show (GBranch x xs) = show x ++ " - " ++ show xs
```

This solution completely avoids the loop, since both required constraints are immediately resolved with the available instances for `Show Bool` and `forall a . Show a => Show [a]`.

Cycle-aware Constraint Resolution While working on their Scrap Your Boilerplate paper [17], Lämmel and Peyton Jones encountered a similar problem. They decided to address it by using a completely different solution, namely implementing cycle-aware constraint resolution [16]. This approach detects when one of the recursive goals is identical to one of its ancestors. When this happens, the algorithm 'ties the (co)recursive knot'. Unfortunately this approach only accepts a very specific group of diverging resolutions, namely those that cycle in such a way that the same constraint occurs multiple times in the derivation. It thus solves the `GRose` example loop.

Non-Cyclic Loops Consider however a `Show` instance declaration for the `HPerf` datatype, as defined above:

```
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 ++ ")"
```

The context constraints are required for calling `show x`, `show a` and `show xs` respectively. Resolving the `Show (Perfect Int)` constraint, goes as follows:

```
    Show (Perfect Int)
|-> Show (Mu HPerf Int)
|-> Show (HPerf (Mu HPerf) Int)
|-> Show Int, Show (Mu HPerf (Int,Int))
|-> Show Int, Show (HPerf (Mu HPerf) (Int,Int))
|-> Show Int, Show (Int,Int), Show (Mu HPerf ((Int,Int),(Int,Int)))
|-> ...
```

The first step originates from the fact that `Perfect` is a type synonym for `Mu HPerf`. The `Show Int` can be immediately dismissed. Using the following for level declaration, the `Show (Int,Int)` can be reduced to `Show Int`:

```
instance (Show a, Show b) => Show (a, b) where
    show = ...
```

It is however straightforward to see that the constraint entailment algorithm diverges, without cycling back to the original constraint. The cycle-aware constraint resolution thus will not be able to avoid the infinite loop. On the other hand, we can easily express this instance declaration using quantified class constraints:

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

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

The constraint entailment derivation now becomes:

```
    Show (Perfect Int)
|-> Show (Mu HPerf Int)
|-> Show Int, forall f x. (Show x, Show y => Show (f y)) => Show (HPerf f x)
```

The `Show Int` constraint is again easily dismissed. The `forall f x. (Show x, Show y => Show (f y)) => Show (HPerf f x)` constraint now exactly matches the generated axiom from the `Show (HPerf f a)` instance declaration and is thus immediately resolvable as well.

5.3 Summary

Extending the Haskell type system with quantified class constraints results in two distinct groups of applications, which now become expressable:

- Quantified class constraints result in a bigger portion of the type class specification becomming formally encodeable. This extra information for the type system can result in more applications becoming expresseable.
- Quantified class constraints also result in terminating type class resolution for an additional group of applications.

Several workarounds for quantified class constraints have been investigated, but none proved to be a full replacement for the actual extension. Quantified class constraints are still very much a useful and widely requested feature.

Chapter 6

Quantified Class Constraints

6.1 Syntax

Figure 6.1 presents the syntax for the extended language. The main difference with the calculus with type classes, as described in Section 4.1, is the definition of constraints. Constraints are now extended to consist of a class constraint, constraint implication or type abstraction. Because of this more flexible syntax, the constraint schemes from the type classes system are now no longer required, since constraints themselves are already more flexible than the constraint schemes were. The program theory thus consists of a triple of axiom sets. The difference between a quantified class constraint and a normal type classes constraint scheme, is that in the constraint scheme all type abstractions have to happen at the top level, followed by a series of implications and finally the class constraint head. Whereas with the more flexible, new constraints, implications and universal quantification can appear in arbitrarily nested positions.

The reader may notice that unlike some earlier formalizations (such as [18]), we do not include conjunctions in the syntax of the constraints. The reason for this is that conjuctions are simply syntactic sugar for a curried series of constraints:

$$\forall \overline{a}.(C_1 \wedge ... \wedge C_n) \Rightarrow C \quad \equiv \quad \forall \overline{a}.C_1 \Rightarrow (... \Rightarrow (C_n \Rightarrow C)...)$$

6.2 Typing

The well-formedness rules for constraints are provided in Figure 6.2. They are mostly straightforward. A class constraint is well-formed if its type parameter is well-formed under the same enviroment. For an implication to be well-formed, both sides have to be checked. And finally, a type abstraction is well-formed under the current environment if a is fresh with respect to Γ and the remaining constraint C is well-formed under the extended environment.

The rules for type well-scopedness, term and declaration typing are completely identical to those for regular type classes, as explained in Section 4.2.

$$
\begin{array}{lcl}
x, y, z, f & ::= & \langle \textit{term variable name} \rangle \\
\colorbox{lightgray}{d} & ::= & \langle \textit{term variable name for dictionary} \rangle \\
a, b, c & ::= & \langle \textit{type variable name} \rangle \\
TC & ::= & \langle \textit{class name} \rangle
\end{array}
$$

$$
\begin{array}{lcll}
pgm & ::= & e \mid cls; pgm \mid inst; pgm & \textit{program} \\
cls & ::= & \mathbf{class}\ A \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \} & \textit{class decl.} \\
inst & ::= & \mathbf{instance}\ A \Rightarrow TC\ \tau\ \mathbf{where}\ \{\ f = e\ \} & \textit{instance decl.}
\end{array}
$$

$$
\begin{array}{lcll}
e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 & \textit{term}
\end{array}
$$

$$
\begin{array}{lcll}
\tau & ::= & a \mid \tau_1 \to \tau_2 & \textit{monotype} \\
\rho & ::= & \tau \mid C \Rightarrow \rho & \textit{qualified type} \\
\sigma & ::= & \rho \mid \forall a.\sigma & \textit{type scheme}
\end{array}
$$

$$
\begin{array}{lcll}
A & ::= & \bullet \mid A, C & \textit{axiom set} \\
C & ::= & Q \mid C_1 \Rightarrow C_2 \mid \forall a.C & \textit{constraint} \\
Q & ::= & TC\ \tau & \textit{class constraint}
\end{array}
$$

$$
\begin{array}{lcll}
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, K : \upsilon \mid \Gamma, T & \textit{typing environment} \\
P & ::= & \langle A_S, A_I, A_L \rangle & \textit{program theory}
\end{array}
$$

Figure 6.1: Quantified class constraints syntax

$\boxed{\Gamma \vdash_{\mathsf{ct}} C}$ Constraint Well-formedness

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

Figure 6.2: Constraint well-formedness

6.3 Type Inference & Elaboration

Type inference and elaboration for both terms and (class and instance) declarations remain completely identical to the algorithm provided in Sections 4.3 and 4.5. For easy reasoning though, the rules for type and constraint elaboration are now presentated seperately from those for the well-formedness relation, as shown in Figure 6.3.

$\boxed{\vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon}$ Type Elaboration

$$\frac{}{\vdash_{\mathtt{ty}} a \rightsquigarrow a}\ \textsc{TyVar} \qquad \frac{\vdash_{\mathtt{ty}} \tau_1 \rightsquigarrow \upsilon_1 \qquad \vdash_{\mathtt{ty}} \tau_2 \rightsquigarrow \upsilon_2}{\vdash_{\mathtt{ty}} \tau_1 \rightarrow \tau_2 \rightsquigarrow \upsilon_1 \rightarrow \upsilon_2}\ \textsc{TyArr}$$

$$\frac{\vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon_1 \qquad \vdash_{\mathtt{ty}} \rho \rightsquigarrow \upsilon_2}{\vdash_{\mathtt{ty}} C \Rightarrow \rho \rightsquigarrow \upsilon_1 \rightarrow \upsilon_2}\ \textsc{TyQual} \qquad \frac{\vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon}{\vdash_{\mathtt{ty}} \forall a.\sigma \rightsquigarrow \forall a.\upsilon}\ \textsc{TyAll}$$

$\boxed{\vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon}$ Constraint Elaboration

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

$$\frac{\vdash_{\mathtt{ct}} C_1 \rightsquigarrow \upsilon_1 \qquad \vdash_{\mathtt{ct}} C_2 \rightsquigarrow \upsilon_2}{\vdash_{\mathtt{ct}} C_1 \Rightarrow C_2 \rightsquigarrow \upsilon_1 \rightarrow \upsilon_2}\ (\mathrm{C}{\Rightarrow})$$

Figure 6.3: Type and constraint elaboration

Chapter 7

Constraint Entailment

The main contribution of this thesis, is the extension of Haskell with the new, much more flexible and powerful quantified class constraints. Of course, this bump in expressive power comes with a more complex specification of constraint entailment, which is the focus of this chapter. As we illustrate in the following sections, the traditional goal-directed constraint entailment used in prior work on type classes (backwards-chaining) is not applicable in a system with quantified class constraints; the overlapping nature of the available axioms results in ambiguities, significantly big search spaces and backtracking, practically making constraint solving intractable. For simplicity of the presentation, we first present a simple specification which is concise and easy to understand (Section 7.1), yet exhibits the aforementioned disadvantages. Later on (Section 7.2), we illustrate how to address the ambiguity issue, by adopting a more complex proof search method, known as focusing [22]. This technique has already been adopted by Cochis [25] for the same reason. Furthermore, the constraint entailment as used by this thesis, closely resembles that of the Implicit Calculus [19].

7.1 Ambiguous Specification

The simple, yet ambiguous constraint entailment relation is presented in Figure 7.1. Rule SpecC is the base case, where the wanted constraint is literally a part of the program theory. Rules ($\forall$IC) and ($\forall$EC) introduce and eliminate type abstraction in the wanted constraint respectively. Introducing a type variable a to the constraint, means that the rest of the constraint C had to be checked under the extended environment. Elimination of a type variable happens by instantiating it with a well-formed type τ. Rules ($\Rightarrow$IC) and ($\Rightarrow$EC) introduce and eliminate constraint implication respectively. Introducing an implication, means the new constraint C_1 becomes available in the program theory, while checking C_2. Eliminating an implication is only possible when the required constraint C_1 is itself entailed under the given program theory and environment.

The main advantage of this approach is its compactness and easy understandability. Unfortunately, it is also highly ambiguous. Consider for instance the task of proving $Show\ a$ under $\Gamma = \bullet, a$ and $P = \langle \bullet; Eq\ a, Show\ a; \bullet \rangle$. It is easy to see that infinately many possible proofs exist. We provide two of them:

$$\dfrac{\mathit{Show}\ a \in \langle \bullet ; \mathit{Eq}\ a, \mathit{Show}\ a; \bullet \rangle}{\langle \bullet ; \mathit{Eq}\ a, \mathit{Show}\ a; \bullet \rangle ; \Gamma \models \mathit{Show}\ a}\ (\textsc{SpecC})$$

$$\dfrac{\dfrac{\dfrac{\mathit{Show}\ a \in \langle \bullet ; \mathit{Eq}\ a, \mathit{Show}\ a; \mathit{Eq}\ a \rangle}{\langle \bullet ; \mathit{Eq}\ a, \mathit{Show}\ a; \mathit{Eq}\ a \rangle ; \Gamma \models \mathit{Show}\ a}\ (\textsc{SpecC})}{\langle \bullet ; \mathit{Eq}\ a, \mathit{Show}\ a; \bullet \rangle ; \Gamma \models \mathit{Eq}\ a \Rightarrow \mathit{Show}\ a}\ (\Rightarrow\mathrm{IC}) \quad \dfrac{\mathit{Eq}\ a \in \langle \bullet ; \mathit{Eq}\ a, \mathit{Show}\ a; \bullet \rangle}{\langle \bullet ; \mathit{Eq}\ a, \mathit{Show}\ a; \bullet \rangle ; \Gamma \models \mathit{Eq}\ a}\ (\textsc{SpecC})}{\langle \bullet ; \mathit{Eq}\ a, \mathit{Show}\ a; \bullet \rangle ; \Gamma \models \mathit{Show}\ a}\ (\Rightarrow\mathrm{EC})$$

Both of these derivations are correct, although the second one is significantly longer than required. This ambiguity in the constraint entailment is extremely undesirable. It causes constraint entailment to become non-determinstic, in that the choices made can determine whether a constraint is entailed, or even whether the entailment takes an infinite number of steps. Furthermore, the ambiguity could even result in multiple distinct possibilities for the resulting computational content contained in the dictionaries.

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

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

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

Figure 7.1: Declarative constraint entailment specification

7.2 Focusing

The solution we adopt to reduce the ambiguity of constraint entailment is a technique called focusing [22], previously used by the Cochis calculus [25]. Notionally, focusing combines top down and bottom up proof search, to dramatically reduce non-determinism. The new constraint entailment relation is presented in Figure 7.2. The first part of the figure is the main entailment relation, which is simply equivalent to the constraint resolution relation. This, in turn, will iteratively simplify the wanted constraint C until we eventually reach a class constraint Q. Rules ($\Rightarrow$R) and ($\forall$R) iteratively strip down the wanted constraint, by recursively checking the remaining constraint under the extended program theory and environment, respectively. Rule (QR) then has to select a constraint from the program theory to match against the wanted class constraint. When the constraints match, the wanted constraint Q

is simplified into an axiom set A, and all constraints in contains have to be entailed as well, in order for the original constraint to be satisfied.

Matching is performed by the constraint matching relation, which focusses on the axiom C and checks whether it matches the wanted constraint Q. Rule $(\Rightarrow\mathrm{L})$ handles the implication case, by recursively checking whether C_2 matches the wanted constraint, and returning C_1 as a remainder. As we mentioned above, we need to accumulate the remaining constraints, such that Rule $(Q\mathrm{R})$ can ensure their satisfiability. Rule $(\forall\mathrm{L})$ instantiates type variable a in C with a (well-formed) type τ. Lastly, the two constraints match, if the resulting class constraint from recursively applying the last two rules, is an exact match with the wanted class constraint Q, which is expressed in Rule $(Q\mathrm{L})$.

Observe that this method of constraint entailment combines top-down reasoning, in the constraint resolution relation, with bottom-up reasoning, by focusing on a single constraint C in the constraint matching relation. This results in a sort of meet-in-the-middle-approach. As opposed to the constraint entailment relation we presented in the previous section, both parts of this relation are syntax directed! The constraint resolution rules are syntax-directed on the shape of the wanted constraint, whereas the constraint matching relation is syntax-directed on the shape of the chosen axiom. Because of this, the search space is dramatically reduced, thus greatly reducing the number of possible proofs for a given wanted constraint.

Consider again the `Show a` example, from Section 7.1. Using the focusing specification, it only has one valid derivation: simply picking the `Show a` axiom from the theory, by the $(Q\mathrm{R})$ rule and immediately matching, using the $(Q\mathrm{L})$ rule.

7.2.1 Remaining Nondeterminism

The reader may notice however, that even the focusing approach, as described above, is still ambiguous. This ambiguity originates from two main sources:

Overlapping Axioms The first source of ambiguity originates from Rule $(Q\mathrm{R})$. Multiple axioms from the program theory might match the wanted constraint. Since solving a constraint using a different axiom might result in a different dictionary, and thus in a different System F term, the computational behavior could possibly be non-deterministic. To avoid this, we follow the solution used by Haskell '98, that is, requiring from the programmer that instance declarations never overlap.

Because of superclasses, multiple distinct proofs might still arise. However, since all computational context originates from instance declarations, and overlapping instances are not allowed, this can never result in distinct computational behaviour.

Polymorphic Instantiation The second source of ambiguity originates from Rule $(\forall\mathrm{L})$. It “guesses” a well-formed type τ for instantiating the type variable b. This is an issue since there are infinitely many possible well-typed types to choose from, which may eventually result in both different entailment behaviour and different distinct computational behaviour. Choosing a "wrong" type to instantiate with, could result in failing to entail a constraint later on. This makes the entailment non-deterministic. Furthermore, even if multiple choices for a type allow the constraint entailment to succeed, they may result in a different computational content. Consider for instance the following example:

```
instance Eq Nat  where ...
instance Eq Bool where ...
instance Eq a    => Eq Char where ...
```

When resolving `Eq Char`, we may choose `a` to be either `Nat` or `Bool`, which could result in a different dictionary, containing different method implementations.

Haskell '98 solves this issue by requiring that all declared variables occur in the head of the axiom. However, since our constraints can be more flexible and allow nested constraints, this requirement has to generalized as well. This is done by the $\mathit{unamb}(C)$ relation, as shown in Figure 7.3. We thus require every axiom, added to the program theory to be unambiguous.

Rule ($\forall$U) collects all declared type variables. Rule (QU) checks that all these variables are determined by the head of the axiom. Finally, Rule ($\Rightarrow$U) checks that all type variables $\overline{a}$ are indeed determined by C_2, but also that C_1 is unambiguous on itself. This is required since Rule ($\Rightarrow$R) extends the program theory with these constraints, so they have to be well behaved by themselves.

This unambiguity requirement is imposed on all constraints that are appended to the program theory.

7.3 Constraint Solving

Typing terms generates two sorts of constraints: class constraints and equality constraints. The algortihm for solving the latter remains completely identical to that for regular type classes (or any HM-based system) as explained in Section 3.3. Our algorithm for solving quantified class constraints in provided in Figure 7.4. It is based on the specification from Figure 7.2 and thus on the focusing approach. The two main differences from the specification are: a) The constraint entailment algorithm now also constructs dictionaries, which act as proofs for the wanted constraints. This is done through the use of dictionary substitutions, which allow to express the dictionary for the wanted constraint in terms of dictionaries for simpler constraints, as well as axioms available from the program theory. This is explained in detail in Chapter 8. b) Just as discussed in Section 4.3, the algorithmic version also allows to partially entail constraints. This way inferred types for top-level term declarations can be simplified.

The constraint solving algorithm consists of three parts. The first relation, the constraint solving relation $\overline{a}; \mathcal{P} \models \mathcal{A}_1 \rightsquigarrow \mathcal{A}_2; \eta$, recursively simplifies the constraints in the given axiom set A_1, until no more constraints exist that can be simplified. The remaining axiom set is returned, together with the accumulated dictionary substitution.

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

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

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

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

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

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

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

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

FIGURE 7.2: Constraint entailment specification using focusing

$\boxed{\mathit{unamb}(C)}$ Unambiguity

$$\frac{\bullet \vdash_{\mathtt{unamb}} C}{\mathit{unamb}(C)} \;\textsc{Unamb}$$

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

$$\frac{\overline{a} \subseteq \mathit{fv}(Q)}{\overline{a} \vdash_{\mathtt{unamb}} Q} \;(Q\mathrm{U}) \qquad \frac{\overline{a}, a \vdash_{\mathtt{unamb}} C}{\overline{a} \vdash_{\mathtt{unamb}} \forall a. C} \;(\forall\mathrm{U}) \qquad \frac{\begin{array}{c}\mathit{unamb}(C_1) \\ \overline{a} \vdash_{\mathtt{unamb}} C_2\end{array}}{\overline{a} \vdash_{\mathtt{unamb}} C_1 \Rightarrow C_2} \;(\Rightarrow\mathrm{U})$$

FIGURE 7.3: Unambiguity

Simplification happens by means of the constraint simplification relation $\overline{a}; \mathcal{P} \models [\mathcal{C}] \rightsquigarrow \mathcal{A}\,;\eta$. It is syntax directed on the wanted constraint $\mathcal{C}$ and iteratively reduces this constraint to a simple class constraint Q. Rule $(\Rightarrow\text{R})$ recursively simplifies C_2, under the extended program theory, to a list of constraints $\overline{C}$. The simplified constraints are then again extended with the C_1 constraint. Rule $(\forall\text{R})$ simplifies $\forall b.C_0$, into $\overline{\forall b.C}$, by simplifying C_0 into $\overline{C}$. Type variable b' in the simplified constraints from Rule $(\forall\text{R})$ is a fresh type variable. Lastly, Rule $(Q\text{R})$ works identically to the corresponding rule in the specification. It selects an axiom from the program theory to focus on, and tries to match the wanted class constraint Q.

The latter happens through the constraint matching relation $\overline{a}; [\mathcal{C}] \models \mathcal{Q} \rightsquigarrow \mathcal{A}; \theta\,;\eta$. It step-by-step deconstructs the focused axiom $\mathcal{C}$ to try and match the wanted class constraint $\mathcal{Q}$. As opposed to the specification, it is both completely syntax directed and completely deterministic in that it never has to make any "choices". Rule $(\Rightarrow\text{L})$ recursively tries to match the focused constraint with C_2, and then adds the C_1 constraint to the remaining constraints to solve. Rule $(\forall\text{L})$ rule recursively matches C to Q. As opposed to the specification, Rule $(Q\text{L})$ does not guess the types to instantiate the free variables in the focused constraint with. Instead, it matches two class constraints by unifying their contained types. The resulting substitution should instantiate all free variables of τ_1, with the matching types from τ_2. This is the case because of the fact that all constraints in the program theory are required to be unambiguous, as discussed in Section 7.2.1. The free variables of τ_2 should not be altered. Because the free variables of τ_2 are in the untouchables $\overline{a}$, we know that $\theta(\tau_1) = \tau_2$.

7.3.1 Search

As explained in Section 7.2.1, multiple matching axioms might exist in the program theory, due to overlapping superclass axioms. Since these axioms in the extended language are not necessarily simple axioms, matching against them may result in residual goals that require recursive resolution. Because this recursive resolution might fail, we have to backtrack over the choice of axiom in the $(Q\text{R})$ rule. Consider for instance the following example:

```
class (E a => C a) => D a where ...
class (G a => C a) => F a where ...
```

Given the axioms `D a`, `F a` and `G a`, and the goal `C a`. The class declarations result in the following superclass axioms:

```
D a => (E a => C a)
F a => (G a => C a)
```

Both of these match the goal. If we pick the first one, resolution gets stuck when recursively solving `E a`. However, if we backtrack and choose the second one instead, we can recursively resolve `G a` against the given axiom.

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

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

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

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

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

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

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

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

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

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

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

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

FIGURE 7.4: Constraint entailment algorithm with dictionary construction

Chapter 8

Dictionary Construction

Similarly to the language with regular type classes, as explained in Chapter 4, we elaborate the extended language into System F, using a dictionary-passing translation method. Just like the constraint solving algorithm for regular type classes, the entailment algorithm described in Section 7 constructs explicit proofs in the form of dictionary substitutions, while entailing a constraint [8].

8.1 Constraint Entailment

The constraint entailment algorithm is presented in Figure 7.4 and explained in Section 7.3. The constraint solving relation doesn't create new dictionary substitutions, but simply combines those created by the constraint simplification relation.

Constraint Simplification The generated dictionary substitution η can construct a proof for the wanted constraint C in terms of the simpler constraints A and the program theory P. For instance, the ($\Rightarrow$R) rule creates a dictionary substitution η', which can construct a proof for $(d_0 : C_1 \Rightarrow C_2)$, using the proofs $\overline{d'}$ for the simplified constraints $\overline{(C_1 \Rightarrow C)}$ and the program theory P. We look at the constructed dictionary substitution, step-by-step:

- The recursive call returns a dictionary substitution η, which can construct a proof for $(d_2 : C_2)$ in term of the proofs for the local simplified constraints $\overline{(d : C)}$, the local constraint $(d_1 : C_1)$ and the program theory P.
- $\eta(d_2)$ is thus a dictionary for C_2 which can refer to dictionary variables bound by the program theory P, assuming a proof is available for d_1 and $\overline{d}$.
- $\overline{[d'\ d_1/d]}$ constructs dictionary placeholders for $\overline{d}$, which can refer to the dictionary variables for the final residual constraints $\overline{(d' : C_1 \Rightarrow C)}$ and the locally assumed proof for $(d_1 : v_1)$.
- $\overline{[d'\ d_1/d]}(\eta(d_2))$ thus replaces the dependancy on the $\overline{d}$ assumption, using $\overline{d'}$ and d_1. It is thus a dictionary for C_2, expressed in terms of dictionary variables bound by the program theory P, assuming proofs for d_1 and $\overline{d'}$ are available.

- The final step explicitly abstracts over d_1. $\lambda(d_1 : \upsilon_1).[d'\ d_1/d](\eta(d_2))$ is thus a proof for d_0, using the program theory P, the dictionary for d_1 and assuming a proof is available for the residual constraints $\overline{(d' : C_1 \Rightarrow C)}$.

Rule ($\forall$R) works similarly. It creates a dictionary substitution η' which can construct a proof for $(d_0 : \forall b. C_0)$, using the program theory P and proofs $\overline{d'}$ for the residual constraints. We again look at the constructed substitution, step by step:

- The recursive call results in a dictionary substitution η, which constructs a proof for $(d_C : C_0)$, in terms of the program theory P and the proofs $\overline{d}$ for the local residual constraints $\overline{C}$.
- $\eta(d_C)$ is thus a proof for C_0, relying only on the program theory P and the assumption that proofs exist for $\overline{(d : C)}$.
- $\overline{[d'\ b/d]}$ constructs dictionary placeholders for $\overline{d}$, in terms of the dictionary variables $\overline{d'}$, which correspond to the final residual constraints $\overline{\forall b. C}$.
- $\overline{[d'\ b/d]\eta(d_C)}$ thus replaces the dependency on the $\overline{d}$ assumption with a dependency on the assumption that proofs are available for $\overline{(d' : \forall b. C)}$.
- Finally, we explicitly abstract over the type variables b, and apply the proofs for $\overline{d'}$ to b. $\Lambda b.\overline{[d'\ b/d]\eta(d_C)}$ is thus a proof for d_0, relying only on the program theory P and the assumption that proofs are available for the residual constraints $\overline{(d' : \forall b. C)}$.

Rule (QR) simply returns the dictionary substitution, generated by the constraint matching relation.

Constraint Matching The constraint matching relation works similarly to the constraint simplification relation, in that it too constructs a dictionary substitution as a way of making the proof of entailment explicit in System F. The created substitution η constructs a proof for Q out of the axiom C and the simpler constraints A.

Rule ($\Rightarrow$L) gets a dictionary substitution η from the recursive call, which shows how to construct a proof for Q from the $(d_2 : C_2)$ axiom and proofs for the simpler constraints A. The rule then creates a new dictionary substitution, which maps the d_2 axiom from η on the $(d : C_1 \Rightarrow C_2)$ axiom, thus creating a proof for Q, using the $(d : C_1 \Rightarrow C_2)$ axiom and the new residual constraints. In order for this to work, a proof d_1 has to exist for $\theta(C_1)$, which is added to the residual constraints, as an extra wanted constraint.

Rule ($\forall$L) works similarly. The recursive call results in the dictionary substitution η, which constructs a proof for Q, using the $(d' : C)$ axiom and the residual constraints A. The rule constructs a new dictionary substitution by mapping the d' axiom onto $(d : \forall b. C)$ and instantiating the b type variable with the corresponding type from Q, as explained next.

Finally, Rule (QL) is straightforward. Since we know $TC\ \tau_1$ and $TC\ \tau_2$ are unifiable, and we know that $\theta(TC\ \tau_1) = TC\ \tau_2$ (explained in Section 3.3), we can simply return the d' axiom as a proof for the wanted constraint $(d : TC\ \tau_2)$. Rule ($\forall$L) afterwards recursively applies the corresponding type $\theta(b)$ from τ_2 for each of the free variables b in τ_1.

Chapter 9

Meta-Theory

This chapter discusses several desirable meta-theortical properties of the extended language. This thesis presented two specifications of the language: a) a concise and easily understandable specification (see Section 7.1), and b) a tractable, less ambiguous specification (see Section 7.2), based on focusing. Furthermore, an algorithmic version of type inference and constraint entailment with elaboration to System F have been provided in Section 7.3 and Chapter 8. In the remainder of this chapter, we focus on the three properties presented below. Corresponding proofs for some of these theorems can be found in Appendix A.

Termination Termination of the type inference algorithm is one of the main desirable properties. It means that infering a type for a program either fails or returns a corresponding type. The process may never diverge.

Type Preservation Preservation of typing under the elaboration to System F means that when a term e is well-typed under Γ, its elaboration t is well-typed as well under the elaborated environment Δ. Furthermore, its type is the elaboration of the original type. The elaboration mapping thus preserves the typing relation.

Equivalence A third and final interesting property we investigate is the equivalence between both specifications and the algorithm. Equivalence between A and B means that A is both sound (all elements from A are in B as well) and complete (all elements from B are also in A) with respect to B. Because equivalence is transitive, we do not have to prove each equivalence to state that both specifications and the algorithm are all equivalent.

Approach Due to the fact that the focusing-based specification and the algorithmic entailment differ in their structure (the former instantiates type variables top-down, while the latter does so bottom-up), we introduce an alternative, intermediate algorithm for constraint entailment. We use this algorithm to bridge the gap between the original algorithm and the specification. The type preservation is proven for the alternative algorithm as well. The discussed properties thus become:

1. Termination of type inference.
2. Soundness of the new algorithm with respect to the focusing specification.
3. Completeness of the new algorithm with respect to the focusing specification.
4. Equivalence between the normal and intermediate type inference algorithm.
5. Preservation of types for the translation of terms, as performed in the new algorithmic rule set.
6. Equivalence between the ambiguous specification and the focusing-based specification.
7. The soundness and completeness of the normal algorithm with respect to the focusing specification follows trivially from (3), (4) and (5).

This is shown graphically in Figure 9.1. Alg_A represents the old version of the algorithm. Alg_B is the alternative, intermediate algorithm, used exclusively in this chapter. Spec_A represents the ambiguous specification, whereas Spec_F represents the more deterministic, focusing-based specification.

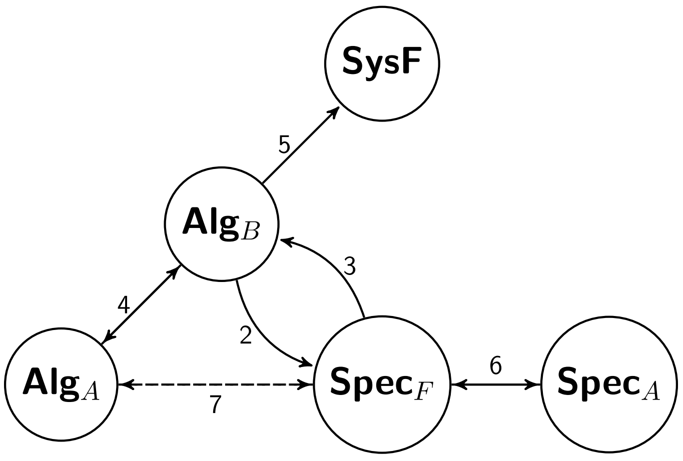


Figure 9.1: Meta theory topics graph

9.1 Alternative Algorithm

Figure 9.2 shows the alternative version of the constraint entailment algorithm, used exclusively in this chapter. The constraint solving relation remains identical, as does the constraint simplification relation. The main difference is located in the constraint matching relation. Instead of building up the residual constraints and dictionary substitution after unifying, the new relation utilizes accumulators. The residual constraints and dictionary substitution is thus accumulated step by step, and passed into the recursive call. Finally, the $(Q\text{L})$ rule unifies and applies the θ

substitution to the accumulators. Because of this, the θ substitution no longer has to be propagated upwards.

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

$$\frac{\begin{array}{cc} \vdash_{\mathtt{ct}} C_1 \rightsquigarrow \upsilon_1 & \overline{a}; \mathcal{A}, (d_1 : C_1) \models [d_2 : C_2] \rightsquigarrow \overline{(d : C)} ; \eta \\ \overline{d'}, d_1, d_2 \text{ fresh} & \eta' = [\lambda(d_1 : \upsilon_1).[\overline{d'\ d_1/d}](\eta(d_2))/d_0] \end{array}}{\overline{a}; \mathcal{A} \models [d_0 : C_1 \Rightarrow C_2] \rightsquigarrow \overline{(d' : C_1 \Rightarrow C)} ; \eta'} \; (\Rightarrow\text{R})$$

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

$$\frac{\mathcal{C} \in \mathcal{A} : \quad \overline{a}; [C] \models \mathcal{Q} \mid (\bullet ; \bullet) \rightsquigarrow (\mathcal{A}' ; \eta)}{\overline{a}; \mathcal{A} \models [\mathcal{Q}] \rightsquigarrow \mathcal{A}' ; \eta} \; (\mathcal{Q}\text{R})$$

$\boxed{\overline{a}; [\mathcal{C}] \models \mathcal{Q} \mid (\mathcal{A} ; \eta) \rightsquigarrow (\mathcal{A}' ; \eta')}$ Constraint Matching

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

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

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

Figure 9.2: Alternative constraint entailment algorithm

9.2 Auxiliary Definitions

Before starting the technical parts of this chapter, we define some small extra auxiliary concepts.

Head The head of an axiom C is defined to be the class constraint, which remains after stripping of all implications and type abstractions. It is the constraint that is matched against when matching two axioms. The definition is as follows:

$$\begin{aligned} \mathit{head}(C_1 \Rightarrow C_2) &= \mathit{head}(C_2) \\ \mathit{head}(\forall a. C) &= \mathit{head}(C) \\ \mathit{head}(Q) &= Q \end{aligned} \tag{9.1}$$

Occurances The number of occurances of a type variable in a class constraint, is defined as the number of occurances of this variable in its contained type, as follows:

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

9.3 Termination

Termination of the type inference algorithm, means that the process of infering a type for a given program always terminates. It may still fail if the program is not well-typed, but it may never diverge. The only difference between our type inference algorithm and the one for standard type classes, is our constraint entailment. Since termination is common knowledge for regular type classes, our termination is uniquely dependant on the termination of our constraint solving algorithm. In order to achieve this, we enforce several conditions on our axioms. Our language conditions are derived from those used in Cochis and generalize those used in Haskell '98.

In order to reason about termination, we represent the entailment process as a tree. The nodes represent wanted constraints. Each edge represents the matching of the parents goal node with an axiom, resulting in the residual goals, which are the child nodes. Showing termination is thus equivalent to showing that the tree has a finite length, or that no infinite paths exist in the tree.

We define the norm $\|\bullet\|$ to be the length of the head of a constraint C as follows:

$$\begin{aligned} \|a\| &= 1 \\ \|\tau_1 \to \tau_2\| &= 1 + \|\tau_1\| + \|\tau_2\| \end{aligned} \tag{9.2}$$

We now state that the this norm strictly decreases at every edge, or at least non-stricty decreases, but with the guarantee that the size always decreases after a finite number of steps. Because of this guarantee, it is easy to see that no infinite path can exist in the entailment tree, since the norm can only take natural number values.

Strictly Decreasing The easiest way to go, would be to require that for each axiom, the norm is strictly decreasing. An initial version of this requirement would look as follows:

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

$$\frac{\begin{array}{ccc} term(C_1) & term(C_2) & \\ Q_1 = head(C_1) & Q_2 = head(C_2) & \|Q_1\| < \|Q_2\| \end{array}}{term(C_1 \Rightarrow C_2)}\ (\Rightarrow\text{T})$$

Figure 9.3: Axiom termination condition

The only rule in the constraint entailment algorithm which actually extends the residual constraint set, is the ⇒L rule. The residual constraints thus originate from the left hand side of a constraint implication. Since the head of these constraints is never altered in any of the other rules, we only have to require one extra condition on the constraints. Namely that for constraint implications $C_1 \Rightarrow C_2$ it holds that $\|head(C_1)\| \leq \|head(C_2)\|$.

Unfortunately, this rule is unstable under type substitution. Consider for example the following axiom:

$$\forall a.C(a \rightarrow a) \Rightarrow C(a \rightarrow Int \rightarrow Int)$$

The context constraint size is 3 and thus strictly smaller than size 5 of the head. However, if we apply the following substitution: $\theta = \bullet, a \mapsto (Int \rightarrow Int \rightarrow Int)$, the requirement breaks, since the context constraint size is now 11, which is bigger than the norm of 10 for the head. A possible solution for this is to add an extra stability requirement to the ⇒T rule premesis:

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

This requirement however can not be applied in an algorithm, since an infinite number of possible substitutions θ exist. An equivalent algorithmic version of this requirement exists, which reasons about the free type variables in the given constraint, instead of over all possible type substitutions. The result is shown in Figure 9.4. The equivalence between the two stability requirements is straightforward to see.

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

$$\frac{\begin{array}{cccc} term(C_1) & term(C_2) & Q_1 = head(C_1) & Q_2 = head(C_2) \\ \|Q_1\| < \|Q_2\| & \multicolumn{3}{c}{\forall a \in fv(C_1) \cup fv(C_2) : \quad occ_a(Q_1) \leq occ_a(Q_2)} \end{array}}{term(C_1 \Rightarrow C_2)}\ (\Rightarrow T)$$

FIGURE 9.4: Axiom termination condition with stability

Decreasing Unfortunately, when we take superclasses into account as well, the requirement that the norm of every axiom has to be strictly decreasing becomes too strong. Consider for instance the following superclass axiom:

$$\forall a.Ord\ a \Rightarrow Eq\ a$$

Both sides have a norm of 1, so the strictly decreasing requirement is not satisfied.

To be able to allow similar superclass axioms, we weaken the condition to requiring that each axiom has to be (non strictly) decreasing. However, to compensate for this, we add the extra condition that the superclass relation is directed acyclic graph (DAC). There can be no loops in the superclass relation. Note that the DAG corresponds to a partial ordering on the type classes. Because of this, no infinite path can exist in the entailment tree. A sequence of applying superclass axioms may exists, where the norms of each step either remain constant or decrease. Because of the condition that a partial ordering on the type classes exists, we know that the length of this sequence is bounded by the height of the superclasses DAG. After this sequence the entailment tree either ends in a leaf, or continues by applying a local or instance axiom. We know from the previous paragraph that the norm strictly decreases. The constraint entailment process thus has to be finite.

9.4 Equivalence

The equivalence between our intermediate algorithm and the focusing-based specification is handled in two seperate parts: soundness and completeness. The only difference between our type inference algorithm and the standard type classes one, is our constraint entailment algorithm. Since the equivalence between standard type classes and their specification has already been well established, we focus purely on the soundness and completeness between our intermediate constraint entailment algorithm and the corresponding focusing-based specification.

9.4.1 Soundness (2)

The soundness theorems are provided as theorems 5, 6 and 7, for the left, right and recursive entailment relation respectively. They are proven by structural induction on the constraints, as shown in the appendix.

Theorem 5 (Sound Left) *If* $\overline{a}; [d : C] \models (d_Q : Q) \mid (A_1 ; \eta) \rightsquigarrow (A'_1, A_2 ; \eta'); \theta$ *and* $\forall \Gamma$ *such that* $\Gamma \vdash_{\mathsf{ct}} C$ *and* $\Gamma \vdash_{\mathsf{ct}} Q$ *then* $\Gamma; [\theta(C)] \models \theta(Q) \rightsquigarrow \theta(A_2)$ *and* $\Gamma \vdash_{\mathsf{ax}} A_2$.

Theorem 6 (Sound Right) *If* $\overline{a}; A \models [d : C] \rightsquigarrow A' ; \eta ; \theta$ *and* $\forall \Gamma$ *such that* $\Gamma \vdash_{\mathsf{ct}} C$ *and* $\Gamma \vdash_{\mathsf{ax}} A$ *and* $fv(C) \subseteq \overline{a}$ *then* $\theta(A); \Gamma \models [\theta(C)] \Leftrightarrow (\forall (d_i : C_i) \in A' : \theta(A); \Gamma \models [\theta(C_i)])$ *and* $\Gamma \vdash_{\mathsf{ax}} A'$.

Theorem 7 (Sound Recursive) *If* $\overline{a}; A \models A_1 \rightsquigarrow A_2; \eta ; \theta$ *and* $\forall \Gamma$ *such that* $\Gamma \vdash_{\mathsf{ax}} A$ *and* $\Gamma \vdash_{\mathsf{ax}} A_1$ *and* $fv(A_1) \subseteq \overline{a}$ *then* $(\forall (d_i : C_i) \in A_1 : \theta(A); \Gamma \models \theta(C_i)) \Leftrightarrow (\forall (d_i : C_i) \in A_2 : \theta(A); \Gamma \models \theta(C_i))$ *and* $\Gamma \vdash_{\mathsf{ax}} A_2$.

9.4.2 Completeness (3)

Unfortunately, due to time constraints, completeness has not been inverstigated for this thesis. This remains interesting future work.

9.4.3 Algorithm Equivalence (4)

The equivalence between the normal algorithm and the intermediate algorithm is stated in three seperate theorems (theorems 10, 9 and 8), corresponding with the left, right and recursive constraint entailment relations respectively. The normal algorithm is marked with an A, whereas the intermediate one is marked with a B. The left rules are the only difference between the two algorithms. Their equivalence has been proven in the appendix. Since the right and recursive rules are identical, their proofs are entirely straighforward.

Theorem 8 (Equivalence Recursive) $\overline{a}; A \models_A A_1 \rightsquigarrow A_2; \eta; \theta$
$\Leftrightarrow \overline{a}; A \models_B A_1 \rightsquigarrow A_2; \eta; \theta$

Theorem 9 (Equivalence Right) $\overline{a}; A \models_A [d : C] \rightsquigarrow A'; \eta; \theta$
$\Leftrightarrow \overline{a}; A \models_B [d : C] \rightsquigarrow A'; \eta; \theta$

Theorem 10 (Equivalence Left) $\overline{a}; [d : C] \models_A (d_Q : Q) \rightsquigarrow A_2; \theta; \eta_2$
$\Leftrightarrow \overline{a}; [d : C] \models_B (d_Q : Q) \mid (A_1; \eta_1) \rightsquigarrow (\theta(A_1), A_2; \theta(\eta_1) \cdot \eta_2); \theta$

9.4.4 Specification Equivalence (6)

The equivalence between the two specifications is stated in theorem 11. The ambiguous relation is marked with an A, whereas the focusing-based relation is annotated with an F.

Theorem 11 (Specification Equivalence) $P; \Gamma \models_A C \Leftrightarrow P; \Gamma \models_F C$

9.5 Type Preserving Translation (5)

Type preservation for types and constraints is represented in theorems 12, 13 and 14. The type inference algorithm is almost identical to the one for standard type classes, for which type preservation has already been extensively studied and proven. The only difference is our constraint entailment algorithm. This is thus the only aspect of type preservation of interest in this thesis. Type preservation for the constraint entailment is formulated in theorems 15, 16 and 17.

Theorem 12 (Type Preservation for Monotypes)
If $\Gamma \vdash_{\mathtt{ty}} \tau$ *and* $\vdash_{\mathtt{ty}} \tau \rightsquigarrow \upsilon$ *and* $\Gamma \rightsquigarrow \Delta$ *then* $\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$.

Theorem 13 (Type Preservation for Constraints)
If $\Gamma \vdash_{\mathtt{ct}} C$ *and* $\vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon$ *and* $\Gamma \rightsquigarrow \Delta$ *then* $\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$.

Theorem 14 (Type Preservation for Polytypes)
If $\Gamma \vdash_{\mathtt{ty}} \sigma$ *and* $\vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon$ *and* $\Gamma \rightsquigarrow \Delta$ *then* $\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$*.*

Theorem 15 (Type Preservation for Left Entailment)
If $\overline{a}; [d' : C] \models (d : Q) \mid (A_1 ; \eta_1) \rightsquigarrow (A'_1, A_2 ; \eta'_1 \cdot \eta_2); \theta$
and $\forall \Gamma$ *such that* $\Gamma \vdash_{\mathtt{ct}} C$ *and* $\Gamma \vdash_{\mathtt{ct}} Q$
and $\vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon'$ *and* $\vdash_{\mathtt{ct}} Q \rightsquigarrow \upsilon$ *and* $unamb(C)$
and $\Gamma \rightsquigarrow \Delta$ *and* $A_2 \rightsquigarrow \Delta_A$ *and* $fv(Q) \subseteq \overline{a}$
then $\theta(\Delta), \theta(\Delta_A), (d' : \theta(\upsilon')) \vdash^{\mathtt{F}}_{\mathtt{tm}} \theta(\eta_2(d)) : \theta(\upsilon)$ *and* $\Gamma \vdash_{\mathtt{ax}} A_2$

Theorem 16 (Type Preservation for Right Entailment)
If $\overline{a}; A \models [d : C] \rightsquigarrow A' ; \eta ; \theta$
and $\forall \Gamma$ *such that* $\Gamma \vdash_{\mathtt{ct}} C$ *and* $\Gamma \vdash_{\mathtt{ax}} A$
and $\vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon$ *and* $\Gamma \rightsquigarrow \Delta$, $A \rightsquigarrow \Delta_A$ *and* $A' \rightsquigarrow \Delta_{A'}$
and $fv(head(C)) \subseteq \overline{a}$ *and* $unamb(C)$
then $\theta(\Delta), \theta(\Delta_A), \theta(\Delta_{A'}) \vdash^{\mathtt{F}}_{\mathtt{tm}} \theta(\eta(d)) : \theta(\upsilon)$ *and* $\Gamma \vdash_{\mathtt{ax}} A'$

Theorem 17 (Type Preservation for Recursive Entailment)
If $\overline{a}; A \models A_1 \rightsquigarrow A_2; \eta ; \theta$
and $\forall \Gamma$ *such that* $\Gamma \vdash_{\mathtt{ax}} A$ *and* $\Gamma \vdash_{\mathtt{ax}} A_1$
and $\Gamma \rightsquigarrow \Delta$, $A \rightsquigarrow \Delta_A$, $A_1 \rightsquigarrow \Delta_{A_1}$ *and* $A_2 \rightsquigarrow \Delta_{A_2}$
and $\forall (d_i : C_i) \in A_1 : \vdash_{\mathtt{ct}} C_i \rightsquigarrow \upsilon_i$
then $\forall (d_i : C_i) \in A_1 : \theta(\Delta), \theta(\Delta_A), \theta(\Delta_{A_2}) \vdash^{\mathtt{F}}_{\mathtt{tm}} \theta(\eta(d_i)) : \theta(\upsilon_i)$ *and* $\Gamma \vdash_{\mathtt{ax}} A_2$

Unfortunately, due to time constraints, we have not proven type preservation for the constraint entailment. However, the examples we have tested with the prototype compiler provide confidence that the elaboration is type preserving. We leave the formal proof as future work.

Chapter 10

Related Work

In this chapter we discuss related work, focusing mostly on two aspects: (a) We compare the language extension with several different investigated workarounds or alternative encodings in Haskell, and, (b) we compare our approach of introducing quantified class constraints in Haskell, to several existing, similar languages which feature quantified constraints.

10.1 Alternative Encodings

Several investigated workarounds for achieving (a fragment of) the added expressive power of quantified class constraints exist, without extending Haskell with a new feature.

Trifonov In her Simulating Quantified Class Constraints paper [28], Trifonov presents two separate encodings for representing quantified class constraints using regular type classes. The first encoding relies only on standard Haskell '98. It operates by taking out the quantified constraints, and placing them in a seperate new class, with new names for both the class and methods. Consider for instance the monad transformer class from Section 5.1. The quantified class constraint `forall m. Monad m => Monad (t m)` should be taken out and placed into a new `Monad_t` class. The transformer class is now represented as follows:

```
class Monad_t t where
    treturn :: Monad m => a -> t m a
    tbind   :: Monad m => t m a -> (a -> t m b) -> t m b

class Monad_t t => Trans t where
    lift :: Monad m => m a -> t m a
```

This approach manages to encode the underlying idea behind quantified class constraints. However, there are three main disadvantages to it: a) The monad methods are in fact available for `t m`, but not under the normal `return` and `bind` names.

b) This workaround for quantified class constraints requires the introduction of several extra class and instance declarations, for each polymorpic predicate, rendering the code significantly longer and more obscure. c) The algorithm for transforming the code using quantified class constraints to regular Haskell '98 code, requires a form of flow analysis of the program. This static flow analysis is not always able to derive all necessary information (e.g. in some cases involving irregular data types). As a solution, Trifonov provides a second, more flexible, approach. This representation requires several (widely-supported) language extensions, beyond Haskell '98, namely for allowing type variables to appear as type parameters in instance heads. The main difference is that the two classes from the above example are merged into one. Furthermore, an extra instance declaration is provided to be able to access the standard monad methods:

```
class MonadT t where
    lift    :: Monad m => m a -> t m a
    treturn :: Monad m => a -> t m a
    tbind   :: Monad m => t m a -> (a -> t m b) -> t m b

instance (Monad m, MonadT t) => Monad (t m) where
    return = treturn
    bind   = tbind
```

This second encoding is sufficiently powerful to express most scenarios involving quantified class constraints in the literature (it has been adopted by the Monatron library [12], for example). The main disadvantages of this approach are: a) The head for the `Monad (t m)` instance is very generic, and will easily overlap with other instances. b) This alternative encoding requires the addition of several extra class and instance declarations, for each polymorphic predicate. These result in significantly longer and more obscure code. c) Unfortunately, even the second encoding does not cover the full range of applications, mentioned by the proposal for quantified class constraints. As Trifonov points out:

> *While not a substitute for a language extension, the second approach appears quite useful in solving typical problems involving quantified constraints.*

MonadZipper A second alternative encoding for quantified class constraints is provided by Schrijvers et al. [24] Their approach involves creating a new method in the transformer class, which returns a witness. This witness is a GADT [21], containing local constraints. This way, the quantified constraints can be encoded, and the required axioms can be extracted later on by explicitly pattern matching against the constraint. Consider for instance their monad zipper datatype, which is an extended form of transformer composition, encoded with quantified class constraints:

```
class (forall m . Monad m => Monad (t m)) => Trans t where
    lift :: Monad m => m a -> t m a
```

```
newtype (t1 * t2) m a = C { runC :: t1 (t2 m) a }

instance (Trans t1, Trans t2) => Trans (t1 * t2) where
    lift = C . lift . lift
```

Which can be encoded using the explict witness, as follows:

```
class Trans t where
    lift :: Monad m => m a -> t m a
    mw   :: Monad m => MonadWitness t m

data MonadWitness (t :: (* -> *) -> (* -> *)) m where
    MW :: Monad (t m) => MonadWitness t m

newtype (t1 * t2) m a = C { runC :: t1 (t2 m) a }

instance (Trans t1, Trans t2) => Trans (t1 * t2) where
    lift :: forall m a . Monad m => m a -> (t1 * t2) m a
    lift = case (mw :: MonadWitness t2 m) of
        MW -> C . lift . lift
    mw   = ...
```

By using quantified class constraints, we have access to the required constraint `Monad (t2 m)`, which we need in order to apply the second `lift`. By explicitly pattern matching against the witness, we can achieve the same goal. In the right-hand-side of the case branch, we have access to the local constraint `Monad (t2 m)`, from within the monad witness.

Unfortunately, since the programmer now effictively takes over parts of the type checker's work,[1] the code becomes much longer and cluttered with boilerplate code.

10.2 Alternative Environments

Cochis Schrijvers et al. [25] describe a very similar language with quantified constraints. In their calculus of coherent implicits (Cochis), they support recusive resolution of quantified constraints, in the setting of Scala. There are however some notable differences with our work:

- Since Cochis is situated in a Scala context, there is no distinction between type class dictionaries and Scala terms. Hence, type class constraints are not witnessed by dictionaries. Instead, implicit terms are resolved by means of supplying regular terms. Unlike the approach discussed in this thesis, there is thus no distinction between instance and superclass axioms.

[1]More accurately, the elaboration aspect of type inference.

- Furthermore, Cochis supports local instances, whereas we only consider globally scoped instance declarations. Local instances are allowed to overlap, since the innermost scoped instances are always dominant. Because of this, the resolution presented by Cochis is completely deterministic. This is opposed to our non-deterministic, yet coherent, entailment (due to overlapping local and superclass axioms).

Their work has been a major inspiration for this thesis, most notably their use of a focusing-based approach to resolve constraints.

Coq Coq [27] already natively supports extremely powerful and flexible class constraints in class and instance contexts. However, due to the increased expressivity, constraint resolution can be highly ambiguous and non-terminating. The language designers address this problem by asking the programmer for directions, in Coq's interactive theorem-proving setting. Unfortunately, this approach does not fit well with Haskell's non-interactive type inference tradition.

Chapter 11

Conclusion

This thesis has elaborated the idea of Hinze and Peyton Jones for extending the Haskell type system with quantified class constraints into a complete formal specification and type inference algorithm, with elaboration for programs. In their original work, Hinze and Peyton Jones gave the first specification for quantified class constraints, which is captured by our ambiguous specification of Section 7.1. In this thesis, we have provided a complete formalization of "core" Haskell with the aforementioned feature, by addressing the three main challenges: a) the complete specification of Haskell with quantified class constraints, b) a type inference algorithm that is sound with respect to the specification, as well as c) a dictionary-passing elaboration strategy, which translates our source language to System F.

Furthermore, a fully functional prototype compiler implementation [1] is provided. It supports higher-kinded datatypes and performs type inference, elaboration into System F and type checking of the generated code.

Lastly, we investigated several interesting meta-theoretical properties behind this extension, such as termination of our type inference and soundness of our inference algorithm wrt the focusing-based specification.

Future Work Some interesting future work remains:

- Further investigating the meta-theory and proving certain remaining properties, such as type preservation for our constraint entailment (as explained in Section 9.5) and completeness of our type inference algorithm wrt the focussing-based specification (as explained in Section 9.4.2).

- Quantified class constraints correspond to first-order logical predicates on types (due to the Curry-Howard correspondence). However, as proposed by GHC feature request #5927 [2], quantified class constraints which only abstract over types, may also be extended with quantification over predicates. This added expressivity would correspond to (a fragment of) second order logic.

[1] https://github.com/gkaracha/quantcs-impl

[2] https://ghc.haskell.org/trac/ghc/ticket/5927

- Lastly, this thesis considers the extension for a subset of Haskell. It would be interesting to see how quantified class constraints can be integrated into GHC's OutsideIn type inference [30], and how it behaves in combination with several mainstream type-level features such as functional dependencies [15] or associated type families [4]. If they prove to be orthogonal with each other (or compatible with minor changes), we would like to provide an implementation of quantified class within the GHC compiler.

Bibliography

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

[2] R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In *Proceedings of the Mathematics of Program Construction*, MPC '98, pages 52–67, London, UK, 1998. Springer-Verlag.

[3] G.-J. Bottu, G. Karachalias, T. Schrijvers, B. C. d. S. Oliveira, and P. Wadler. Quantified class constraints. In *Proceedings of the 10th International Symposium on Haskell*, 2017. submitted.

[4] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253, Sept. 2005.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[6] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, Mar. 2006.

[7] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge University Press, New York, NY, USA, 1989.

[8] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, Mar. 1996.

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

[10] R. Hinze. 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, pages 195–228, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] R. Hinze and S. Peyton Jones. Derivable type classes. In *Proceedings of the Fourth Haskell Workshop*, pages 227–236. Elsevier Science, 2000.

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

[13] M. P. Jones. A theory of qualified types. In B. Krieg-Brückner, editor, *ESOP '92*, volume 582 of *LNCS*, pages 287–306. Springer Berlin Heidelberg, 1992.

[14] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, UK, 1995. Springer-Verlag.

[15] M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Programming Languages and Systems*, volume 1782 of *LNCS*, pages 230–244. Springer Berlin Heidelberg, 2000.

[16] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. *SIGPLAN Not.*, 38(3):26–37, Jan. 2003.

[17] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. *SIGPLAN Not.*, 40(9):204–215, Sept. 2005.

[18] J. G. Morris. A simple semantics for haskell overloading. *SIGPLAN Not.*, 49(12):107–118, Sept. 2014.

[19] B. C. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: A new foundation for generic programming. *SIGPLAN Not.*, 47(6):35–44, June 2012.

[20] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, Jan. 2007.

[21] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, Sept. 2006.

[22] F. Pfenning. Lecture notes on focusing, 2010. https://www.cs.cmu.edu/~fp/courses/oregon-m10/04-focusing.pdf.

[23] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

[24] T. Schrijvers and B. C. Oliveira. Monads, zippers and views: Virtualizing the monad stack. *SIGPLAN Not.*, 46(9):32–44, Sept. 2011.

[25] T. Schrijvers, B. C. d. S. Oliveira, and P. Wadler. Cochis: Deterministic and coherent implicits. Report CW 705, KU Leuven, Department of Computer Science, May 2017.

[26] N. Shärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. Technical report, 2002.

[27] M. Sozeau and N. Oury. First-class type classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008. Proceedings*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008.

[28] V. Trifonov. Simulating quantified class constraints. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 98–102, New York, NY, USA, 2003. ACM.

[29] D. Vytiniotis, S. Peyton Jones, and T. Schrijvers. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 39–50, NY, USA, 2010. ACM.

[30] D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. Outsidein(x): Modular type inference with local assumptions. *Journal of Functional Programming*, 21:333–412, September 2011.

[31] P. Wadler and S. Blott. 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, pages 60–76, New York, NY, USA, 1989. ACM.

[32] J. B. Wells. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. Technical report, Boston, MA, USA, 1993.

Appendices

Appendix A

Selected Proofs

This chapter contains the proofs for some of the main desirable meta theoretical properties of the new extended system, as presented in Chapter 9. This chapter thus uses the alternative constraint entailment algorithm, as presented in Figure 9.2.

A.1 Preliminaries

Before setting out to investigate the meta theory behind the new language extension, some new concepts are introduced.

Typing Environment Elaboration Figure A.1 provides the $\Gamma \rightsquigarrow \Delta$ elaboration relation for translating the Γ typing environment into Δ, a System F typing environment. The EnvTy rule is straightforward and handles the type variable binding case. EnvTm provides the translation of a term variable binding, by elaborating the corresponding type σ. EnvTyCon and EnvDataCon show the translation of type constructor and data constructor entries in the environment, respectively. Note that data constructors are already bound to System F types, even in the source language. Finally EnvEmpty handles the final empty case.

$$\frac{\Gamma \rightsquigarrow \Delta}{\Gamma, a \rightsquigarrow \Delta, a}\ \textsc{EnvTy} \qquad \frac{\Gamma \vdash_{\mathtt{ty}} \tau \rightsquigarrow \upsilon \qquad \Gamma \rightsquigarrow \Delta}{\Gamma, x : \sigma \rightsquigarrow \Delta, x : \upsilon}\ \textsc{EnvTm} \qquad \frac{}{\bullet \rightsquigarrow \bullet}\ \textsc{EnvEmpty}$$

$$\frac{\Gamma \rightsquigarrow \Delta}{\Gamma, T \rightsquigarrow \Delta, T}\ \textsc{EnvTyCon} \qquad \frac{\Gamma \rightsquigarrow \Delta}{\Gamma, K : \upsilon \rightsquigarrow \Delta, K : \upsilon}\ \textsc{EnvDataCon}$$

Figure A.1: Typing environment elaboration

Non-Overlapping Substitution The type substitution in this thesis text is defined to be a non-overlapping subsitution. Formally, the definition is provided in figure A.2.

$\boxed{\theta(\sigma) = \sigma}$ Type Substitution Applied to Types

$$\begin{array}{lcl}
\theta(a) & = & \begin{cases} a & \text{, if } a \notin dom(\theta) \\ \tau & \text{, if } [\tau/a] \in \theta \end{cases} \\
\theta(\tau_1 \to \tau_2) & = & \theta(\tau_1) \to \theta(\tau_2) \\
\theta(C \Rightarrow \rho) & = & \theta(C) \Rightarrow \theta(\rho) \\
\theta(\forall a.\sigma) & = & \begin{cases} \forall a.\theta(\sigma) & \text{, if } a \notin dom(\theta) \\ \forall a.\theta'(\sigma) & \text{, if } a \in dom(\theta)\ \&\ \theta' = \theta \setminus a \end{cases}
\end{array}$$

$\boxed{\theta(C) = C}$ Type Substitution Applied to Constraints

$$\begin{array}{lcl}
\theta(TC\ \tau) & = & TC\ \theta(\tau) \\
\theta(C_1 \Rightarrow C_2) & = & \theta(C_1) \Rightarrow \theta(C_2) \\
\theta(\forall a.C) & = & \begin{cases} \forall a.\theta(C) & \text{, if } a \notin dom(\theta) \\ \forall a.\theta'(C) & \text{, if } a \in dom(\theta)\ \&\ \theta' = \theta \setminus a \end{cases}
\end{array}$$

Figure A.2: Non-overlapping substitution definition

A.2 System F Lemmas

The following are several useful lemmas in System F, required for proving the main theorems in the following sections. All of them have been already been extensively described and proven elsewhere.

Lemma 1 represents the preservation of types under substitution in System F.

Lemma 1 (Substitution)

1. *If* $\Delta_1, (x : \upsilon_2), \Delta_2 \vdash^{F}_{\mathtt{tm}} t_1 : \upsilon_1$ *and* $\Delta \vdash^{F}_{\mathtt{tm}} t_2 : \upsilon_2$ *then* $\Delta_1, [t_2/x]\Delta_2 \vdash^{F}_{\mathtt{tm}} [t_2/x]t_1 : \upsilon_1$.

2. *If* $\Delta_1, a, \Delta_2 \vdash^{F}_{\mathtt{tm}} t_1 : \upsilon_1$ *and* $\Delta \vdash^{F}_{\mathtt{ty}} \upsilon_2$ *then* $\Delta_1, [\upsilon_2/a]\Delta_2 \vdash^{F}_{\mathtt{tm}} [\upsilon_2/a]t_1 : [\upsilon_2/a]\upsilon_1$.

3. *If* $\Delta_1, a, \Delta_2 \vdash^{F}_{\mathtt{ty}} \upsilon_1$ *and* $\Delta \vdash^{F}_{\mathtt{ty}} \upsilon_2$ *then* $\Delta_1, [\upsilon_2/a]\Delta_2 \vdash^{F}_{\mathtt{ty}} [\upsilon_2/a]\upsilon_1$.

Lemma 2 describes weakening for the term typing and type well-formedness relations in System F.

Lemma 2 (Weakening)

1. *If* $\Delta \vdash^{F}_{\mathtt{tm}} t : \upsilon_1$ *and* $x \notin dom(\Delta)$ *and* $\Delta \vdash^{F}_{\mathtt{ty}} \upsilon_2$ *then* $\Delta, (x : \upsilon_2) \vdash^{F}_{\mathtt{tm}} t : \upsilon_1$.

2. *If* $\Delta \vdash^{F}_{\mathtt{tm}} t : \upsilon$ *and* $b \notin dom(\Delta)$ *then* $\Delta, b \vdash^{F}_{\mathtt{tm}} t : \upsilon$.

3. *If* $\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon_1$ *and* $x \notin dom(\Delta)$ *and* $\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon_2$ *then* $\Delta, (x : \upsilon_2) \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon_1$.
4. *If* $\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$ *and* $b \notin dom(\Delta)$ *then* $\Delta, b \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$.

Permutation Lemmas In order to state the permutation lemmas for System F, we first define *well-formed* typing environments. The well-formedness relation takes the form $\vdash^{\mathtt{F}}_{\mathtt{wf}} \Delta$ and is given by the following rules:

$$\frac{}{\vdash^{\mathtt{F}}_{\mathtt{wf}} \bullet}\ \textsc{WfNil} \qquad \frac{a \notin \Delta \qquad \vdash^{\mathtt{F}}_{\mathtt{wf}} \Delta}{\vdash^{\mathtt{F}}_{\mathtt{wf}} \Delta, a}\ \textsc{WfTy}$$

$$\frac{x \notin dom(\Delta) \qquad \Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon \qquad \vdash^{\mathtt{F}}_{\mathtt{wf}} \Delta}{\vdash^{\mathtt{F}}_{\mathtt{wf}} \Delta, x : \upsilon}\ \textsc{WfTm}$$

Lemma 3 (Permutation) *If* Δ_1 *is a permutation of* Δ_2 *and* $\vdash^{\mathtt{F}}_{\mathtt{wf}} \Delta_1$ *and* $\vdash^{\mathtt{F}}_{\mathtt{wf}} \Delta_2$ *then the following hold:*

1. $\Delta_1 \vdash^{\mathtt{F}}_{\mathtt{tm}} t : \upsilon$ *if and only if* $\Delta_2 \vdash^{\mathtt{F}}_{\mathtt{tm}} t : \upsilon$.
2. $\Delta_1 \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$ *if and only if* $\Delta_2 \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$.

A.3 Source Language Lemmas

Lemma 4 (Unification Soundness) *If* $\theta = unify(\overline{a}; E, \tau_1 \sim \tau_2)$ *then* $\theta(\tau_1) = \theta(\tau_2)$

Proof: By induction on the derivation.

$\boxed{unify(\overline{a}; \bullet)} \quad = \bullet$

Empty base case.

$\boxed{unify(\overline{a}; E, b \sim b)} \quad = unify(\overline{a}; E)$

The result follows trivially, since $\theta(b) = \theta(b)$ for any θ.

$\boxed{unify(\overline{a}; E, b \sim \tau)} \quad = unify(\overline{a}; \theta(E)) \cdot \theta$

Where $b \notin \overline{a}$, $b \notin fv(\tau)$ and $\theta = [\tau/b]$.
We take $\theta' = unify(\overline{a}; \theta(E))$.
By applying the substitution, we know that $[\tau/b]b = \tau$. We also know that $[\tau/b]\tau = \tau$, since $b \notin fv(\tau)$.
Because of this $\theta'(\theta(b)) = \theta'(\theta(\tau))$ is equivalent to $\theta'(\tau) = \theta'(\tau)$, which trivially holds for any θ'.

$\boxed{unify(\overline{a}; E, \tau \sim b)}$ $\quad = unify(\overline{a}; \theta(E)) \cdot \theta$

Where $b \notin \overline{a}$, $b \notin fv(\tau)$ and $\theta = [\tau/b]$.
The proof is identical to the previous case.

$\boxed{unify(\overline{a}; E, (\tau_1 \to \tau_2) \sim (\tau_3 \to \tau_4))}$ $\quad = unify(\overline{a}; E, \tau_1 \sim \tau_3, \tau_2 \sim \tau_4) = \theta$

By the induction hypothesis:

$$\theta(\tau_2) = \theta(\tau_4) \tag{A.1}$$

By recursively applying the unify rules:

$$\theta = unify(\overline{a}; \theta_1(E, \tau_1 \sim \tau_3)) \cdot \theta_1$$

$$\begin{aligned} \theta_2 &= unify(\overline{a}; \theta_1(E, \tau_1 \sim \tau_3)) \\ &= unify(\overline{a}; \theta_1(E), \theta_1(\tau_1) \sim \theta_1(\tau_3)) \end{aligned} \tag{A.2}$$

By applying the induction hypothesis on equation A.2:

$$(\theta_2 \cdot \theta_1)\ (\tau_1) = (\theta_2 \cdot \theta_1)\ (\tau_3)$$

$$\theta(\tau_1) = \theta(\tau_3) \tag{A.3}$$

So by combining equation A.1 and A.3, it follows that:

$$\theta(\tau_1 \to \tau_2) = \theta(\tau_3 \to \tau_4)$$

The following lemmas are easily proven:

Lemma 5 (Deterministic Elaboration of Types) *If* $\vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon_1$ *and* $\vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon_2$ *then* $\upsilon_1 = \upsilon_2$*.*

Lemma 6 represents the property that the substitution is translation preserving.

Lemma 6 *If* $\Gamma \vdash_{\mathtt{ty}} \sigma$ *and* $\Gamma \vdash_{\mathtt{ty}} \theta(\sigma)$ *and* $\vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon$ *then* $\vdash_{\mathtt{ty}} \theta(\sigma) \rightsquigarrow \theta(\upsilon)$*.*

Lemma 7 describes the property that, when all type instantiations occur from a well-scoped class constraint, the codomain of the constructed substitution has to be well-formed as well under the same environment.

Lemma 7 *If* $\theta = unify(\overline{a}; \tau_1 \sim \tau_2)$ *and* $\Gamma \vdash_{\mathtt{ct}} TC\ \tau_2$ *and* $fv(\tau_1) \subseteq \overline{a}$ *then* $\Gamma \vdash_{\mathtt{codom}}$ *codom*(θ)

Lemma 8 describes weakening for the term typing and type and constraint well-formedness relations in the source language.

Lemma 8 (Weakening)

1. *If* $P; \Gamma_1 \vdash_{\mathtt{tm}} t : \upsilon$ *and* $\vdash_{\mathtt{wf}} \Gamma_1, \Gamma_2$ *then* $P; \Gamma_1, \Gamma_2 \vdash_{\mathtt{tm}} t : \upsilon$.
2. *If* $\Gamma_1 \vdash_{\mathtt{ty}} \sigma$ *and* $\vdash_{\mathtt{wf}} \Gamma_1, \Gamma_2$ *then* $\Gamma_1, \Gamma_2 \vdash_{\mathtt{ty}} \sigma$.
3. *If* $\Gamma_1 \vdash_{\mathtt{ct}} C$ *and* $\vdash_{\mathtt{wf}} \Gamma_1, \Gamma_2$ *then* $\Gamma_1, \Gamma_2 \vdash_{\mathtt{ct}} C$.
4. *If* $\Gamma_1 \vdash_{\mathtt{ax}} A$ *and* $\vdash_{\mathtt{wf}} \Gamma_1, \Gamma_2$ *then* $\Gamma_1, \Gamma_2 \vdash_{\mathtt{ax}} A$.

A.4 Soundness

Lemma 9 represents the soundness of the left algorithmic constraint entailment rules wrt their specification.

Lemma 9 (Sound Left) *If* $\bar{a}; [d : C] \models (d_Q : Q) \mid (A_1 ; \eta) \rightsquigarrow (A_1', A_2 ; \eta'); \theta$ *and* $\forall \Gamma$ *such that* $\Gamma \vdash_{\mathtt{ct}} C$ *and* $\Gamma \vdash_{\mathtt{ct}} Q$ *then* $\Gamma; [\theta(C)] \models \theta(Q) \rightsquigarrow \theta(A_2)$ *and* $\Gamma \vdash_{\mathtt{ax}} A_2$.

Proof: By structural induction on the constraint C.

$\boxed{(\Rightarrow\mathbf{L})}$ $\bar{a}; [d : C_1 \Rightarrow C_2] \models (d_Q : Q) \mid (A_1 ; \eta) \rightsquigarrow (A_1', (d_1 : C_1'), A_2 ; \eta'); \theta$

From the rule hypothesis:

$$\bar{a}; [d_2 : C_2] \models (d_Q : Q) \mid (A_1, (d_1 : C_1) ; \eta \cdot [d\ d_1/d_2]) \rightsquigarrow (A_1', (d_1 : C_1'), A_2 ; \eta'); \theta \tag{A.4}$$

The goal to prove becomes:

$$\Gamma; [\theta(C_1 \Rightarrow C_2)] \models \theta(Q) \rightsquigarrow \theta(C_1', A_2)$$

and $\Gamma \vdash_{\mathtt{ax}} (d_1 : C_1'), A_2$. From the left focusing rules, it is easy to see that $C_1' = \theta(C_1)$. Furthermore, since $\theta(\theta(C_1)) = \theta(C_1)$, the goal becomes:

$$\Gamma; [\theta(C_1 \Rightarrow C_2)] \models \theta(Q) \rightsquigarrow \theta(C_1), \theta(A_2)$$

Since we know $\Gamma \vdash_{\mathtt{ct}} C_1 \Rightarrow C_2$, inverting Rule (C$\Rightarrow$) tells us that $\Gamma \vdash_{\mathtt{ct}} C_1$ and $\Gamma \vdash_{\mathtt{ct}} C_2$.
So by applying the induction hypothesis on Equation A.4:

$$\Gamma; [\theta(C_2)] \models \theta(Q) \rightsquigarrow \theta(A_2)$$

$$\Gamma \vdash_{\mathtt{ax}} A_2$$

It thus follows from Rule ($\Rightarrow$L) that:

$$\Gamma; [\theta(C_1) \Rightarrow \theta(C_2)] \models \theta(Q) \rightsquigarrow \theta(C_1), \theta(A_2)$$

From the definition of substitution we know that $\theta(C_1) \Rightarrow \theta(C_2) = \theta(C_1 \Rightarrow C_2)$. We thus know that:

$$\Gamma; [\theta(C_1 \Rightarrow C_2)] \models \theta(Q) \rightsquigarrow \theta(C_1), \theta(A_2)$$

Furthermore, since we know that $\Gamma \vdash_{\mathtt{ct}} C_1$, and we know $\Gamma \vdash_{\mathtt{ct}} Q$, it follows from lemma 7 that $\Gamma \vdash_{\mathtt{codom}} \mathit{codom}(\theta)$ and thus $\Gamma \vdash_{\mathtt{ct}} \theta(C_1)$. Finally, since we know $\Gamma \vdash_{\mathtt{ax}} A_2$ and we know $C_1' = \theta(C_1)$ and from the side-condition that d_1 is fresh, it follows that:

$$\Gamma \vdash_{\mathtt{ax}} (d_1 : C_1'), A_2$$

$\boxed{(\forall\mathbf{L})}$ $\quad \bar{a}; [d : \forall b.C] \models (d_Q : Q) \mid (A_1 ; \eta) \rightsquigarrow (A_1', A_2 ; \eta'); \theta$

From the rule hypothesis:

$$\bar{a}; [d' : C] \models (d_Q : Q) \mid (A_1 ; \eta \cdot [d\ b/d']) \rightsquigarrow (A_1', A_2 ; \eta'); \theta \tag{A.5}$$

The goal to be proven is the following:

$$\Gamma; [\theta(\forall b.C)] \models \theta(Q) \rightsquigarrow \theta(A_2)$$

and $\Gamma \vdash_{\mathtt{ax}} A_2$.
Since we know that $\Gamma \vdash_{\mathtt{ct}} \forall b.C$, inverting Rule $(C\forall)$ tells us that $\Gamma, b \vdash_{\mathtt{ct}} C$.
Applying weakening (Lemma 8) on $\Gamma \vdash_{\mathtt{ct}} Q$ gives us $\Gamma, b \vdash_{\mathtt{ct}} Q$.
The induction hypothesis, applied to Equation A.5 leads to:

$$\Gamma, b; [\theta(C)] \models \theta(Q) \rightsquigarrow \theta(A_2)$$

$$\Gamma, b \vdash_{\mathtt{ax}} A_2 \tag{A.6}$$

If we take $TC\ \tau_1 = \mathit{head}(\forall b.C)$ and $Q = TC\ \tau_2$, then we know that $\mathit{tch}(\forall b.C) \subseteq \mathit{fv}(\tau_1)$ since all bound variables of a constraint should appear in its head (See Section 7.2.1.)
Furthermore, since we know that $\theta = \mathit{unify}(\bar{a}; \tau_1 \sim \tau_2)$ from $(Q\mathrm{L})$, it follows that $b \in \mathit{dom}(\theta)$ since obviously $b \in \mathit{tch}(\forall b.C)$.
Since we know that τ is contained within τ_2, and we know that $\Gamma \vdash_{\mathtt{ct}} Q$, it follows from inverting (CQ) that $\Gamma \vdash_{\mathtt{ty}} \tau_2$ and by extension $\Gamma \vdash_{\mathtt{ty}} \tau$.
Furthermore, we know $\theta = [\tau/b]\theta'$ (where $\theta' = \theta \setminus [\tau/b]$) since $\Gamma \vdash_{\mathtt{ty}} \tau$, which means that the codomain of the substution is independent: e.g. it doesn't depend on any previous substitutions, only on Γ. Because of this the substitution can be permutated.
We thus know:

$$\theta(C) = ([\tau/b]\theta')\ (C) = [\tau/b](\theta'(C))$$

And because of the definition of substitution:

$$\theta(\forall b.C) = \forall b.(\theta'(C))$$

Finally by applying Rule (∀L):

$$\Gamma; [\forall b.(\theta'(C))] \models \theta(Q) \rightsquigarrow \theta(A_2)$$

which is thus equivalent to:

$$\Gamma; [\theta(\forall b.C)] \models \theta(Q) \rightsquigarrow \theta(A_2)$$

Lastly, because of (QL) we know that $A_2 = \theta(A'_2)$ for some A'_2. We know $b \in \mathit{dom}(\theta)$ and since $\Gamma \vdash_{\mathsf{ct}} Q$, we also know that $\mathit{codom}(\theta)$ is well-formed under Γ (by lemma 7). A_2 thus doesn't contain a b variable. It follows from equation A.6 that $\Gamma \vdash_{\mathsf{ax}} A_2$.

$\boxed{(Q\mathbf{L})}$ $\quad \overline{a}; [d : TC\ \tau_1] \models (d' : TC\ \tau_2) \mid (A_1 ; \eta) \rightsquigarrow (\theta(A_1) ; \theta(\eta \cdot [d/d'])); \theta$

From the rule hypothesis:

$$\theta = \mathit{unify}(\overline{a}; \tau_1 \sim \tau_2)$$

The goal to be proven is the following:

$$\Gamma; [\theta(TC\ \tau_1)] \models \theta(TC\ \tau_2) \rightsquigarrow \bullet$$

and $\Gamma \vdash_{\mathsf{ax}} \bullet$, which trivially holds.
By (QL), this goal reduces to:

$$\theta(TC\ \tau_1) = \theta(TC\ \tau_2)$$

which follows immediately from Lemma 4.
Lemma 10 represents the soundness of the right algorithmic constraint entailment rules wrt their specification.

Lemma 10 (Sound Right) *If* $\overline{a}; A \models [d : C] \rightsquigarrow A' ; \eta ; \theta$ *and* $\forall \Gamma$ *such that* $\Gamma \vdash_{\mathsf{ct}} C$ *and* $\Gamma \vdash_{\mathsf{ax}} A$ *and* $\mathit{fv}(C) \subseteq \overline{a}$ *then* $\theta(A); \Gamma \models [\theta(C)] \Leftrightarrow (\forall (d_i : C_i) \in A' : \theta(A); \Gamma \models [\theta(C_i)])$ *and* $\Gamma \vdash_{\mathsf{ax}} A'$.

Proof: By structural induction on the constraint C.

$\boxed{(\Rightarrow\mathbf{R})}$ $\quad \overline{a}; A \models [d : C_1 \Rightarrow C_2] \rightsquigarrow \overline{(d' : C_1 \Rightarrow C)} ; \eta' ; \theta$

From the rule hypothesis:

$$\overline{a}; A, (d_1 : C_1) \models [d_2 : C_2] \rightsquigarrow \overline{(d_C : C)} ; \eta ; \theta \tag{A.7}$$

$$\eta' = [\lambda(d_1 : v_1).([d'\ d_1/d_C](\eta(d_2)))/d]$$

The goals to be proven are the following:

$$(\theta(A); \Gamma \models [\theta(C_1 \Rightarrow C_2)]) \Leftrightarrow (\forall C_i \in \overline{C_1 \Rightarrow C} : \theta(A); \Gamma \models [\theta(C_i)])$$

and $\Gamma \vdash_{\mathsf{ax}} \overline{(d' : C_1 \Rightarrow C)}$
Since we know $\Gamma \vdash_{\mathsf{ct}} C_1 \Rightarrow C_2$, inverting (C⇒) tells us that $\Gamma \vdash_{\mathsf{ct}} C_1$ and $\Gamma \vdash_{\mathsf{ct}} C_2$.
Because of this, $\Gamma \vdash_{\mathsf{ax}} A$ and the fact that d_1 is fresh by the side condition of the rule, we know that $\Gamma \vdash_{\mathsf{ax}} A, (d_1 : C_1)$ holds as well.
Since $fv(C_1 \Rightarrow C_2) \subseteq \overline{a}$, obviously $fv(C_2) \subseteq \overline{a}$ holds as well.
From the induction hypothesis on equation A.7, it thus follows that:

$$(\theta(A, C_1); \Gamma \models [\theta(C_2)]) \Leftrightarrow (\forall C_i \in \overline{C} : \theta(A, C_1); \Gamma \models [\theta(C_i)]) \tag{A.8}$$

$$\Gamma \vdash_{\mathsf{ax}} \overline{(d_C : C)}$$

We split up the goal equivalence in 2 implications:

- $(\forall C_i \in \overline{C_1 \Rightarrow C} : \theta(A); \Gamma \models [\theta(C_i)]) \Rightarrow (\theta(A); \Gamma \models [\theta(C_1 \Rightarrow C_2)])$:

  The left hand side is obviously equivalent to:

  $$\forall C_i \in \overline{C} : \theta(A); \Gamma \models [\theta(C_1 \Rightarrow C_i)]$$

  Which is in turn equivalent to:

  $$\forall C_i \in \overline{C} : \theta(A); \Gamma \models [\theta(C_1) \Rightarrow \theta(C_i)]$$

  By inverting (⇒R), we know that:

  $$\forall C_i \in \overline{C} : \theta(A), \theta(C_1); \Gamma \models [\theta(C_i)]$$

  Which is equivalent to:

  $$\forall C_i \in \overline{C} : \theta(A, C_1); \Gamma \models [\theta(C_i)]$$

  Because of equation A.8, if follows that:

  $$\theta(A, C_1); \Gamma \models [\theta(C_2)]$$

  Which is equivalent to:

  $$\theta(A), \theta(C_1); \Gamma \models [\theta(C_2)]$$

  By applying (⇒R):

  $$\theta(A); \Gamma \models [\theta(C_1) \Rightarrow \theta(C_2)]$$

  Which is equivalent to:

  $$\theta(A); \Gamma \models [\theta(C_1 \Rightarrow C_2)]$$

- $(\theta(A); \Gamma \models [\theta(C_1 \Rightarrow C_2)]) \Rightarrow (\forall C_i \in \overline{C_1 \Rightarrow C} : \theta(A); \Gamma \models [\theta(C_i)])$:

  The left hand side is equivalent to:

  $$\theta(A); \Gamma \models [\theta(C_1) \Rightarrow \theta(C_2)]$$

  By inverting ($\Rightarrow$R), we know:

  $$\theta(A), \theta(C_1); \Gamma \models [\theta(C_2)]$$

  Which is equivalent to:

  $$\theta(A, C_1); \Gamma \models [\theta(C_2)]$$

  By equation A.8, it follows that:

  $$\forall C_i \in \overline{C} : \theta(A, C_1); \Gamma \models [\theta(C_i)]$$

  Which is equivalent to:

  $$\forall C_i \in \overline{C} : \theta(A), \theta(C_1); \Gamma \models [\theta(C_i)]$$

  By ($\Rightarrow$R):

  $$\forall C_i \in \overline{C} : \theta(A); \Gamma \models [\theta(C_1 \Rightarrow C_i)]$$

  Which is equivalent to:

  $$\forall C_i \in \overline{C_1 \Rightarrow C} : \theta(A); \Gamma \models [\theta(C_i)]$$

Lastly, since $\Gamma \vdash_{\mathsf{ax}} (d_1 : C_1)$ and $\Gamma \vdash_{\mathsf{ax}} \overline{(d_C : C)}$, (C$\Rightarrow$) tells us that $\Gamma \vdash_{\mathsf{ax}} \overline{(d' : C_1 \Rightarrow C)}$

$\boxed{(\forall\mathbf{R})}$ $\quad \overline{a}; A \models [d : \forall b. C_0] \rightsquigarrow \overline{(d' : \forall b'. C)}\,; \eta'\,; \theta$

From the rule hypothesis:

$$\overline{a}, b; A \models [d_C : C_0] \rightsquigarrow \overline{(d'' : C)}\,; \eta\,; \theta \tag{A.9}$$

$$\eta' = [\Lambda b'. \overline{[d'\ b'/d'']}(\eta(d_C))/d]$$

The goals to be proven are:

$$(\theta(A); \Gamma \models [\theta(\forall b. C_0)]) \Leftrightarrow (\forall C_i \in \overline{\forall b'. C} : \theta(A); \Gamma \models [\theta(C_i)])$$

and $\Gamma \vdash_{\mathsf{ax}} \overline{(d' : \forall b'. C)}$.
Since we know $\Gamma \vdash_{\mathsf{ct}} \forall b. C_0$, it follows from inverting (C$\forall$) that $\Gamma, b \vdash_{\mathsf{ct}} C_0$.
By applying weakening (lemma 8) on $\Gamma \vdash_{\mathsf{ct}} A$, we get $\Gamma, b \vdash_{\mathsf{ct}} A$.
Since $fv(\forall b. C_0) \subseteq \overline{a}$, we know $fv(C_0) \subseteq \overline{a}, b$.
So from the induction hypothesis, applied to equation A.9, it follows that:

$$(\theta(A); \Gamma, b \models [\theta(C_0)]) \Leftrightarrow (\forall C_i \in \overline{C} : \theta(A); \Gamma, b' \models [\theta(C_i)]) \tag{A.10}$$

and $\Gamma, b' \vdash_{\mathsf{ax}} \overline{(d'' : C)}$.
Because of (C$\forall$), it follows that $\Gamma \vdash_{\mathsf{ax}} \overline{(d' : \forall b'.C)}$.

Since b' is a freshly generated type variable in ($\forall$R), we know that

$$\theta(\forall b'.C_i) = \forall b'.\theta(C_i) \tag{A.11}$$

Take $TC\ \tau_1 = head(\forall b.C_0)$. We know $tch(\forall b.C_0) \subseteq fv(\tau_1)$ since all declared variables have to occur in the head of the constraint.
We know $b \notin dom(\theta)$, since $fv(\tau_1) \subseteq \overline{a}$ and $\theta = unify(\overline{a}; \tau_1 \sim \tau_2)$.
It thus follows that:

$$\theta(\forall b.C_0) = \forall b.\theta(C_0) \tag{A.12}$$

We split the goal equivalence in 2 implications:

- $(\forall C_i \in \overline{\forall b'.C} : \theta(A); \Gamma \models [\theta(C_i)]) \Rightarrow (\theta(A); \Gamma \models [\theta(\forall b.C_0)])$:

  The left hand side is equivalent to:

  $$\forall C_i \in \overline{C} : \theta(A); \Gamma \models [\theta(\forall b'.C_i)]$$

  Because of equation A.11:

  $$\forall C_i \in \overline{C} : \theta(A); \Gamma \models [\forall b'.\theta(C_i)]$$

  By inverting ($\forall$R), it follows that:

  $$\forall C_i \in \overline{C} : \theta(A); \Gamma, b' \models [\theta(C_i)]$$

  It follows from equation A.10 that:

  $$\theta(A); \Gamma, b \models [\theta(C_0)]$$

  By applying ($\forall$R):

  $$\theta(A); \Gamma \models [\forall b.\theta(C_0)]$$

  So by equation A.12:

  $$\theta(A); \Gamma \models [\theta(\forall b.C_0)]$$

- $(\theta(A); \Gamma \models [\theta(\forall b.C_0)]) \Rightarrow (\forall C_i \in \overline{\forall b'.C} : \theta(A); \Gamma \models [\theta(C_i)])$:

  Because of equation A.12:

  $$\theta(A); \Gamma \models [\forall b.\theta(C_0)]$$

  By inverting ($\forall$R):

  $$\theta(A); \Gamma, b \models [\theta(C_0)]$$

  By applying equation A.10, we know that:

  $$\forall C_i \in \overline{C} : \theta(A); \Gamma, b' \models [\theta(C_i)]$$

By applying $(\forall\text{R})$:

$$\forall C_i \in \overline{C} : \theta(A); \Gamma \models [\forall b'.\theta(C_i)]$$

It follows from equation A.11 that:

$$\forall C_i \in \overline{C} : \theta(A); \Gamma \models [\theta(\forall b'.C_i)]$$

Which is obviously equivalent to:

$$\forall C_i \in \overline{\forall b'.C} : \theta(A); \Gamma \models [\theta(C_i)]$$

$\boxed{(Q\mathbf{R})}$ $\quad \overline{a}; A \models [(d : Q)] \rightsquigarrow A' \,;\eta\,; \theta$

From the rule hypothesis:

$$(d_i : C_i) \in A : \overline{a}; [d_i : C_i] \models (d : Q) \mid (\bullet\,;\bullet) \rightsquigarrow (A'\,;\eta); \theta \tag{A.13}$$

The goal to be proven is the following:

$$(\theta(A); \Gamma \models [\theta(Q)]) \Leftrightarrow (\forall C_i \in A' : \theta(A); \Gamma \models [\theta(C_i)])$$

and $\Gamma \vdash_{\mathtt{ax}} A'$
Since $\Gamma \vdash_{\mathtt{ax}} A$, we know that $\forall (d_i : C_i) \in A : \Gamma \vdash_{\mathtt{ct}} C_i$.
Thus by applying lemma 9 on equation A.13:

$$\Gamma; [\theta(C_i)] \models \theta(Q) \rightsquigarrow \theta(A') \tag{A.14}$$

and $\Gamma \vdash_{\mathtt{ax}} A'$.
We split the wanted equivalence in 2 parts:

- $(\forall C_i \in A' : \theta(A); \Gamma \models [\theta(C_i)]) \Rightarrow (\theta(A); \Gamma \models [\theta(Q)])$:

  The left hand side is equivalent to:

  $$\forall C_i \in \theta(A') : \theta(A); \Gamma \models [C_i]$$

  Equation A.14 is equivalent to:

  $$(d_i : C_i) \in \theta(A) : \Gamma; [C_i] \models \theta(Q) \rightsquigarrow \theta(A')$$

  Thus, by $(Q\text{R})$, we know:

  $$\theta(A); \Gamma \models [\theta(Q)]$$

- $(\theta(A); \Gamma \models [\theta(Q)]) \Rightarrow (\forall C_i \in A' : \theta(A); \Gamma \models [\theta(C_i)])$:

  By equation A.14 and inverting $(Q\text{R})$, it follows that:

  $$\forall C_i \in \theta(A') : \theta(A); \Gamma \models [C_i]$$

  Which is equivalent to:

  $$\forall C_i \in A' : \theta(A); \Gamma \models [\theta(C_i)]$$

Lemma 11 represents the soundness of the recursive algorithmic constraint entailment rules wrt their specification.

Lemma 11 (Sound Recursive) *If* $\overline{a}; A \models A_1 \rightsquigarrow A_2; \eta; \theta$ *and* $\forall \Gamma$ *such that* $\Gamma \vdash_{\mathsf{ax}} A$ *and* $\Gamma \vdash_{\mathsf{ax}} A_1$ *and* $fv(A_1) \subseteq \overline{a}$ *then* $(\forall (d_i : C_i) \in A_1 : \theta(A); \Gamma \models \theta(C_i)) \Leftrightarrow (\forall (d_i : C_i) \in A_2 : \theta(A); \Gamma \models \theta(C_i))$ *and* $\Gamma \vdash_{\mathsf{ax}} A_2$.

Proof: By induction on the constraint entailment derivation.

$\boxed{\textbf{Stop}}$ $\overline{a}; A \models A_1 \rightsquigarrow A_1; \bullet; \bullet$

The goal becomes:

$$(\forall (d_1 : C_i) \in A_1 : \theta(A); \Gamma \models \theta(C_i)) \Leftrightarrow (\forall (d_i : C_i) \in A_1 : \theta(A); \Gamma \models \theta(C_i))$$

and $\Gamma \vdash_{\mathsf{ax}} \bullet$.
Both of which are trivially true.

$\boxed{\textbf{Step}}$ $\overline{a}; A \models A_1, (d : C) \rightsquigarrow A_3; (\eta_2 \cdot \eta_1); \theta$

From the rule hypothesis:

$$\overline{a}; A \models [d : C] \rightsquigarrow A_2; \eta_1; \theta_1 \qquad (A.15)$$

$$\overline{a}; A \models A_1, A_2 \rightsquigarrow A_3; \eta_2; \theta_2 \qquad (A.16)$$

where $\theta = \theta_2 \cdot \theta_1$.
The goal to be proven becomes:

$$(\forall (d_i : C_i) \in A_1, (d : C) : \theta(A); \Gamma \models d_i : \theta(C_i)) \Leftrightarrow (\forall (d_i : C_i) \in A_3 : \theta(A); \Gamma \models d_i : \theta(C_i))$$

and $\Gamma \vdash_{\mathsf{ax}} A_3$.
Since $\Gamma \vdash_{\mathsf{ax}} A_1$, for any $(d_i : C_i) \in A_1 : \Gamma \vdash_{\mathsf{ct}} C_i$.
So by applying lemma 10 on equation A.15:

$$(\theta_1(A); \Gamma \models [\theta_1(C)]) \Leftrightarrow (\forall (d_i : C_i) \in A_2 : \theta_1(A); \Gamma \models [\theta_1(C_i)]) \qquad (A.17)$$

and $\Gamma \vdash_{\mathsf{ax}} A_2$.
Since we know $fv(A_1) \subseteq \overline{a}$, by the way A_2 is constructed in the left rules, we know $fv(A_2) \subseteq \overline{a}$ and thus by extension $fv(A_1, A_2) \subseteq \overline{a}$.
Furthermore, because we know $\Gamma \vdash_{\mathsf{ax}} A_1$ and $\Gamma \vdash_{\mathsf{ax}} A_2$, obviously $\Gamma \vdash_{\mathsf{ax}} A_1, A_2$ holds as well.
It then follows from the induction hypothesis, applied to equation A.16, that:

$$(\forall (d_i : C_i) \in A_1, A_2 : \theta_2(A); \Gamma \models \theta_2(C_i)) \Leftrightarrow (\forall (d_i : C_i) \in A_3 : \theta_2(A); \Gamma \models \theta_2(C_i)) \qquad (A.18)$$

and $\Gamma \vdash_{\mathsf{ax}} A_3$.
It is obvious to see that:

$$(\forall (d_i : C_i) \in A_1, A_2 : \theta_2(A); \Gamma \models \theta_2(C_i)) \Leftrightarrow$$
$$(\forall (d_i : C_i) \in A_1 : \theta_2(A); \Gamma \models \theta_2(C_i) \textit{ and } \forall (d_i : C_i) \in A_2 : \theta_2(A); \Gamma \models \theta_2(C_i)) \qquad (A.19)$$

$$
\begin{aligned}
&(\forall (d_i : C_i) \in A_1, (d : C) : \theta(A); \Gamma \models d_i : \theta(C_i)) \Leftrightarrow \\
&(\forall (d_i : C_i) \in A_1 : \theta(A); \Gamma \models d_i : \theta(C_i) \textit{ and } \theta(A); \Gamma \models \theta(C))
\end{aligned}
\tag{A.20}
$$

Because of freshness conditions and the separation between the creation of θ_1 and θ_2, we know that:

$$\forall (d_i : C_i) \in A_2 : \theta_1(C_i) = \theta_2(\theta_1(C_i))$$

$$\forall (d_i : C_i) \in A_3 : \theta_2(C_i) = \theta_2(\theta_1(C_i))$$

$$\forall (d_i : C_i) \in A_1, A_2 : \theta_2(C_i) = \theta_2(\theta_1(C_i))$$

We know that $dom(\theta) = tch(A)$, so they completely overlap. Because of the non-overlapping substitution definition, it follows that:

$$\theta(A) = \theta_1(A) = \theta_2(A) = A$$

Thus, by combining these equalities with equations A.17, A.19 and A.20:

$$(\forall (d_i : C_i) \in A_1, A_2 : \theta(A); \Gamma \models \theta(C_i)) \Leftrightarrow (\forall (d_i : C_i) \in A_1, (d : C) : \theta(A); \Gamma \models \theta(C_i))$$

Finally, because of transitivity of equivalence and equation A.18:

$$(\forall (d_i : C_i) \in A_1, (d : C) : \theta(A); \Gamma \models \theta(C_i)) \Leftrightarrow (\forall (d_i : C_i) \in A_3 : \theta(A); \Gamma \models \theta(C_i))$$

A.5 Equivalence between Algorithmic Constraint Entailments

Lemmas 12, 13 and 14 describe the equivalence between the two different versions of the constraint entailment algorithm, as decribed in Figures 7.4 and 9.2. The algorithmic relation from Figure 7.4 is annotated with an A, whereas the intermediate algorithmic rules from Figure 9.2 are annotated with a B. The proofs for lemma 13 and 14 are entirely straighforward.

Lemma 12 (Equivalence Left) $\bar{a}; [d : C] \models_A (d_Q : Q) \rightsquigarrow A_2; \theta ; \eta_2$
$\Leftrightarrow \bar{a}; [d : C] \models_B (d_Q : Q) \mid (A_1 ; \eta_1) \rightsquigarrow (\theta(A_1), A_2 ; \theta(\eta_1) \cdot \eta_2); \theta$

Proof: By structural induction on the constraint C.

$\boxed{(\Rightarrow\mathbf{L})}$

$$
\begin{aligned}
&\bar{a}; [d : C_1 \Rightarrow C_2] \models_A (d_Q : Q) \rightsquigarrow A_2, (d_1 : \theta(C_1)); \theta ; [d\ d_1/d_2] \cdot \eta_2 \\
\Leftrightarrow &\bar{a}; [d : C_1 \Rightarrow C_2] \models_B (d_Q : Q) \mid (A_1 ; \eta_1) \rightsquigarrow (\theta(A_1), (d_1 : \theta(C_1)), A_2 ; \theta(\eta_1) \cdot [d\ d_1/d_2] \cdot \eta_2); \theta
\end{aligned}
$$

We split the equivalence in two parts:

- $\bar{a}; [d : C_1 \Rightarrow C_2] \models_A (d_Q : Q) \rightsquigarrow A_2, (d_1 : \theta(C_1)); \theta ; [d\ d_1/d_2] \cdot \eta_2$
  $\Rightarrow \bar{a}; [d : C_1 \Rightarrow C_2] \models_B (d_Q : Q) \mid (A_1 ; \eta_1) \rightsquigarrow (A' ; \theta(\eta_1) \cdot [d\ d_1/d_2] \cdot \eta_2); \theta$

where $A' = \theta(A_1), (d_1 : \theta(C_1)), A_2$:
From the rule hypothesis:

$$\overline{a}; [d_2 : C_2] \models_A (d_Q : Q) \rightsquigarrow A_2; \theta\, ; \eta_2$$

By applying the induction hypothesis:

$$\overline{a}; [d_2 : C_2] \models_B (d_Q : Q) \mid (A'_1\, ; \eta'_1) \rightsquigarrow (\theta(A'_1), A_2\, ; \theta(\eta'_1) \cdot \eta_2); \theta$$

Apply the $(\Rightarrow\mathrm{L})_B$ rule with $A'_1 = A_1, (d_1 : C_1)$ and $\eta'_1 = \eta_1 \cdot [d\ d_1/d_2]$:

$$\overline{a}; [d : C_1 \Rightarrow C_2] \models_B (d_Q : Q) \mid (A_1\, ; \eta_1) \rightsquigarrow (A'\, ; \theta(\eta_1) \cdot [d\ d_1/d_2] \cdot \eta_2); \theta$$

where $A' = \theta(A_1), (d_1 : \theta(C_1)), A_2$,
since $\theta([d\ d_1/d_2]) = [d\ d_1/d_2]$.

- $\overline{a}; [d : C_1 \Rightarrow C_2] \models_B (d_Q : Q) \mid (A_1\, ; \eta_1) \rightsquigarrow (A'\, ; \theta(\eta_1) \cdot [d\ d_1/d_2] \cdot \eta_2); \theta$
  $\Rightarrow \overline{a}; [d : C_1 \Rightarrow C_2] \models_A (d_Q : Q) \rightsquigarrow A_2, (d_1 : \theta(C_1)); \theta\, ; [d\ d_1/d_2] \cdot \eta_2$
  where $A' = \theta(A_1), (d_1 : \theta(C_1)), A_2$:
  From the rule hypothesis:

  $$\overline{a}; [d_2 : C_2] \models_B d_Q : Q \mid (A_1, (d_1 : C_1)\, ; \eta_1 \cdot [d\ d_1/d_2]) \rightsquigarrow (A'\, ; \theta(\eta_1) \cdot [d\ d_1/d_2] \cdot \eta_2); \theta$$

  By applying the induction hypothesis:

  $$\overline{a}; [d_2 : C_2] \models_A (d_Q : Q) \rightsquigarrow A_2; \theta\, ; \eta_2$$

  Finally apply $(\Rightarrow\mathrm{L})_A$:

  $$\overline{a}; [d : C_1 \Rightarrow C_2] \models_A (d_Q : Q) \rightsquigarrow A_2, (d_1 : \theta(C_1)); \theta\, ; [d\ d_1/d_2] \cdot \eta_2$$

$\boxed{(\forall\mathbf{L})}$

$$\overline{a}; [d : \forall b.C] \models_A (d_Q : Q) \rightsquigarrow A_2; \theta\, ; [d\ \theta(b)/d'] \cdot \eta_2$$
$$\Leftrightarrow \overline{a}; [d : \forall b.C] \models_B (d_Q : Q) \mid (A_1\, ; \eta_1) \rightsquigarrow (\theta(A_1), A_2\, ; \theta(\eta_1) \cdot [d\ \theta(b)/d'] \cdot \eta_2); \theta$$

We split the equivalence in two parts:

- $\overline{a}; [d : \forall b.C] \models_A (d_Q : Q) \rightsquigarrow A_2; \theta\, ; [d\ \theta(b)/d'] \cdot \eta_2$
  $\Rightarrow \overline{a}; [d : \forall b.C] \models_B (d_Q : Q) \mid (A_1\, ; \eta_1) \rightsquigarrow (\theta(A_1), A_2\, ; \theta(\eta_1) \cdot [d\ \theta(b)/d'] \cdot \eta_2); \theta$:
  From the rule hypothesis:

  $$\overline{a}; [d' : C] \models_A (d_Q : Q) \rightsquigarrow A_2; \theta\, ; \eta_2$$

  By applying the induction hypothesis:

  $$\overline{a}; [d' : C] \models_B (d_Q : Q) \mid (A'_1\, ; \eta'_1) \rightsquigarrow (\theta(A'_1), A_2\, ; \theta(\eta'_1) \cdot \eta_2); \theta$$

  Apply the $(\forall\mathrm{L})_B$ rule with $A'_1 = A_1$ and $\eta'_1 = \eta_1 \cdot [d\ b/d']$:

  $$\overline{a}; [d : \forall b.C] \models_B (d_Q : Q) \mid (A_1\, ; \eta_1) \rightsquigarrow (\theta(A_1), A_2\, ; \theta(\eta_1) \cdot [d\ \theta(b)/d'] \cdot \eta_2); \theta$$

  since $\theta([d\ b/d']) = [d\ \theta(b)/d']$.

- $\overline{a}; [d : \forall b.C] \models_B (d_Q : Q) \mid (A_1 ; \eta_1) \rightsquigarrow (\theta(A_1), A_2 ; \theta(\eta_1) \cdot [d\ \theta(b)/d'] \cdot \eta_2); \theta$
  $\Rightarrow \overline{a}; [d : \forall b.C] \models_A (d_Q : Q) \rightsquigarrow A_2; \theta ; [d\ \theta(b)/d'] {\cdot} \eta_2$:
  From the rule hypothesis:

  $$\overline{a}; [d' : C] \models_B d_Q : Q \mid (A_1 ; \eta_1 \cdot [d\ b/d']) \rightsquigarrow (\theta(A_1), A_2 ; \theta(\eta_1) \cdot [d\ \theta(b)/d'] \cdot \eta_2); \theta$$

  By applying the induction hypthesis:

  $$\overline{a}; [d : \forall b.C] \models_A (d_Q : Q) \rightsquigarrow A_2; \theta ; [d\ \theta(b)/d'] {\cdot} \eta_2$$

$$\boxed{(Q\mathbf{L})} \qquad \begin{aligned} &\overline{a}; [d' : TC\ \tau_1] \models_A (d : TC\ \tau_2) \rightsquigarrow \bullet; \theta ; [d'/d] \\ \Leftrightarrow &\overline{a}; [d' : TC\ \tau_1] \models_B (d : TC\ \tau_2) \mid (A_1 ; \eta_1) \rightsquigarrow (\theta(A_1) ; \theta(\eta_1) \cdot [d'/d]); \theta \end{aligned}$$

We split the equivalence in two parts:

- $\overline{a}; [d' : TC\ \tau_1] \models_A (d : TC\ \tau_2) \rightsquigarrow \bullet; \theta ; [d'/d]$
  $\Rightarrow \overline{a}; [d' : TC\ \tau_1] \models_B (d : TC\ \tau_2) \mid (A_1 ; \eta_1) \rightsquigarrow (\theta(A_1) ; \theta(\eta_1) \cdot [d'/d]); \theta$:
  From the rule hypothesis:

  $$\mathit{unify}(\overline{a}; \tau_1 \sim \tau_2)$$

  The result follows immediately from the $(Q\mathrm{L})_B$ rule.

- $\overline{a}; [d' : TC\ \tau_1] \models_B (d : TC\ \tau_2) \mid (A_1 ; \eta_1) \rightsquigarrow (\theta(A_1) ; \theta(\eta_1) \cdot [d'/d]); \theta$
  $\Rightarrow \overline{a}; [d' : TC\ \tau_1] \models_A (d : TC\ \tau_2) \rightsquigarrow \bullet; \theta ; [d'/d]$:
  From the rule hypothesis:

  $$\mathit{unify}(\overline{a}; \tau_1 \sim \tau_2)$$

  The result follows immediately from the $(Q\mathrm{L})_A$ rule.

Lemma 13 (Equivalence Right) $\overline{a}; A \models_A [d : C] \rightsquigarrow A' ; \eta ; \theta$
$\Leftrightarrow \overline{a}; A \models_B [d : C] \rightsquigarrow A' ; \eta ; \theta$

Lemma 14 (Equivalence Recursive) $\overline{a}; A \models_A A_1 \rightsquigarrow A_2; \eta ; \theta$
$\Leftrightarrow \overline{a}; A \models_B A_1 \rightsquigarrow A_2; \eta ; \theta$

A.6 Preservation of Typing under Elaboration

Lemma 15 and lemma 17 state that the translation preserves the well-formedness of types. Lemma 16 is analogous for the translation of constraints.

Lemma 15 (Type Preservation for Monotypes) *If* $\Gamma \vdash_{\mathtt{ty}} \tau$ *and* $\vdash_{\mathtt{ty}} \tau \rightsquigarrow \upsilon$ *and* $\Gamma \rightsquigarrow \Delta$ *then* $\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$.

Proof: By structural induction on the type τ and the corresponding well-formedness and elaboration rules.

$\boxed{\textbf{TyVar}}$ $\quad \Gamma \vdash_{\mathtt{ty}} a$ *and* $\vdash_{\mathtt{ty}} a \rightsquigarrow a$

From the well-formedness rule hypothesis, it follows that $a \in \Gamma$, so obviously $a \in \Delta$ holds as well. Because of F-TyVar, we may conclude that:

$$\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} a$$

$\boxed{\textbf{TyArr}}$ $\quad \Gamma \vdash_{\mathtt{ty}} \tau_1 \to \tau_2$ *and* $\vdash_{\mathtt{ty}} \tau_1 \to \tau_2 \rightsquigarrow \upsilon_1 \to \upsilon_2$

By the rule hypothesis:

$$\Gamma \vdash_{\mathtt{ty}} \tau_1 \ \textit{and} \ \vdash_{\mathtt{ty}} \tau_1 \rightsquigarrow \upsilon_1$$

$$\Gamma \vdash_{\mathtt{ty}} \tau_2 \ \textit{and} \ \vdash_{\mathtt{ty}} \tau_2 \rightsquigarrow \upsilon_2$$

So it follows from the induction hypothesis:

$$\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon_1$$

$$\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon_2$$

Finally, because of F-TyArr:

$$\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon_1 \to \upsilon_2$$

Lemma 16 (Type Preservation for Constraints) *If* $\Gamma \vdash_{\mathtt{ct}} C$ *and* $\vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon$ *and* $\Gamma \rightsquigarrow \Delta$ *then* $\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$.

Proof: By structural induction on the constraint C and the corresponding well-formedness and elaboration rules.

$\boxed{(\mathbf{C}Q)}$ $\quad \Gamma \vdash_{\mathtt{ct}} TC\ \tau$ *and* $\vdash_{\mathtt{ct}} TC\ \tau \rightsquigarrow T_{TC}\ \upsilon$

By the rule hypothesis we know:

$$\Gamma \vdash_{\mathtt{ty}} \tau \ \textit{and} \ \vdash_{\mathtt{ty}} \tau \rightsquigarrow \upsilon$$

Theorem 17 then tells us that:

$$\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$$

As explained in text, the well-formedness rule also has the implicit premises that *TC* is in store. Because of this $T_{TC} \in \Delta$ has to hold as well. It thus follows from F-TyCon that:

$$\Delta \vdash^{F}_{ty} T_{TC}$$

So by applying TyApp, we know that:

$$\Delta \vdash^{F}_{ty} T_{TC}\ \upsilon$$

$\boxed{\textbf{(C}\forall\textbf{)}}$ $\Gamma \vdash_{ct} \forall a.C$ *and* $\vdash_{ct} \forall a.C \rightsquigarrow \forall a.\upsilon$

From the rule hypothesis:

$$\Gamma, a \vdash_{ct} C \textit{ and } \vdash_{ct} C \rightsquigarrow \upsilon$$

So from the induction hypothesis, it follows that:

$$\Delta_2 \vdash^{F}_{ty} \upsilon$$

Where $\Gamma, a \rightsquigarrow \Delta_2$. And since we know $\Delta_2 = \Delta, a$, by rule F-TyAll:

$$\Delta \vdash^{F}_{ty} \forall a.\upsilon$$

$\boxed{\textbf{(C}\Rightarrow\textbf{)}}$ $\Gamma \vdash_{ct} C_1 \Rightarrow C_2$ *and* $\vdash_{ct} C_1 \Rightarrow C_2 \rightsquigarrow \upsilon_1 \to \upsilon_2$

From the rule hypothesis it follows that:

$$\Gamma \vdash_{ct} C_1 \textit{ and } \vdash_{ct} C_1 \rightsquigarrow \upsilon_1$$

$$\Gamma \vdash_{ct} C_2 \textit{ and } \vdash_{ct} C_2 \rightsquigarrow \upsilon_2$$

So by the induction hypothesis:

$$\Delta \vdash^{F}_{ty} \upsilon_1$$

$$\Delta \vdash^{F}_{ty} \upsilon_2$$

So by applying F-TyArr:

$$\Delta \vdash^{F}_{ty} \upsilon_1 \to \upsilon_2$$

Lemma 17 (Type Preservation for Polytypes) *If* $\Gamma \vdash_{ty} \sigma$ *and* $\vdash_{ty} \sigma \rightsquigarrow \upsilon$ *and* $\Gamma \rightsquigarrow \Delta$ *then* $\Delta \vdash^{F}_{ty} \upsilon$.

Proof: By structural induction on the type σ and corresponding well-formedness and elaboration rules.

$\boxed{\textbf{TyVar / TyArr}}$ $\sigma = \tau$

The result follows directly from lemma 15.

$\boxed{\textbf{TyQual}}$ $\Gamma \vdash_{\mathtt{ty}} C \Rightarrow \rho$ *and* $\vdash_{\mathtt{ty}} C \Rightarrow \rho \rightsquigarrow \upsilon_1 \to \upsilon_2$

By the rule hypothesis:

$$\Gamma \vdash_{\mathtt{ct}} C \textit{ and } \vdash_{\mathtt{ct}} C \rightsquigarrow \upsilon_1$$

$$\Gamma \vdash_{\mathtt{ty}} \rho \textit{ and } \vdash_{\mathtt{ty}} \rho \rightsquigarrow \upsilon_2$$

So if follows from the induction hypothesis that:

$$\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon_2$$

And it follows from lemma 16:

$$\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon_1$$

So by applying F-TyArr:

$$\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon_1 \to \upsilon_2$$

$\boxed{\textbf{TyAll}}$ $\Gamma \vdash_{\mathtt{ty}} \forall a.\sigma$ *and* $\vdash_{\mathtt{ty}} \forall a.\sigma \rightsquigarrow \forall a.\upsilon$

From the rule hypothesis we know:

$$\Gamma, a \vdash_{\mathtt{ty}} \sigma \textit{ and } \vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon$$

So from the induction hypothesis, it is clear that:

$$\Delta_2 \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$$

Where $\Gamma, a \rightsquigarrow \Delta_2$. Since we know that $\Delta_2 = \Delta, a$, this becomes equivalent to:

$$\Delta, a \vdash^{\mathtt{F}}_{\mathtt{ty}} \upsilon$$

So by F-TyAll it follows that:

$$\Delta \vdash^{\mathtt{F}}_{\mathtt{ty}} \forall a.\upsilon$$

Appendix B

Poster

KATHOLIEKE UNIVERSITEIT LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Master
Computer Science

Master Thesis
Gert-Jan Bottu

Promotor
Prof. Tom Schrijvers

Daily Advisor
George Karachalias

2016-2017

DTAI
Declaratieve Talen en Artificiële Intelligentie

Quantified Class Constraints

Goal

Extend the current Haskell type system with Quantified Class Constraints[1].

Expressiveness

```
class Trans t where
  lift :: Monad m => m a -> t m a


newtype (t1 :*: t2) m a
  = C { runC :: t1 (t2 m) a }

instance (MonadTrans t1, MonadTrans t2)
       => MonadTrans (t1 :*: t2) where
  lift :: C . lift . lift
```

Wanted: Monad (t2 m)

Solution

```
class (forall m. Monad m => Monad (t m))
      => Trans t where
  lift :: Monad m => m a -> t m a

newtype (t1 :*: t2) m a
  = C { runC :: t1 (t2 m) a }

instance (MonadTrans t1, MonadTrans t2)
       => MonadTrans (t1 :*: t2) where
  lift :: C . lift . lift
```

QCCs: Requirement can be made an explicit constraint in the `MonadTrans` class declaration.
Added advantage: Code becomes much more self explanatory: "A monad transformer can lift any monad `m` into a new monad `(t m)`."

Non-Termination

```
data GRose f a = GBranch a (f (GRose f a))

instance (Show a, Show (f (Grose f a)))
         => Show (GRose f a) where
  show (GBranch x xs) = show x ++ " " ++ show xs
```

```
Show (GRose [] Bool)
↦ Show Bool, Show [ GRose [] Bool ]
↦ Show Bool, Show (GRose [] Bool)
↦ ...
```

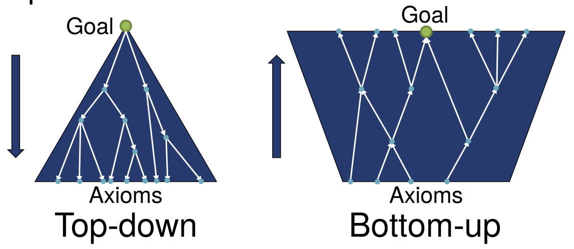
Solution

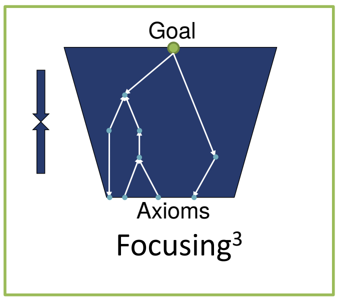
```
data GRose f a = GBranch a (f (GRose f a))

instance (Show a, forall x. Show x => Show (f x))
         => Show (GRose f a) where
  show (GBranch x xs) = show x ++ " " ++ show xs
```

Standard type checker: Diverges on `Show (GRose [] Bool)`.
QCCs: `Show Bool` and `Show (GRose [] Bool)` are derivable from the available context.
⇒ avoids loop.

Approach

- **Curry-Howard correspondence**: Class constraints correspond to logical predicates on types.
- **Quantified Class Constraints**: Correspond to higher order logic, making the language more expressive.
- **Proof search**: Interleaved bottom-up and top-down.





Encountered Issues

Highly non deterministic:

Focusing reduces non determinism, but does not eliminate it, since it involves picking axioms, trying to match the wanted constraint. Different choices might result in a different result.
Solution: Coherence

Termination:

Care has to be taken to guarantee the termination of the proof search.

Coherence:

Multiple ways of matching wanted constraints may arise. Needed to ensure that all possible derivations have the same runtime behavior.

Related Work

[1]: Ralf Hinze and Simon Peyton Jones. Derivable type classes. Electronic Notes in Theoretical Computer Science, 41(1):5-35, 2001.
[2]: Valery Trifonov. Simulating quantified class constraints. Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '03, pages 98-102, 2003.
[3]: Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. Journal of Logic and Computation 2 (1992), 297–347.

Appendix C

Research Article

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.

[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]:

```
class Trans t where
  lift :: Monad m ⇒ m a → (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 *Monad* instance of the form:

```
instance Monad m ⇒ Monad (T m) where . . .
```

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

```
newtype (t₁ ∗ t₂) m a = C { runC :: t₁ (t₂ m) a }

instance (Trans t₁, Trans t₂) ⇒ Trans (t₁ ∗ t₂) where
  lift = C · lift · lift
```

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:

```
class (∀m.Monad m ⇒ Monad (t m)) ⇒ Trans t where
  lift :: Monad m ⇒ m a → (t m) a
```

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*:

```
data Mu h a = In { out :: h (Mu h) a }

data List₂ f a = Nil | Cons a (f a)

type List = Mu List₂
```

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

```
instance (∀f.Functor f ⇒ Functor (h f)) ⇒ Functor (Mu h)
  where fmap f (In x) = In (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 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

```
data Perfect a = Zero a | Succ (Perfect (a, a))
```

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

```
data HPerf f a = HZero a | 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:

```
class (∀f.Functor f ⇒ Functor (h f)) ⇒ HFunctor h where
  hfmap :: (Functor f, Functor g)
        ⇒ (∀x.f x → g x) → (∀x.h f x → 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 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:

```
class HFunctor h where
  ffmap :: Functor f ⇒ (a → b) → (h f a → h f b)
  hfmap :: (Functor f, Functor g)
        ⇒ (∀x.f x → g x) → (∀x.h f x → h g x)
```

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

```
data GRose f a = GBranch a (f (GRose f a))
```

and its *Show* instance

```
instance (Show a, Show (f (GRose f a))) ⇒ Show (GRose f a)
  where show (GBranch x xs) = unwords [show x, "−", show xs]
```

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 (*Show* (*GRose* [] *Bool*)). Indeed, it would recursively resolve the instance context: *Show Bool* is easily dismissed, but *Show* [*GRose* [] *a*] requires resolving *Show* (*GRose* [] *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:

```
instance (Show a, ∀x.Show x ⇒ Show (f x)) ⇒ Show (GRose f a)
  where show (GBranch x xs) = unwords [show x, "−", show xs]
```

This would avoid the diverging loop in the type system extension they sketch, because the two recursive resolvents, *Show Bool* and $\forall x.Show\ x \Rightarrow Show\ [x]$ 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.

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

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 $P,_{L}\,C$ to denote that we extend the local component of the triple, and similar notation

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

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

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

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

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

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

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 ($\Rightarrow$IC) and ($\Rightarrow$EC) correspond to implication introduction and elimination, respectively. Similarly, Rules ($\forall$IC) and ($\forall$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 $P = \langle \bullet, \bullet, \mathit{Eq}\, a \rangle$, here are only two of the infinitely many proofs of $P; \Gamma \models \mathit{Eq}\, a$:

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

versus

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

where $P' = P ,_{\mathrm{L}} \mathit{Eq}\, a$. 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 ($\Rightarrow$R) and ($\forall$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

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

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

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

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

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

Figure 3. Tractable Constraint Entailment

goal and there are no new goals. Rule ($\Rightarrow$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 ($\forall$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:

$$\frac{\dfrac{\mathit{Eq}\, a \in P \qquad \Gamma; [\mathit{Eq}\, a] \models \mathit{Eq}\, a \rightsquigarrow \bullet\;(Q\mathrm{L})}{P; \Gamma \models [\mathit{Eq}\, a]}\;(Q\mathrm{R})}{P; \Gamma \models \mathit{Eq}\, a}$$

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.

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

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 (∀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.

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

The third instance gives rise to the axiom $\forall a.C\ a \Rightarrow D\ Int$. When resolving $D\ Int$ 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 (∀U) processes a quantifier by adding the new type variable to the list of determinable type variables $\overline{a}$. Finally, Rule (⇒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 (⇒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 (⇒I) and the local axioms added during constraint entailment checking in Rule (⇒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.

$\boxed{unamb(C)}$ Unambiguity

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

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

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

Figure 4. Unambiguity

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:

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

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:

E	::=	$\bullet \mid E, \tau_1 \sim \tau_2$	*type equalities*

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

θ	::=	$\bullet \mid \theta \cdot [\tau/a]$	*type substitution*
η	::=	$\bullet \mid \eta \cdot [t/d]$	*evidence substitution*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\mathit{unify}(\overline{a}; E) = \theta_\perp$ 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

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

$$\frac{\begin{array}{c}\Gamma, a \vdash_{\mathtt{ty}} \sigma \qquad \vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon \qquad \Gamma, a \vdash_{\mathtt{ct}} C_i \qquad \vdash_{\mathtt{ct}} C_i \rightsquigarrow \upsilon_i \qquad d, \overline{d}^n \text{ fresh} \qquad fdata = \mathbf{data}\ T_{TC}\ a = K_{TC}\ \overline{\upsilon}^n\ \upsilon \\ fval_1 = \mathbf{let}\ f : (\forall a.T_{TC}\ a \to \upsilon) = \Lambda a.\lambda(d : T_{TC}\ a).proj^{n+1}_{TC}(d) \qquad fval^i_2 = \mathbf{let}\ d_i : (\forall a.T_{TC}\ a \to \upsilon_i) = \Lambda a.\lambda(d : T_{TC}\ a).proj^{i}_{TC}(d)\end{array}}{\Gamma \vdash_{\mathtt{cls}} (\mathbf{class}\ (C_1, \ldots, C_n) \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \}) : [\,\overline{d_i : \forall a.TC\ a \Rightarrow C_i}^n\,]; [f : \forall a.TC\ a \Rightarrow \sigma] \rightsquigarrow fdata; fval_1, \overline{fval_2}^n}\ \textsc{Class}$$

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

$$\frac{\begin{array}{c}\mathbf{class}\ (C'_1, \ldots, C'_m) \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \} \qquad \overline{b} = fv(\tau) \qquad \overline{d}, \overline{d}', d_I \text{ fresh} \qquad \mathcal{P}_I = \mathcal{P},_{\mathrm{L}}\ \overline{d : C} \qquad \Gamma_I = \Gamma, \overline{b} \\ \Gamma_I \vdash_{\mathtt{ct}} C_i \qquad \overline{b}; \mathcal{P}_I,_{\mathrm{L}} (\,d_I : \forall \overline{b}.\overline{C}^n \Rightarrow TC\ \tau); \Gamma_I \vdash_{\mathtt{tm}} e : [\tau/a]\sigma \rightsquigarrow t \qquad \vdash_{\mathtt{ct}} C_i \rightsquigarrow \upsilon_i \qquad \overline{b}; \mathcal{P}_I \models \overline{d' : [\tau/a]C'} \rightsquigarrow \bullet; \eta\end{array}}{\mathcal{P};\Gamma \vdash_{\mathtt{inst}} (\mathbf{instance}\ (C_1, \ldots, C_n) \Rightarrow TC\ \tau\ \mathbf{where}\ \{\ f = e\ \}) : [\,d_I : \forall \overline{b}.\overline{C} \Rightarrow TC\ \tau] \rightsquigarrow \mathbf{let}\ d_I : (\forall \overline{b}.\overline{\upsilon} \to T_{TC}\ \tau) = \Lambda \overline{b}.\lambda\overline{(d : \upsilon)}.K_{TC}\ \tau\ \overline{\eta(d')}\ t}\ \textsc{Instance}$$

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]

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

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

[5]We omit the evidence substitutions for brevity.

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

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

Figure 8. Subsumption Rule

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.

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.

$$\begin{array}{llll}
fpgm & ::= & t \mid fval; fpgm \mid fdata; fpgm & \textit{program} \\
fval & ::= & \mathbf{let}\ x : \upsilon = t & \textit{value binding} \\
fdata & ::= & \mathbf{data}\ T\ a = K\ \overline{\upsilon} & \textit{datatype} \\
\\
t & ::= & x \mid K \mid \lambda(x : \upsilon).t \mid t_1\ t_2 \mid \Lambda a.t \mid t\ \upsilon & \textit{term} \\
 & \mid & \mathbf{let}\ x : \upsilon = t_1\ \mathbf{in}\ t_2 \mid \mathbf{case}\ t_1\ \mathbf{of}\ K\ \overline{x} \rightarrow t_2 & \\
\\
\upsilon & ::= & a \mid \upsilon_1 \rightarrow \upsilon_2 \mid \forall a.\upsilon \mid T\ \upsilon & \textit{type}
\end{array}$$

Figure 9. System F Syntax

This operation is expressed succinctly by relation $\overline{a}; \mathcal{P}; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow t$, 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 $\vdash_{ty} \sigma \rightsquigarrow \upsilon$:

$$\frac{}{\vdash_{ty} a \rightsquigarrow a}\ \textsc{TyVar} \qquad \frac{\vdash_{ty} \tau_1 \rightsquigarrow \upsilon_1 \qquad \vdash_{ty} \tau_2 \rightsquigarrow \upsilon_2}{\vdash_{ty} \tau_1 \rightarrow \tau_2 \rightsquigarrow \upsilon_1 \rightarrow \upsilon_2}\ \textsc{TyArr}$$

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

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 $\vdash_{ct} C \rightsquigarrow \upsilon$ which performs elaboration of constraints:

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

$$\frac{\vdash_{ct} C_1 \rightsquigarrow \upsilon_1 \qquad \vdash_{ct} C_2 \rightsquigarrow \upsilon_2}{\vdash_{ct} C_1 \Rightarrow C_2 \rightsquigarrow \upsilon_1 \rightarrow \upsilon_2}\ (\mathrm{C}\Rightarrow)$$

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 ($\upsilon_1 \rightarrow \upsilon_2$), 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

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

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:

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

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:

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

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

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

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

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

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

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

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:

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

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 $\forall a.C\ (a \to a) \Rightarrow C\ (a \to Int \to Int)$. The head's size 5 is strictly greater than the context constraint's size 3. Yet, if we instantiate a to $(Int \to Int \to Int)$, 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:

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

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

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

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:

$$\begin{array}{l}\textbf{class}\ Monad\_t\ t\ \textbf{where}\\ \quad treturn :: Monad\ m \Rightarrow a \rightarrow t\ m\ a\\ \quad tbind\ \ \ :: Monad\ m \Rightarrow t\ m\ a \rightarrow (a \rightarrow t\ m\ b) \rightarrow t\ m\ b\\ \textbf{class}\ Monad\_t\ t \Rightarrow Trans\ t\ \textbf{where}\\ \quad lift :: Monad\ m \Rightarrow m\ a \rightarrow t\ m\ a\end{array}$$

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:

$$\begin{array}{l}\textbf{class}\ MonadT\ t\ \textbf{where}\\ \quad lift\ \ \ \ \ :: Monad\ m \Rightarrow m\ a \rightarrow t\ m\ a\\ \quad treturn :: Monad\ m \Rightarrow a \rightarrow t\ m\ a\\ \quad tbind\ \ \ :: Monad\ m \Rightarrow t\ m\ a \rightarrow (a \rightarrow t\ m\ b) \rightarrow t\ m\ b\end{array}$$

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.

$$\begin{array}{l}\textbf{instance}\ (Monad\ m, MonadT\ t) \Rightarrow Monad\ (t\ m)\ \textbf{where}\\ \quad return = treturn\\ \quad (>>=)\ \ = tbind\end{array}$$

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:

$$\begin{array}{l}\textbf{class}\ Trans\ t\ \textbf{where}\\ \quad lift\ \ :: Monad\ m \Rightarrow m\ a \rightarrow t\ m\ a\\ \quad mw :: Monad\ m \Rightarrow MonadWitness\ t\ m\end{array}$$

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.

$$\begin{array}{l}\textbf{data}\ MonadWitness\ (t :: (* \rightarrow *) \rightarrow (* \rightarrow *))\ m\ \textbf{where}\\ \quad MW :: Monad\ (t\ m) \Rightarrow MonadWitness\ t\ m\end{array}$$

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.

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

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.

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.

[9]See GHC feature request #5927.

A Additional Judgments

A.1 Well-formedness of Types & Constraints

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

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

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

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 $\Gamma \vdash_{\mathtt{ct}} C$, given by the following rules:

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

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

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

A.2 Program Typing

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

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

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

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

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

A.3 Unification Algorithm

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

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

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 $\overline{a}$. 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 $\mathcal{P}; \Gamma \vdash_{\mathtt{pgm}} pgm : \sigma \rightsquigarrow fpgm$:

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

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

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

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

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

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

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

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

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

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

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

B.2 Well-formedness of Types

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

$$\frac{a \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} a}\ \textsc{TyVar} \qquad \frac{T \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} T}\ \textsc{TyCon} \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} v_1 \qquad \Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} v_2}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} v_1 \to v_2}\ \textsc{TyArr}$$

$$\frac{\Delta, a \vdash^{\mathsf{F}}_{\mathtt{ty}} v}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \forall a.v}\ \textsc{TyAll} \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} v_1 \qquad \Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} v_2}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} v_1\ v_2}\ \textsc{TyApp}$$

B.3 Value Binding Typing

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

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

B.4 Datatype Declaration Typing

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

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

B.5 Program Typing

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

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

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

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

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

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)

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

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

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

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

Appendix D

Research Article (Translated Summary in Dutch)

Gekwantificeerde Klasse Constraints in Haskell

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—Gekwantificeerde klasse constraints zijn meer dan vijftien jaar geleden voorgesteld om de expressiviteit van type classes te verhogen van Horn clausules tot eerste orde logica. Ondanks de grote vraag naar deze uitbreiding, is ze nooit in detail uitgewerkt of geïmplementeerd. Verschillende alternatieve oplossingen en workarounds zijn voorgesteld, maar geen van deze bleek een volwaardig alternatief voor gekwantificeerde klasse constraints.

Dit artikel is de vertaalde samenvatting van de Quantified Class Constraints paper [1], en de bijhorende thesistekst. Het artikel herbekijkt het idee van gekwantificeerde klasse constraints en werkt het uit tot een volwaardig, praktisch ontwerp. We bekijken de voordelen van de extensie op twee vlakken: toegevoegde flexibiliteit en expressiviteit in het uitdrukken van constraints enerzijds en terminatie van type class resolutie anderzijds. We geven zowel een specificatie van het type systeem, als een algoritme voor type-inferentie dat gelijktijdig expressies naar System F vertaalt. Bovendien bespreken we terminatiecondities voor type-inferentie en hebben we een prototype geïmplementeerd.

1 Introductie

Dit artikel is de vertaalde samenvatting van de Quantified Class Constraints paper [1], en de bijhorende thesistekst.

Type classes zijn bedacht door Wadler en Blott [2] als een manier om ad-hoc polymorphisme te introduceren. Ondertussen zijn type classes (of een afgeleide vorm hiervan) doorgebroken in verschillende mainstream programmeertalen, zoals Haskell [2], Coq [3], of in een meer beperkte vorm, Rust [4], [5]. Ondanks dat het gebruik van type classes ondertussen wijdverspreid is, zowel in de academische wereld als in de industrie, en ondanks jaren van intensief onderzoek rond type classes en extensies hiervan, zijn ze niet zonder hun beperkingen.

Tijdens hun Derivable Type Classes [6] onderzoek, hetgeen een mogelijke manier bespreekt om generisch programmeren te introduceren in Haskell, kwamen Hinze en Peyton Jones in 2001 een dergelijke type classes beperking tegen. Ze vonden een set van instantie declaraties die, ondanks redelijk te lijken, niet uitdrukbaar waren in Haskell, aangezien ze het constraint entailment algoritme in een oneindige lus brachten. Hun voorbeeld wordt uitgelegd in Sectie 2.2.1. Als een mogelijke oplossing stelden ze voor om Haskell uit te breiden met polymorfe constraints, die ze "gekwantificeerde klasse constraints" noemden. Het toevoegen van deze constraints zou de expressiviteit van type classes verhogen tot wat feitelijk overeenkomt met eerste orde logica op types. Ze boden verder een schets aan van hoe de eigenlijke extensie eruit zou komen te zien, samen met een specificatie van het type systeem.

Spijtig genoeg hebben geen van beide hun werk rond gekwantificeerde klasse constraints later voortgezet. Het GHC ticket #2893 [1] is in 2008 geopend, om meer flexibele constraints te vragen in de taal, maar is tot op de dag van vandaag nog niet opgelost. Verschillende workarounds (zoals [7]) zijn voorgesteld, en alternatieve coderingen (zoals [8]) bedacht, om het gedrag en toegevoegde expressiviteit van gekwantificeerde klasse constraints te kunnen simuleren, zoals uitgelegd in Sectie 7. Spijtig genoeg bleken geen van allemaal een waardig alternatief voor de eigenlijke uitbreiding van de taal te zijn.

Dit artikel en bijhorende paper / thesis bouwt voort op het idee van Hinze en Peyton Jones om Haskell uit te breiden met gekwantificeerde klasse constraints. De belangrijkste bijdragen van dit werk zijn:

1) We geven een overzicht van de twee belangrijkste redenen om gekwantificeerde klasse constraints te introduceren (Sectie 2): a) ze bieden een meer natuurlijke en krachtigere manier om de specificatie van een klasse uit te drukken, en b) ze resulteren in terminatie voor constraint resolutie voor een grotere groep toepassingen.
2) We bouwen voort op de schets van Hinze en Peyton Jones [6] voor gekwantificeerde klasse constraints en breiden het uit tot een volledige specificatie van het systeem (Sectie 3). We gebruiken hiervoor een techniek, gekend als focussing, die we lenen van Cochis [9], een calculus voor Scala implicits, en passen het aan naar een Haskell setting. We beschrijven twee belangrijke verschillen: a) in tegenstelling tot Cochis, ondersteunen we ook superklasses, en b) we gebruiken een globale set van niet-overlappende instanties (in tegenstelling tot de lokaal gescopedte instanties in de Scala setting van Cochis).
3) We tonen een type-inferentie algoritme, wat dat van Haskell '98 uitbreidt (Sectie 4). Het algorithme vertaalt verder ook expressies naar System F, dmv een dictionary gebaseerde vertaling (Sectie 5).
4) We bespreken de terminatie condities voor een systeem met gekwantificeerde klasse constraints (Sectie 6).
5) Ten slotte bieden we ook een prototype implementatie aan, waarmee de lezer alle voorbeelden in deze paper kan uittesten.

1. https://ghc.haskell.org/trac/ghc/ticket/2893

2 Motivatie

Deze sectie illustreert de toegevoegde expressiviteit van gekwantificeerde klasse constraints, door middel van voorbeelden. We splitsen de voorbeelden op in twee grote categorieën: 1) exacter uitdrukbare vereisten voor klasses, en 2) terminatie voor constraint resolutie voor een grotere groep toepassingen.

2.1 Exacte Specificatie

2.1.1 Monad Transformer

Beschouw de bekende monad transformer klasse [10]:

```
class Trans t where
    lift :: Monad m => m a -> t m a
```

Impliciet wordt aangenomen dat voor eender welk type `T` wat `Trans` instantieert, een `Monad` instantie bestaat van deze vorm:

```
instance Monad m => Monad (T m) where ...
```

Aangezien deze vereiste echter niet expliciet gemaakt kan worden, heeft het type systeem geen toegang tot deze informatie. Beschouw bijvoorbeeld de monad zipper [11], zoals beschreven door Schrijvers et al. Dit code fragment wordt niet aanvaard omwille van deze reden:

```
newtype (t1 * t2) m a = C { runC :: t1 (t2 m) a }

instance (Trans t1, Trans t2) => Trans (t1 * t2)
    where lift = C . lift . lift
```

Het idee achter de code is om monad `m` te `liften` naar `(t2 m)` en dan naar `t1 (t2 m)`. De tweede `lift` is echter enkel geldig als `(t2 m)` een monad is, en het type systeem kan niet nagaan dat dit klopt. Er bestaan workarounds voor dit probleem, zoals uitgelegd in Sectie 7, maar zij resulteren in significant langere en complexe code.

Door gebruik te maken van gekwantificeerde klasse constraints, kan dit voorbeeld eenvoudig uitgedrukt worden:

```
class (forall m . Monad m => Monad (t m)) Trans t
    where lift :: Monad m => m a -> t m a
```

2.1.2 Tweede Orde Functors

Een tweede toepassing kan gevonden worden in het werk van Hinze [12]. Hij stelt parametrische datatypes voor als de fixpoint `Mu` van een tweede orde functor:

```
data Mu h a = In { out :: h (Mu h) a }

data List' f a = Nil | Cons a (f a)
type List = Mu List'
```

Een tweede orde functor is een type constructor die functors naar functors brengt. Dit kan beknopt uitgedrukt worden dmv gekwantificeerde klasse constraints, zoals bijvoorbeeld in de `Functor` instantie voor `Mu`:

```
instance (forall f . Functor f => Functor (h f))
      => Functor (Mu h) where
    fmap f (In x) = In (fmap f x)
```

Hoewel dit Hinze's favoriete manier is om de code te formuleren, merkt hij op dat:

> *Unfortunately, the extension has not been implemented yet. It can be simulated within Haskell 98 [8], but the resulting code is somewhat clumsy.*

Voor een tweede voorbeeld beschouwen het werk van Hinze [13], wat perfect binaire bomen uitdrukt met het volgende geneste datatype [14]:

```
data Perfect a = Zero a | Succ (Perfect (a , a))
```

Dit datatype kan ook uitgedrukt worden als fixpoint `Mu` van de tweede orde functor `HPerf`:

```
data HPerf f a = HZero a | HSucc (f (a , a))
type Perfect = Mu HPerf
```

2.2 Terminatie voor Corecursieve Resolutie

2.2.1 Corecursieve Resolutie

Gekwantificeerde klasse constraints zijn oorspronkelijk voorgesteld door Hinze en Peyton Jones [6]. Zij kwamen een set van instantie declaraties tegen die, ondanks redelijk te lijken, het Haskell type systeem in een oneindige lus brachten. Beschouw bijvoorbeeld hun `GRose` datatype:

```
data GRose f a = GBranch a (f (GRose f a))

instance (Show a, Show (f (GRose f a)))
      => Show (GRose f a) where
    show (GBranch x xs) = show x ++ " - " ++ show xs
```

De twee constraints in de instantie context zijn noodzakelijk voor de twee `show` aanroepen. De standaard type class resolutie divergeert bij het oplossen van de `(Show (GRose [] Bool))` constraint:

```
    Show (GRose [] Bool)
|-> Show Bool, Show [GRose [] Bool]
|-> Show Bool, Show (GRose [] Bool)
|-> ...
```

Om dit probleem op te lossen, stelden Hinze en Peyton Jones voor om de `GRose` instantie uit te drukken met gekwantificeerde klasse constraints:

```
instance (Show a, forall x . Show x => Show (f x))
      => Show (GRose f a) where
    show (GBranch x xs) = show x ++ " - " ++ show xs
```

Beide constraints kunnen nu onmiddellijk opgelost worden mbv de beschikbare instanties voor `Bool` en `[a]`.

2.2.2 Lusdetectie

Na hetzelfde non-terminatie probleem tegen te komen in hun Scrap You Boilerplate [7] werk, implementeerden Lämmel en Peyton Jones een constraint resolutie algoritme met lusdetectie [15]. Dit algoritme detecteert wanneer een van de recursieve constraints overeenkomt met een van zijn voorouders. Wanneer dit gebeurt, sluit het algoritme "de (co)recursieve lus". Spijtig genoeg werkt deze methode enkel voor heel specifieke gevallen, namelijk diegene die in een lus geraken.

2.2.3 Non Cyclische Lus

Beschouw echter de `Show` instantie voor het `HPerf` datatype:

```
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 ++ ")"
```

Het standaard resolutie algoritme divergeert:

```
    Show (Perfect Int)
|-> Show (Mu HPerf Int)
|-> Show (HPerf (Mu HPerf) Int)
|-> Show Int, Show (Mu HPerf (Int,Int))
|-> Show Int, Show (HPerf (Mu HPerf) (Int,Int))
|-> Show Int, Show (Int,Int),
      Show (Mu HPerf ((Int,Int),(Int,Int)))
|-> ...
```

Aangezien er echter nooit eenzelfde constraint terugkomt in het divergerende proces, kan dit probleem niet opgelost worden dmv lusdetectie. Dit kan echter uitgedrukt worden met gekwantificeerde klasse constraints, op een manier waarop de resolutie wel termineert.

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

instance (Show a, forall 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 Besluit

Haskell uitbreiden met gekwantificeerde klasse constraints resulteert in twee aparte groepen van toepassingen die nu uitdrukbaar worden:

1) Gekwantificeerde klasse constraints laten toe om meer van de specificatie van een klasse expliciet te maken. Deze extra informatie laat het type systeem toe om meer toepassingen te aanvaarden.
2) Verder zorgen deze constraints voor een grotere groep toepassingen voor welke terminerende constraint resolutie mogelijk wordt.

Verschillende workarounds voor gekwantificeerde klasse constraints zijn onderzocht, maar geen enkele bleek een volwaardig alternatief te zijn voor de eigenlijke extensie. Gekwantificeerde klasse constraints zijn nog steeds een bijzonder nuttige en veel aangevraagde feature.

3 Type Systeem Specificatie

Deze sectie presenteert de specificatie voor onze core Haskell calculus met gekwantificeerde klasse constraints.

3.1 Syntax

Figuur 1 toont de, grotendeels standaard, syntax voor onze core Haskell taal, met gekwantificeerde klasse constraints. Een programma *pgm* bestaat uit klasse declaraties *cls*, instantie declaraties *inst* en een enkele top level expressie *e*. Om de regels te vereenvoudigen, kiezen we voor klasses met één enkele parameter en één methode.

Een term e bestaat uit de λ-calculus, met recursieve let-bindings. Gelijkaardig aan alle andere Damas-Milner extensies, discrimineren we bij types tussen monotypes τ, qualified types ρ en polytypes σ. Om de representatie eenvoudig te houden, bevat de formalisatie geen hoger kinded types, hoewel deze wel aanwezig zijn in de prototype implementatie.

Het enige punt waar onze formalisatie verschilt van Haskell '98, zijn de constraints. In standaard Haskell '98, bestaan constraints C in type annotaties, klasse en instantie contexts uit simpele klasse constraints Q. De axiomas, afgeleid uit instanties of superklasses, zijn dus Horn clausules van de vorm $\forall\overline{a}.Q_1 \wedge ... \wedge Q_n \Rightarrow Q_0$.

In ons uitgebreid systeem zijn flexibelere constraints C toegelaten waar, naar analogie met hoger orde types, type abstracties en implicaties in geneste posities kunnen plaatsvinden.

De figuur toont ten slotte ook de type omgeving Γ en de programmatheorie P. De eerstgenoemde is volledig standaard. De programmatheorie bestaat uit een triple van 3 axioma sets: de superklasse axiomas A_S, de instantiatie axiomas A_I en de lokale axiomas A_L. De notatie $P,_{\text{L}} C$ wordt gebruikt om aan te duiden dat het axioma C toegevoegd wordt aan de lokale axioma set van de theorie P. De andere twee sets worden analoog uitgebreid. De programmatheorie wordt opgesplitst om restricties op de sets expliciet te kunnen maken, en om te benadrukken dat, in tegenstelling tot Haskell '98 (waar lokale axiomas enkel basis klasse constraints Q bevatten), alle drie de componenten dezelfde constraints ondersteunen.

$$
\begin{array}{lcl}
pgm & ::= & e \mid cls; pgm \mid inst; pgm \\
cls & ::= & \textbf{class}\ A \Rightarrow TC\ a\ \textbf{where}\ \{\ f :: \sigma\ \} \\
inst & ::= & \textbf{instance}\ A \Rightarrow TC\ \tau\ \textbf{where}\ \{\ f = e\ \} \\
\\
e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
\\
\tau & ::= & a \mid \tau_1 \rightarrow \tau_2 \\
\rho & ::= & \tau \mid C \Rightarrow \rho \\
\sigma & ::= & \rho \mid \forall a.\sigma \\
\\
A & ::= & \bullet \mid A, C \\
C & ::= & Q \mid C_1 \Rightarrow C_2 \mid \forall a.C \\
Q & ::= & TC\ \tau \\
\\
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, K : \upsilon \mid \Gamma, T \\
P & ::= & \langle A_S, A_I, A_L \rangle
\end{array}
$$

Fig. 1. Gekwantificeerde klasse constraints syntax

3.2 Type Systeem

Figuur 2 toont de goedgevormdheidsregels voor constraints. Deze relaties voor types zijn volledig standaard en kunnen gevonden worden in de bijhorende paper of thesis.

De typeringsregels voor termen, klasses en instantie declaraties zijn weergegeven in Figuur 3. De regels zijn bijna volledig standaard, op een paar kleine vereenvoudigingen na: 1) We volgen de Barendregt conventie [16] en stellen dat alle variabelen die gebonden worden, uniek zijn. 2) We volgen Vytiniotis et al. [17] en kozen voor recursieve let-bindings, die niet gegeneraliseerd zijn. Buiten dit, is het enige interessante verschil de constraint entailment relatie $P; \Gamma \models C$, die gebruikt wordt in de ($\Rightarrow$E) regel. Deze relatie wordt in detail uitgelegd in Sectie 3.3. Meer uitleg rond de typeringsrelatie voor termen, klasses en instantie declaraties kan gevonden worden in de bijhorende paper of thesis.

$\boxed{\Gamma \vdash_{ct} C}$ Constraint Goedgevormdheid

$$\frac{\Gamma \vdash_{ty} \tau}{\Gamma \vdash_{ct} TC\ \tau}\ \mathrm{C}Q \qquad \frac{\Gamma \vdash_{ct} C_1 \qquad \Gamma \vdash_{ct} C_2}{\Gamma \vdash_{ct} C_1 \Rightarrow C_2}\ (\mathrm{C}\Rightarrow)$$

$$\frac{a \notin \Gamma \qquad \Gamma, a \vdash_{ct} C}{\Gamma \vdash_{ct} \forall a.C}\ (\mathrm{C}\forall)$$

Fig. 2. Goedgevormheid van Constraints

3.3 Constraint Entailment

We presenteren de constraint entailment in twee stappen. Eerst tonen we een eenvoudig te begrijpen, maar bijzonder ambigue specificatie. Vervolgens presenteren we een syntaxgerichte, semi-algoritmische variant, die een deel van de amibiguïteit verhelpt. Deze tweede specificatie is gebaseerd op de focusing techniek [18].

3.3.1 Declaratieve Specificatie

De hoog-niveau, declaratieve constraint entailment specificatie wordt getoond in Figuur 4. Indien we de constraints C interpreteren als logische formules, komen deze regels perfect overeen met de eerste orde predicaten logica regels. Dit zijn de regels die voorgesteld werden door Hinze en Peyton Jones [6]. Hoewel de regels erg compact en eenvoudig te begrijpen zijn, zijn ze ook bijzonder ambigu. Oneindig veel verschillende (geldige) bewijzen kunnen bestaan voor eenzelfde constraint. Een voorbeeld hiervan wordt gegeven in de bijhorende paper en thesis.

3.3.2 Focusing-gebaseerde Specificatie

Om een groot deel van de ambiguïteit te vermijden, introduceren we een tweede, focusing-gebaseerde specificatie. Dit idee werd eerder al toegepast in de Cochis calculus [9], voor dezelfde reden. Het idee achter focusing is om constraint entailment syntax-gericht te maken, zodat op eender welk moment telkens maar één enkele regel van toepassing is. Deze specificatie wordt getoond in Figuur 5. De hoofdrelatie $P; \Gamma \models C$ is nu gedefinieerd aan de hand van twee hulprelaties: $P; \Gamma \models [C]$ en $\Gamma; [C] \models Q \rightsquigarrow A$. Beide zijn syntax-gericht op de constraint C tussen vierkante haakjes.

De hoofdrelatie is equivalent aan de eerste hulprelatie $P; \Gamma \models [C]$, die focust op de te bewijzen constraint C (dewelke we vanaf nu aanduiden met "het doel"). Er zijn drie verschillende regels, een voor elk van de drie mogelijke syntactische vormen van C. De ($\Rightarrow$R) en ($\forall$R) regels ontbinden het doel door respectievelijk implicaties en type abstracties weg te strippen. Wanneer het doel uiteindelijk een basis klasse constraint Q is, selecteert Regel (QR) een axioma C uit de programmatheory P om met het doel te matchen.

Dit matchen wordt behandeld door de tweede hulprelatie $\Gamma; [C] \models Q \rightsquigarrow A$, die checkt of C matcht met het doel Q. Deze relatie resulteert verder in een axioma set A van resterende (vereenvoudigde) constraints, die recursief verder gecheckt moeten worden. Opnieuw bestaan er drie verschillende regels, voor de mogelijke syntactische vormen van C. Regel (QL) is het basis geval waarin het axioma identiek is aan het doel. Regel ($\Rightarrow$L) behandelt het implicatie geval $C_1 \Rightarrow C_2$ door recursief C_2 te matchen met het doel, en C_1 toe te voegen aan de resterende axiomas. Tot slot behandelt Regel ($\forall$L) het universele quantificatie geval, door de type variabele te instantiëren met een monotype τ, zodat het axioma recursief matcht met het doel.

Het is dus eenvoudig om in te zien dat deze specificatie het aantal mogelijke bewijzen voor eenzelfde constraint dramatisch vermindert.

3.4 Resterend Nondeterminisme

Ondanks dat de constraint entailment specificatie door focusing nu type-gericht is, bestaan er nog steeds twee oorzaken van nondeterminisme. De specificatie is dus nog steeds ambigu.

3.4.1 Overlappende Axiomas

De eerste bron van nondeterminisme is de (QR) regel. De programmatheorie kan verschillende axiomas bevatten die de entailment doen slagen. Aangezien de proofs in Haskell methode implementaties bevat, kan dit leiden tot verschillend gedrag tijdens het uitvoeren. We lossen dit op door, net als in Haskell '98, te eisen dat er geen overlappende instanties bestaan.

Door superklasses kunnen er nog steeds verschillende bewijzen bestaan voor eenzelfde constraint. Maar doordat alle methode implementaties uit de instantie declaraties komen, en er geen overlappende instanties bestaan, kan dit nooit leiden tot verschillend gedrag tijdens het uitvoeren.

3.4.2 Polymorfe Instantiatie

De tweede bron van nondeterminisme is de ($\forall$L) regel, aangezien deze een type τ "kiest" om b mee te instantiëren. Er zijn echter oneindig veel mogelijke, goedgevormde monotypes om uit te kiezen. Deze keuze kan zowel het gedrag van de entailment, als het resulterende uitvoer gedrag beïnvloeden. Een "fout" type kiezen, kan ertoe leiden dat het entailen later faalt, wat de entailment nondeterministisch maakt. Bovendien kunnen er meerdere types bestaan waarvoor de entailment slaagt, maar waarvoor die leiden tot andere resulterende methode implementaties. Een voorbeeld hiervan kan gevonden worden in de bijhorende paper en thesis.

Haskell '98 lost dit probleem op door te vereisen dat alle gebonden type variabelen voorkomen in het hoofd van de constraint. Aangezien onze constraints flexibeler zijn en geneste constraints toelaten, moet deze vereiste veralgemeend worden. Dit wordt weergegeven in de *unamb*(C) relatie in Figuur 6. We vereisen dat elk axioma dat aan een programmatheorie toegevoegd wordt, ondubbelzinnig is.

4 Type-Inferentie

We bieden tevens een type-inferentie algoritme aan, met elaboratie van programma's naar System F. Deze sectie focust enkel op type-inferentie, terwijl Sectie 5 het elaboratie-aspect (gemarkeerd in grijs in de figuren) behandelt.

Aangezien type-inferentie en elaboratie naar System F gelijktijdig gebeuren, markeren we alle constraints met de bijhorende System F bewijs term, of dictionary variabele d. Dit wordt aangeduid met kalligrafische letters.

Zoals alle Hindley-Milner gebaseerde systemen, bestaat het type-inferentie proces uit twee aparte fases: 1) Eerst worden twee sets van constraints gegenereerd: geannoteerde constraints $\mathcal{A}$ en type gelijkheden E, van de vorm:

$$E \ ::= \ \bullet \mid E, \tau_1 \sim \tau_2 \qquad \textit{type gelijkheden}$$

2) Vervolgens worden deze constraints opgelost, resulterende in type substituties θ en dictionary substituties η:

$$\begin{aligned} \theta \ &::= \ \bullet \mid \theta \cdot [\tau/a] && \textit{type substitutie} \\ \eta \ &::= \ \bullet \mid \eta \cdot [t/d] && \textit{dictionary substitutie} \end{aligned}$$

$\boxed{P; \Gamma \vdash_{\mathsf{tm}} e : \sigma}$ Term Typering

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

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

$\boxed{\Gamma \vdash_{\mathsf{cls}} cls : A_S ; \Gamma_c}$ Klasse Declaratie Typering

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

$\boxed{P; \Gamma \vdash_{\mathsf{inst}} inst : A_I}$ Klasse Instantie Typering

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

Fig. 3. Type Systeem Specificatie

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

$$\frac{C \in P}{P; \Gamma \models C}\ (\textsc{SpecC}) \qquad \frac{P; \Gamma, a \models C}{P; \Gamma \models \forall a.C}\ (\forall\mathrm{IC}) \qquad \frac{\begin{array}{c}P; \Gamma \models \forall a.C \\ \Gamma \vdash_{\mathsf{ty}} \tau\end{array}}{P; \Gamma \models [\tau/a]C}\ (\forall\mathrm{EC})$$

$$\frac{P,_{\mathrm{L}} C_1; \Gamma \models C_2}{P; \Gamma \models C_1 \Rightarrow C_2}\ (\Rightarrow\mathrm{IC}) \qquad \frac{\begin{array}{c}P; \Gamma \models C_1 \Rightarrow C_2 \\ P; \Gamma \models C_1\end{array}}{P; \Gamma \models C_2}\ (\Rightarrow\mathrm{EC})$$

Fig. 4. Declaratieve Constraint Entailment Specificatie

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

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

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

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

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

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

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

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

Fig. 5. Focussing-gebaseerde Constraint Entailment Specificatie

$\boxed{unamb(C)}$ Ondubbelzinnigheid

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

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

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

Fig. 6. Ondubbelzinnigheid

4.1 Constraints Genereren

De constraint generatie voor termen wordt getoond in Figuur 7. De relatie krijgt de vorm $\Gamma \vdash_{\mathsf{tm}} e : \tau \rightsquigarrow t \mid \mathcal{A}; E$ en genereert dus een type t, een set van benodigde constraints $\mathcal{A}$ en een set gelijkheden E. De regels zijn standaard en meer uitleg kan gevonden worden in de bijhorende paper en thesis.

Het type checken van klasse en instantie declaraties is gepresenteerd in Figuur 8. Beide zijn grotendeels gelijk aan de specificatie van Figuur 3. Het meest noemenswaardige verschil is de manier waarop de methode implementatie e behandeld wordt in de instantie. Aangezien de methode een type opgelegd krijgt vanuit de klasse declaratie, moet het type niet afgeleid, maar gecheckt worden. Dit gebeurt via de subsumptie regel van Figuur 9. De regel checkt dat het type τ_1, door type-inferentie, ondergebracht is in het opgelegde σ type. Dit betekent dat als een term type τ_1 heeft, het automatisch ook type σ toegekend kan worden.

Het type checken van programma's is eenvoudig en kan gevonden worden in de bijhorende paper of thesistekst.

4.2 Constraints Oplossen

4.2.1 Gelijkheden Oplossen

Gelijkheden worden opgelost dmv unificatie: $unify(\overline{a}; E) = \theta_{\perp}$. De functie neemt een set van "onaantastbare" variabelen (alle andere type variabelen in de type signaturen zijn unificeerbaar) en een

$\boxed{\Gamma \vdash_{\texttt{tm}} e : \tau \leadsto t \mid \mathcal{A}; E}$ Term Typering

$$\frac{\overline{b}, \overline{d} \text{ fresh} \qquad (x : \forall \overline{a}. \overline{C} \Rightarrow \tau) \in \Gamma}{\Gamma \vdash_{\texttt{tm}} x : [\overline{b}/\overline{a}]\tau \leadsto x\ \overline{b}\ \overline{d} \mid \overline{(d : [\overline{b}/\overline{a}]C)}; \bullet} \textsc{TmVar}
\qquad
\frac{\begin{array}{c} a \text{ fresh} \qquad \Gamma, x : a \vdash_{\texttt{tm}} e_1 : \tau_1 \leadsto t_1 \mid \mathcal{A}_1; E_1 \\ \Gamma, x : \tau_1 \vdash_{\texttt{tm}} e_2 : \tau_2 \leadsto t_2 \mid \mathcal{A}_2; E_2 \qquad \vdash_{\texttt{ty}} \tau_1 \leadsto \upsilon_1 \end{array}}{\Gamma \vdash_{\texttt{tm}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2 \leadsto \textbf{let}\ x : \upsilon_1 = t_1\ \textbf{in}\ t_2 \mid (\mathcal{A}_1, \mathcal{A}_2); (E_1, E_2, a \sim \tau_1)} \textsc{TmLet}$$

$$\frac{a \text{ fresh} \qquad \Gamma, x : a \vdash_{\texttt{tm}} e : \tau \leadsto t \mid \mathcal{A}; E}{\Gamma \vdash_{\texttt{tm}} \lambda x.e : a \to \tau \leadsto \lambda(x : a).t \mid \mathcal{A}; E} \textsc{TmAbs}
\qquad
\frac{a \text{ fresh} \qquad \Gamma \vdash_{\texttt{tm}} e_1 : \tau_1 \leadsto t_1 \mid \mathcal{A}_1; E_1 \qquad \Gamma \vdash_{\texttt{tm}} e_2 : \tau_2 \leadsto t_2 \mid \mathcal{A}_2; E_2}{\Gamma \vdash_{\texttt{tm}} e_1\ e_2 : a \leadsto t_1\ t_2 \mid \mathcal{A}_1, \mathcal{A}_2; E_1, E_2, \tau_1 \sim \tau_2 \to a} \textsc{TmApp}$$

Fig. 7. Contraint Generatie voor Termen (incl Elaboratie)

$\boxed{\Gamma \vdash_{\texttt{cls}} cls : \mathcal{A}_S; \Gamma_c \leadsto fdata; \overline{fval}}$ Klasse Declaratie Typering

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

$\boxed{\mathcal{P}; \Gamma \vdash_{\texttt{inst}} inst : \mathcal{A}_I \leadsto fval}$ Klasse Instantie Typering

$$\frac{\begin{array}{c} \textbf{class}\ (C'_1, \ldots, C'_m) \Rightarrow TC\ a\ \textbf{where}\ \{\ f : \sigma\ \} \\ \overline{b} = fv(\tau) \qquad \overline{d}, \overline{d'}, d_I \text{ fresh} \qquad \mathcal{P}_I = \mathcal{P}, _{\text{L}} \overline{d : C}, _{\text{L}} (d_I : \forall \overline{b}. \overline{C}^n \Rightarrow TC\ \tau) \qquad \Gamma_I = \Gamma, \overline{b} \\ \Gamma_I \vdash_{\texttt{ct}} C_i \qquad \overline{b}; \mathcal{P}_I; \Gamma_I \vdash_{\texttt{tm}} e : [\tau/a]\sigma \leadsto t \qquad \vdash_{\texttt{ct}} C_i \leadsto \upsilon_i \qquad \overline{b}; \mathcal{P}_I \models \overline{d' : [\tau/a]C'} \leadsto \bullet; \eta \end{array}}{\mathcal{P}; \Gamma \vdash_{\texttt{inst}} (\textbf{instance}\ (C_1, \ldots, C_n) \Rightarrow TC\ \tau\ \textbf{where}\ \{\ f = e\ \}) : [d_I : \forall \overline{b}. \overline{C} \Rightarrow TC\ \tau] \leadsto \textbf{let}\ d_I : (\forall \overline{b}. \overline{\upsilon} \to T_{TC}\ \tau) = \Lambda \overline{b}. \lambda \overline{(d : \upsilon)}. K_{TC}\ \tau\ \overline{\eta(d')}\ t} \textsc{Instance}$$

Fig. 8. Declaratie Elaboratie

$\boxed{\overline{a}; \mathcal{P}; \Gamma \vdash_{\texttt{tm}} e : \sigma \leadsto t}$ Expliciet Geannoteerde Term Checking

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

Fig. 9. Subsumptie

set van gelijkheden, en genereert de meest algemene unificerende substitutie θ, of faalt indien deze niet bestaat. De definitie is standaard en kan teruggevonden worden in de bijhorende paper of thesis.

4.2.2 *Type Class Constraints Oplossen*

Het op focusing gebaseerde algoritme om constraints op te lossen, is te vinden in Figuur 10, en bestaat naar analogie met de specificatie in Figuur 5 uit een hoofdrelatie en twee hulprelaties. De hoofdrelatie $\overline{a}; \mathcal{P} \models \mathcal{A}_1 \leadsto \mathcal{A}_2; \eta$ vereenvoudigt alle benodigde constraints uit $\mathcal{A}_1$ naar resterende constraints $\mathcal{A}_2$. Dit gebeurt via de eerste hulprelatie. De reden dat we niet vereisen dat constraints volledig entailed zijn, is omdat op deze manier top level signaturen vereenvoudigd kunnen worden via deze relatie. In de instantie declaratie vereisen we daarentegen wel dat alle superklasse constraints en constraints van de methode type signatuur volledig entailed zijn.

De eerste hulprelatie $\overline{a}; \mathcal{P} \models [C] \leadsto \mathcal{A}; \eta$ vereenvoudigt constraint C naar een set van eenvoudigere constraints, zonder de onaantastbare variabelen $\overline{a}$ te instantiëren. De relatie is opnieuw syntax-gericht op de constraint C, en bestaat dus uit 3 regels, voor de verschillende syntactische vormen van C. Regels ($\Rightarrow$R) en ($\forall$R) vereenvoudigen recursief het hoofd van het doel. Wanneer het doel uiteindelijk een basis klasse constraint Q is, selecteert (QR) een matchend axioma uit de programmatheorie en vervangt het doel met de vereenvoudigde constraints $\mathcal{A}$. Matching gebeurt via de tweede hulprelatie.

De matching relatie $\overline{a}; [C] \models Q \leadsto \mathcal{A}; \theta; \eta$ focust op het axioma C en checkt of deze het doel Q kan matchen. Het belangrijkste verschil met de specificatie is dat ($\forall$L) nu geen types meer moet "kiezen" om mee te instantiëren, maar dat deze bepaald worden door de θ substitutie. Deze substitutie wordt in de (QL) regel gecreëerd. De gebonden b variabelen worden dus feitelijk unificatie variabelen.

5 Vertaling naar System F

Deze sectie bespreekt het elaboratie-aspect van het algoritme uit Sectie 4. De vertaling gebeurt naar System F, uitgebreid met data types en recursieve let-bindings. Net als in onze Haskell subset, tonen we geen kinds voor System F. Verder vereenvoudigen we de taal door enkel data types met één type parameter en één data constructor toe te laten en enkel case expressies met één tak. Deze taal is volledig standaard en de syntax-, typerings- en uitvoeringsregels kunnen gevonden worden in de bijhorende paper en thesis.

Voor de elaboratie van type classes, gebruiken we de klassieke benadering, een dictionary gebaseerde vertaling. Type class constraints worden dus vertaald naar expliciete System F termen, ook wel dictionaries genoemd. Deze dienen als bewijs dat aan de constraint voldaan is, en bevatten de implementaties van de methodes van de bijhorende klasse.

De elaboratie van constraints wordt uitgelegd in Figuur 11. Enkel de ($\Rightarrow$C) regel is noemenswaardig, aangezien het constraints van de vorm $C_1 \Rightarrow C_2$ vertaalt naar dictionary transformers (System F functie types van de vorm $\upsilon_1 \to \upsilon_2$). De elaboratie van

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

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

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

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

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

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

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

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

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

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

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

Fig. 10. Constraint Entailment met Dictionary Constructie

termen wordt getoond in Figuur 7. Een meer gedetailleerde uitleg van deze regels kan gevonden worden in de bijhorende paper of thesistekst.

$\boxed{\vdash_{\mathsf{ct}} C \rightsquigarrow \upsilon}$ Constraint Elaboratie

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

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

Fig. 11. Elaboratie van Constraints

5.1 Dictionary Constructie

Het constraint entailment algoritme van Figuur 10 construeert expliciete bewijzen voor constraints, in de vorm van dictionary substituties.

De dictionary substitutie η in de vereenvoudigingsrelatie toont aan hoe een bewijs voor het doel C geconstrueerd kan worden van de programmatheory $\mathcal{P}$ en de resterende constraints $\mathcal{A}$.

Gelijkaardig hieraan toont de dictionary substitutie η in de matching relatie hoe een bewijs voor het doel Q geconstrueerd kan worden aan de hand van de programmatheorie $\mathcal{P}$ en de vereenvoudigde constraints $\mathcal{A}$. Meer details over hoe deze substituties opgebouwd worden, kunnen gevonden worden in de bijhorende paper en thesistekst.

5.2 Declaratie Elaboratie

De vertaling van klasse en instantie declaraties wordt getoond in Figuur 8. Een klasse wordt vertaald naar een nieuw System F dictionary type, met een enkele data constructor K_{TC}, n argumenten voor de verschillende superklasse dictionaries en 1 argument voor de methode implementatie. Vervolgens worden tevens $n + 1$ projectie functies gegenereerd die de n superklasse dictionaries en de methode hieruit kunnen extraheren.

Een instantie wordt vertaald naar een dictionary die, gegeven de dictionaries voor zijn context constraints, de dictionary data constructor van de bijhorende klasse volledig instantieert.

6 Terminatie van Constraint Resolutie

Aangezien onze extensie zich onderscheidt van Haskell '98 in zijn constraint entailment, is dit bepalend voor de terminatie van het type-inferentie algoritme. Deze sectie bespreekt de nodige condities onder welke terminatie houdt. Deze condities zijn aangepaste versies van de terminatie condities in Cochis [9].

We bespreken terminatie door het resolutieproces voor te stellen als een boom, waarin de knooppunten overeenkomen met doel constraints. Een zijde tussen een ouder node en kind node, komt overeen met het matchen van een axioma met de ouder node, waarvan de kind nodes de resterende constraints zijn. Indien resolutie zou divergeren, zou er dus een oneindig pad moeten bestaan in de boom, die een oneindige sequentie van axioma applicaties uitdrukt.

Om aan te duiden dat een dergelijk pad niet kan bestaan, gebruiken we de norm $\|\bullet\|$, die de lengte van het hoofd van een constraint uitdrukt, alsvolgt:

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

We willen dus aantonen dat de norm bij elke zijde strikt afneemt. Aangezien de norm altijd een natuurlijk getal is, weten we dan dat er geen oneindig pad kan bestaan en het resolutieproces termineert.

6.1 Terminatie Conditie

De conditie dat de norm bij elke stap strikt afneemt voor elk axioma, wordt alsvolgt uitgedrukt:

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

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

De $(\Rightarrow\mathrm{T})$ regel drukt uit dat de norm van het hoofd van de resterende constraint C_1 altijd kleiner moet zijn dan die van het hoofd van C_2. Bovendien moet deze conditie blijven gelden onder type substitutie. Dit wordt bereikt door te eisen dat elke type variabele (die dus geïnstantieerd kan worden door de substitutie) maar maximaal zoveel mag voorkomen in C_1 als in C_2.

6.2 Superklasse Conditie

Spijtig genoeg is deze conditie te sterk voor superklasse axiomas. Bijvoorbeeld voor het superklasse axioma `forall a . Ord a => Eq a` is de norm van zowel `Ord a` als `Eq a` 1, waardoor het dus niet zou voldoen aan de terminatie conditie.

Voor superklasse constraints stellen we dus een alternatieve conditie voor, namelijk een die enkel vereist dat de norm niet-strikt afneemt. We voegen dan de extra conditie toe dat de superklasse relatie een "directed acylcic graph" (DAG) vormt. We weten verder dat er slechts een eindig aantal klasses bestaan.

Elke ononderbroken sequentie van superklasse axioma applicaties in de boom, moet dus eindig zijn (aangezien er slechts een eindig aantal klasses bestaan en doordat de superklasse relatie een DAG vormt, kunnen er geen herhalingen optreden). Voor de lengte van deze sequentie kan de norm van het doel dus constant blijven of afnemen. Na het einde van deze sequentie (dus na een eindig aantal stappen) moet het pad eindigen of neemt de norm strikt af, door het toepassen van een instantie of lokaal axioma. We weten dus dat de norm altijd na een eindig aantal stappen afneemt en het resolutieproces dus termineert.

7 Gerelateerd Werk

Deze sectie bestudeert gerelateerd werk. We focussen vooral op het vergelijken van onze extensie met bestaande encoderingen en workarounds voor gekwantificeerde constraints.

7.1 De Coq Stelling Bewijzer

Coq ondersteunt willekeurige formules in klasse en instantie contexten. Dit omvat dus tevens een vorm van gekwantificeerde constraints. Het nadeel van deze flixibiliteit is dat de resolutie ambigu en nonterminerend is. Coq lost dit op door, indien nodig, de gebruiker te vragen resolutie manueel op te lossen. Deze interactie zou echter onaanvaardbaar zijn in Haskells type-inferentie proces.

7.2 Trifonovs Workaround

Trifonov [8] bestudeerde een alternatieve encodering om gekwantificeerde klasse constraints te kunnen simuleren met enkel Haskell '98 syntax. Het achterliggende idee draait rond het introduceren van nieuwe klasses, om de gekwantificeerde constraint uit te kunnen drukken. Een uitgewerkt voorbeeld hiervan kan gevonden worden in haar paper, of in de bijhorende paper of thesis tekst. Deze encodering kan echter niet alle toepassingen waarvoor gekwantificeerde klasse constraints aanvraagd worden, uitdrukken. Tevens kunnen de methodes uit de nieuw geïntroduceerde klasses niet onder dezelfde naam aangeroepen worden als de bekende methodes die ze moeten uitbreiden.

Voor deze reden bespreekt Trifonov verder nog een tweede encodering, die o.a. door de Monatron library [19] overgenomen werd. Het verschil is dat de tweede encodering steunt op een aantal wijdverspreide uitbreidingen van Haskell '98. In ruil hiervoor dienen minder nieuwe klasses geïntroduceerd te worden, en kunnen de gebruikelijke methode namen gebruikt worden. Spijtig genoeg werkt ook deze alternatieve encodering niet met alle toepassingen waarvoor gekwantificeerde klasse constraints aangevraagd worden. Bovendien maken de extra toegevoegde klasses en instanties de code aanzienlijk langer, minder leesbaar en overlappen ze vaak met andere instanties.

7.3 MonadZipper

Als alternatief voor Trifonovs encodering, bespreken Schrijvers en Oliveira [11] een andere workaround voor gekwantificeerde klasse constraints, in de context van hun monad zipper datatype, een uitgebreide vorm van monad transformer compositie. Hun workaround komt erop neer dat een extra `witness` methode toegevoegd wordt, die een `Witness` GADT teruggeeft. Deze bevat de missende informatie voor het type systeem, die alternatief via gekwantificeerde klasse constraints uitgedrukt zou kunnen worden. Dit is mogelijk aangezien GADTs lokale constraints kunnen bevatten. Door expliciet te pattern matchen tegen deze `Witness`, kan dit axioma expliciet gemaakt worden voor het type systeem. Het nadeel van deze methode is dat een deel van het werk van het type systeem nu naar de programmeur gaat, waardoor de code significant langer en meer complex wordt. Een uitgewerkt voorbeeld kan gevonden worden in hun werk, of in de bijhorende paper of thesis bij dit artikel.

7.4 Cochis

De "calculus of coherent implicits", Cochis [9] is een van de belangrijkste bronnen van inspiratie voor dit artikel en bijhorende paper / thesis. Cochis ondersteunt, net als dit werk, recursieve resolutie van gekwantificeerde constraints, in een Scala omgeving. Er zijn echter twee belangrijke verschillen: 1) Doordat Cochis zit situeert in een Scala omgeving, wordt er geen onderscheid gemaakt tussen type class constraints en klassieke termen. Hierdoor wordt er dus ook geen onderscheid gemaakt tussen instantie en superklasse axiomas. 2) Bovendien ondersteunt Cochis lokale instanties, dit in tegenstelling tot onze globaal gescopedte instanties. Lokale instanties mogen overlappen, waarbij de binnenste instantie altijd prioriteit krijgt. Hierdoor is de resolutie van Cochis volledig deterministisch. Dit in tegenstelling tot die in ons werk, die nondeterministisch, maar coherent is, door overlappende lokale en superklasse axiomas.

8 Conclusie

Dit artikel (en bijhorende paper en thesis) presenteerden een volledig uitgewerkt ontwerp van gekwantificeerde klasse constraints. We hebben aangetoond dat deze feature de expressiviteit van type classes gevoelig uitbreidt, en anderzijds terminerende resolutie mogelijk maakt voor een grotere groep applicaties.

Interessant toekomstig werk om verder te onderzoeken omvat: 1) De metatheorie achter de uitbreiding onderzoeken. 2) Het systeem verder uitbreiden met abstractie over predikaten [2], en op deze manier de expressiviteit dus verder uitbreiden tot (een fractie van) tweede orde logica. 3) Onderzoeken hoe deze extensie geïntegreerd kan worden in GHC's OutsideIn type-inferentie [20], en hoe ze samenwerkt met verschillende mainstream type systeem features in GHC (zoals functionele afhankelijkheden [21]). Indien de extensie compatibel zou blijken, kunnen we een implementatie creëren van gekwantificeerde klasse constraints in GHC.

References

[1] G.-J. Bottu, G. Karachalias, T. Schrijvers, B. C. d. S. Oliveira, and P. Wadler, "Quantified class constraints," in *Proceedings of the 10th International Symposium on Haskell*, 2017, submitted. [Online]. Available: https://people.cs.kuleuven.be/~george.karachalias/papers/quantcs.pdf

2. https://ghc.haskell.org/trac/ghc/ticket/5927

[2] P. Wadler and S. Blott, “How to make ad-hoc polymorphism less ad hoc,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’89. New York, NY, USA: ACM, 1989, pp. 60–76.

[3] M. Sozeau and N. Oury, “First-class type classes,” in *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008. Proceedings*, ser. LNCS, vol. 5170. Springer, 2008, pp. 278–293.

[4] N. Shärli, S. Ducasse, O. Nierstrasz, and A. Black, “Traits: Composable units of behavior,” Tech. Rep., 2002.

[5] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black, “Traits: A mechanism for fine-grained reuse,” *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 2, pp. 331–388, Mar. 2006.

[6] R. Hinze and S. Peyton Jones, “Derivable type classes,” in *Proceedings of the Fourth Haskell Workshop*. Elsevier Science, 2000, pp. 227–236.

[7] R. Lämmel and S. Peyton Jones, “Scrap your boilerplate with class: Extensible generic functions,” *SIGPLAN Not.*, vol. 40, no. 9, pp. 204–215, Sep. 2005.

[8] V. Trifonov, “Simulating quantified class constraints,” in *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, ser. Haskell ’03. New York, NY, USA: ACM, 2003, pp. 98–102.

[9] T. Schrijvers, B. C. d. S. Oliveira, and P. Wadler, “Cochis: Deterministic and coherent implicits,” KU Leuven, Department of Computer Science, Report CW 705, May 2017.

[10] M. P. Jones, “Functional programming with overloading and higher-order polymorphism,” in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK, UK: Springer-Verlag, 1995, pp. 97–136.

[11] T. Schrijvers and B. C. Oliveira, “Monads, zippers and views: Virtualizing the monad stack,” *SIGPLAN Not.*, vol. 46, no. 9, pp. 32–44, Sep. 2011.

[12] R. Hinze, “Adjoint folds and unfolds: Or: Scything through the thicket of morphisms,” in *Proceedings of the 10th International Conference on Mathematics of Program Construction*, ser. MPC’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 195–228.

[13] ——, “Perfect trees and bit-reversal permutations,” *J. Funct. Program.*, vol. 10, no. 3, pp. 305–317, 2000.

[14] R. S. Bird and L. G. L. T. Meertens, “Nested datatypes,” in *Proceedings of the Mathematics of Program Construction*, ser. MPC ’98. London, UK: Springer-Verlag, 1998, pp. 52–67.

[15] R. Lämmel and S. Peyton Jones, “Scrap your boilerplate: A practical design pattern for generic programming,” *SIGPLAN Not.*, vol. 38, no. 3, pp. 26–37, Jan. 2003.

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

[17] D. Vytiniotis, S. Peyton Jones, and T. Schrijvers, “Let should not be generalized,” in *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, ser. TLDI ’10. NY, USA: ACM, 2010, pp. 39–50.

[18] F. Pfenning, “Lecture notes on focusing,” 2010, https://www.cs.cmu.edu/~fp/courses/oregon-m10/04-focusing.pdf.

[19] M. Jaskelioff, “Monatron: an extensible monad transformer library,” in *Proceedings of the 20th international conference on Implementation and application of functional languages*, ser. IFL’08. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 233–248.

[20] D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann, “Outsidein(x): Modular type inference with local assumptions,” *Journal of Functional Programming*, vol. 21, pp. 333–412, September 2011.

[21] M. P. Jones, “Type classes with functional dependencies,” in *Programming Languages and Systems*, ser. LNCS, G. Smolka, Ed. Springer Berlin Heidelberg, 2000, vol. 1782, pp. 230–244.

Master thesis filing card

Student: Gert-Jan Bottu

Title: Quantified Class Constraints in Haskell

Dutch title: Gekwantificeerde Klasse Constraints in Haskell

UDC: 681.3

Abstract:
Type classes have become one of the corner stones of the Haskell language, alongside several other (functional) programming languages. However, despite it having become omnipresent in functional programming, almost three decades of intensive research having gone into it, and countless extensions having been build on top of it, they are not without their limitations. While working on their Derivable Type Classes paper, Hinze and Peyton Jones encountered a set of, albeit perfectly reasonable instance declarations, which surprisingly could not be implemented in Haskell at the time. They proposed to extend the language with polymorphic constraints, which they called "quantified class constraints". These constraints would essentially boost the expressive power of type classes to first-order logic on types. They provided only a sketch of what the extension would look like, as well as a specification for typing.
Several workarounds and alternative encodings for adding the expressiveness, offered by quantified class constraints, without actually extending the Haskell type system, have been investigated. Unfortunately, none of them proved to be a worthy replacement of the actual language extension. Despite the extension being a much requested feature, no-one has thusfar successfully taken on the challenge of continuing upon their work, to the best of our knowledge.
Fifteen years after date, this thesis has elaborated on the idea of Hinze and Peyton Jones. We describe a formal specification of the extension and provide an algorithm for both type inference and elaboration into System F. The main challenge proved to be designing a coherent constraint entailment algorithm. For this, we borrowed the idea of focussing from the calculus of coherent implicits (Cochis), which describes a very similar language, supporting recursive resolution of quantified constraints in a Scala setting, by Schrijvers et al. [25] Furthermore, we investigated the meta-theory behind this extension and provide a prototype implementation of a Haskell compiler, extended with quantified class constraints.

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, specialisation Artificial Intelligence

Thesis supervisor: Prof. dr. ir. Tom Schrijvers

Assessor: Ir. George Karachalias, Dr. ir. Dominique Devriese

Mentor: Ir. George Karachalias