Optional Type Classes for Haskell

Rodrigo Ribeiro[1], Carlos Camarão[2], Lucília Figueiredo[3], and Cristiano Vasconcellos[4]

[1] DECSI, Universidade Federal de Ouro Preto (UFOP), João Monlevade
rodrigo@decsi.ufop.br
[2] DCC, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte
camarao@dcc.ufmg.br
[3] DECOM, Universidade Federal de Ouro Preto (UFOP), Ouro Preto
luciliacf@gmail.com
[4] DCC, Universidade do Estado de Santa Catarina (UDESC), Joinville
cristiano.vasconcellos@udesc.br

Abstract This paper explores an approach for allowing type classes to be optionally declared by programmers, i.e. for allowing programmers to overload symbols without having to declare the types of these symbols in type classes.
The idea is based on defining the type of an overloaded symbol without an anotated type as the anti-unification of instance types defined for the symbol in a module, by automatically creating a type class with that single overloaded name. This depends on a modularization of instance visibility, as well as on a redefinition of Haskell's ambiguity rule. The paper presents the modifications to Haskell's module system that are necessary for allowing instances to have a modular scope, based on previous work published by one of the authors. The definition of the type of overloaded symbols as the anti-unification of available instance types and the redefined ambiguity rule is also based on previous works by the authors.
The added flexibility to Haskell-style of overloading is illustrated by defining a type system and a type inference algorithm that allows overloaded record fields.

1 Introduction

The versions of Haskell supported by GHC [3] — the prevailing Haskell compiler — are becoming complex, to the point of affecting the view of Haskell as the best choice for general-purpose software development. A basic issue in this regard is the need of extending the language to allow multiple parameter type classes (MPTCs). This extension is thought to require additional mechanisms, such as functional dependencies or type families. In another paper [1], we have shown that the introduction of MPTCs in the language can and should be done without the need of additional mechanisms: a simplifying change is sufficient, to Haskell's ambiguity rule. Interested readers are referred to [1]. The main ideas are summarized below.

Haskell with MPTCs uses constrained types of the form $\forall \overline{a}.\, C \Rightarrow \tau$, where C is a set of constraints and τ is the well-known simple (unconstrained, unquantified) type; a constraint is a class name followed by a sequence of type variables.

In (GHC) Haskell, ambiguity is a property of a type: a type $\forall \overline{a}.\, C \Rightarrow \tau$ is ambiguous if there exists a type variable that occurs in the constraints (C) that is not uniquely determined from the set of type variables that occur in the simple type (τ). This unique determination specifies that, for each type variable a that occurs in C but not in τ there must exist a functional dependency $b \mapsto c$ for some b in τ (or a similar unique determination specified via type families). Notation $b \mapsto c$ is used, instead of $b \to c$, to avoid confusion with the notation used to denote functional types.

A new definition, which we prefer to call here *expression ambiguity* (in [1] it is called *delayed closure ambiguity*), uses a similar property, of variable reachability, that is independent of functional dependencies and type families: a type variable a that occurs in a set of constraints is reachable from the set of type variables in τ if it occurs in τ or there exists a type variable b in C that is reachable. For example, in $C \Rightarrow b$, where $C = (D\, a\, b, E a)$, type variable a is reachable from the set of type variables in b.

The presence of unreachable variables in a constraint $pi \in C$ characterizes overloading resolution, or, in other words, that overloading is resolved for π: it characterizes that there is no context in which an expression with such a type could be placed that could instantiate such unreachable variables.

The presence of unreachable variables does not necessarily imply ambiguity. Ambiguity is a property of an expression, and it depends on the context in which the expression occurs, and on entailment (or satisfiability) of constraints.

Entailment of constraints and its algorithmic (functional) counterpart are well-known in the Haskell world (see e.g. [5,8,1]).

Informally, a set of constraints C is entailed (or satisfied) in a program P if there exists a substitution ϕ such that $\phi(C)$ is contained in the set of instance declarations of P, or is implied by the transitivity implied by the set of class and instance declarations occuring in P. For a formal definition, see e.g. [5,1]. In this case we say that C is entailed by $\phi(C)$.

For example, *Eq* `[[`*Integer*`]]` is entailed if we have instances *Eq Integer* and *Eq a* `=>` *Eq* `[`*a*`]`, visible in the context where an expression whose type has a constraint *Eq* `[[Integer]]` occurs.

If overloading is resolved for a constraint C occurring in a type $\sigma = C, D \Rightarrow \tau$ then exactly one of the following holds:

- C is entailed by a single instance; in this case a type simplification (also called "improvement") occurs: σ can be simplified to $D \Rightarrow \tau$;
- C is entailed by more than instance; in this case we have a type error: ambiguity;
- C is not entailed (by any instance); in this case we have also a type error: unsatisfiability.

Note that variables in a single constraint are either all reachable or all unreachable. If they are unreachable, either the constraint can be removed, in the case of single entailment, or there is a type error (either ambiguity, in the case of two or more entailments, or unsatisfiability, in the case of no entaiment).

Instead of being dependent on the specification or not of functional dependencies or type families, ambiguity depends on the existence of (two or more) instances in a program context when overloading is resolved for a constraint on the type of an expression.

The possibility of a modular control of the visibility of instance definitions conforms to this simplifying change. This is the subject of Section 3.

Also in conformance with this change is the possibility, explored in this paper, of allowing type classes to be optionally declared by programmers, i.e. for allowing programmers to overload symbols without having to declare the types of these symbols in type classes.

A type system and a type inference algorithm for a core-Haskell language where type classes can be optionally declared is presented in Section 4. The idea is based on defining the type of unanottated overloaded symbol as the anti-unification of instance types defined for the symbol in a module, by automatically creating a type class with a single overloaded name. This depends on a modularization of instance visibility (as well as on a redefintion of Haskell's ambiguity rule).

The paper presents the modifications to Haskell's module system that are necessary to allow instances to have a modular scope, based on previous work published by one of the authors. The definition of the type of overloaded symbols as the anti-unification of available instance types and the redefined ambiguity rule is also based on previous works by the authors.

The added flexibility to Haskell-style of overloading is illustrated by defining a type system and a type inference algorithm that allows overloaded record fields (Section 5).

2 Preliminaries

In this section we introduce some basic definitions and notations. We consider that meta-variables defined can appear primed or subscripted.

Meta-variable usage is defined in the paper as follows: x, y denote term variables, α, β ($a, b, ...$ in examples) type variables, e a term, τ, ρ simple types, κ a constraint set, $x : \tau$ a constraint, σ a type, Γ a typing context, that is, a set of pairs written as $x : \sigma$, and S a substitution.

The notation $\overline{a}^{\,n}$, or simply $\overline{a}$, denotes the sequence $a_1, \ldots, a_n$, where $n \geq 0$. When used in a context of a set, it denotes the corresponding set of elements in the sequence $\{a_1, \ldots, a_n\}$.

A substitution is a function from type variables to simple type expressions (cf. Section 4). The identity substitution denoted by id. $S\sigma$ represents the capture-free operation of substituting $S(\alpha)$ for each free occurrence of α in σ.

We overload the substitution application on constraints, constraint sets and sets of types. Definition of application on these elements is straightforward. The symbol $\circ$ denotes function composition and $dom(S) = \{\alpha \mid S(\alpha) \neq \alpha\}$.

The notation $S[\overline{\alpha} \mapsto \overline{\tau}]$ denotes the updating of S such that $\overline{\alpha}$ maps to $\overline{\tau}$, that is, the substitution S' such that $S'(\beta) = \tau_i$ if $\beta = \alpha_i$, for $i = 1, ..., n$, otherwise $S(\beta)$. Also, $[\overline{\alpha} \mapsto \overline{\tau}] = id[\overline{\alpha} \mapsto \overline{\tau}]$.

2.1 Anti-unification of instance types

A type τ is a generalization — also called (first-order) *anti-unification* [2] — of simple types $\overline{\tau}^{\,n}$ if there exist substitutions $\overline{S}^{\,n}$ such that $S_i(\tau) = \tau_i$, for $i = 1, \ldots, n$.

We call a function that gives the least generalization of a finite set of simple types the *least common generalization* (*lcg*).

An algorithm for computing the *lcg* of a finite set of types in presented in Figure 1. The concept of least common generalization was studied by Gordon Plotkin [6,7], that defined a function for constructing a generalization of two symbolic expressions.

$$lcg(\mathbb{T}) = \tau \quad \texttt{where } (\tau, S) = lcg'(\mathbb{T}, \emptyset)\texttt{, for some } S$$

$$lcg'(\{\tau\}, S) = (\tau, S)$$

$$lcg'(\{\tau_1, \tau_2\} \cup \mathbb{T}, S) = lcg''(\tau, \tau', S') \quad \texttt{where } \begin{array}{l}(\tau, S_0) = lcg''(\tau_1, \tau_2, S)\\ (\tau', S') = lcg'(\mathbb{T}, S_0)\end{array}$$

$$\begin{array}{l} lcg''(C\,\overline{\tau}^{\,n},\ D\,\overline{\rho}^{\,m}, S) = \\ \quad \texttt{if } S(\alpha) = (C\,\overline{\tau}^{\,n},\ D\,\overline{\rho}^{\,m}) \texttt{ for some } \alpha \texttt{ then } (\alpha, S) \\ \quad \texttt{else} \\ \quad\quad \texttt{if } n \neq m \texttt{ then } (\beta, S[\beta \mapsto (C\,\overline{\tau}^{\,n},\ D\,\overline{\rho}^{\,m})]) \\ \quad\quad\quad \texttt{where } \beta \texttt{ is a fresh type variable} \\ \quad\quad \texttt{else } (\psi\,\overline{\tau'}^{\,n}, S_n) \\ \quad\quad\quad \texttt{where } (\psi, S_0) = \begin{cases} (C, S) & \texttt{if } C = D \\ (\alpha, S\,[\alpha \mapsto (C, D)]) & \texttt{otherwise, } \alpha \texttt{ is fresh} \end{cases} \\ \quad\quad\quad\quad\quad (\tau'_i, S_i) = lcg''(\tau_i, \rho_i, S_{i-1}),\ \texttt{for } i = 1, \ldots, n \end{array}$$

Figure 1. Least Common Generalization

3 Modularization of Instances

This paper does not attempt to discuss any major revision to Haskell's module system. We summarize in subsection 3.1 the work, presented in [4], that allows a modular control of the visibility of instance definitions. This has the additional benefit of enabling type classes to be optionally declared by programmers, by the introduction of a single additional rule (to account for the possibility of type classes to be declared or not):

Definition 1 (Type of overloaded variable).
If the type of an overloaded variable (i.e. a variable that is introduced in an instance definition) is not explicitly annotated in a type class declaration, then the variable's type is the anti-unification of instance types defined for the variable in the current module; otherwise, it is the annotated type.

Instance modularization and the rule of expression ambiguity, that considers the context where an expression occurs to detect whether an expression is ambiguous or not, has profound consequences. Consider, for example:

```
module A where
  class Show t ...
  class Read t ...
  instance Show Int ...
  instance Read Int ...
  f = show . read

module B where
  import A
  instance Read Bool ...
  instance Show Bool ...
  g = f "True"
```

The definition of f in module A is well-typed, because constraints (*Show* a, *Read* a) can be removed; this occurs because there exists a single instance, in module A, for each constraint, that entails it. As a result, f has type *String*$\rightarrow$ *String*. Its use in module B is (then) also well-typed.

That means: f's semantics is a function that receives a value of type *String* and returns a value of type *String*, according to the definition of f given in module A. The semantics of an expression involves passing a (dictionary) value that is given in the context of usage if, *and only if*, the expression has a constrained type.

3.1 Instance visibility control: a summary

Modularization of instance definitions is allowed by means of the importation and exportion of instances as shown in [4]. Essentially, import and export clauses

can specify, instead of just names, occurrences of `instance` T D, where T is a type and D is a class name. We have:

$$\texttt{module}\ M\ (\texttt{instance}\ T\ D\ \texttt{where}\ \ldots)$$

specifies that the instance of type T for class D is exported in module M.

$$\texttt{import}\ M\ (\texttt{instance}\ T\ D)$$

specifies that the instance of type T for class D is imported from M, in the module where the import clause occurs.

Alternatively could simply give a name to an instance, in an instance declaration, and use that name in import and export clauses (see [4]).

3.2 Pros and Cons of Instance Modularization

Among the advantages of this simple change, we cite (following [4]):

- programmers have better control of which entities are necessary and should be in the scope of each module in a program;
- it is possible to define and use more than one instance for the same type in a program;
- problems with orphan instances (i.e. instances defined in a module where neither the definition of the data type nor the definition of the type class) do not occur (for example, distinct instances of *Either* for class *Monad*, say one from package *mtl* and another from *transformers*, can be used in a program);
- the introduction of newtypes, as well as the use of functions that include additional (-by) parameters (such as e.g. *sortBy* in module *Data.List*) can be avoided.

With instance modularization, programmers need to be aware of which entities are exported and imported (i.e. which entities are visible in the scope of a module) and their types, in particular if they are overloaded or not.

A simple change like a type annotation for a variable exported from a module, can lead to a change in the semantics of using this variable in another module.

4 Core-Haskell with Optional Type Classes

5 Records with overloaded fields

6 Conclusion

References

1. Carlos Camarão and Rodrigo Ribeiro and Lucília Figueiredo. Ambiguity and Constrained Polymorphism. *Science of Computer Programming*, ?(?):?–?, 2016.

2. C.C. Chang and H.J. Keisler. *Model Theory*. Dover Books on Mathematics, 2012. 3rd ed.
3. Glasgow Haskell Compiler home page. `http://www.haskell.org/ghc/`.
4. Marco Silva and Carlos Camarão. Controlling the Scope of Instances in Haskell. In *Proc. of SBLP'2011*, pages 29–30, 2011.
5. Mark Jones. *Qualified Types: Theory and Practice*. PhD thesis, Distinguished Dissertations in Computer Science. Cambridge Univ. Press, 1994.
6. Gordon D Plotkin. A note on inductive generalisation. *Machine intelligence*, 5(1):153–163, 1970.
7. Gordon D Plotkin. A further note on inductive generalisation. *Machine Intelligence*, 6, 1971.
8. Peter Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM TOPLAS*, 27(6):1216–1269, November 2005.