Optional Type Classes

April 14, 2016

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 un-anottated 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 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 — the prevailing Haskell compiler — are becoming complex, to the point of affecting the view of Haskell as being the best choice for general-purpose software development. A critical issue in this regard is the need of extending the language to allow multiple parameter type classes (MPTCs). Unfortunately, this extension is thought to require additional mechanisms, such as functional dependencies or type families. In another paper [1], we have shown how MPTCs can be introduced in the language without the need of additional mechanisms. A simplifying change is sufficient, in Haskell's ambiguity rule. Interested readers are referred to [1]; a brief summary is included 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 $b \rightarrow c$, to avoid confusion with the notation used to denote functional types.

Our definition of ambiguity uses a similar property, of variable reachability, that is independent of functional dependencies or type families: a type variable a that occurs in a constraint $\pi \in C$ is reachable from the set of type variables in τ if it occurs in τ or there exists a type variable b in the same constraint (π) where a occurs 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: it characterizes that there is no context in which an expression with such a type could be placed that could instantiate such unreachable variables. It 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. [2, 3, 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 declarations occuring in P. For a formal definition, see e.g. [2, 1]. In this case we say that C is entailed by $\phi(C)$. For example, ...

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 unrechable, 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).

Neither Functional dependencies nor type families are needed. Instead, 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 is in conformance with this simplifying change. This is the subject of section 2.

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.

The idea is based on defining the type of un-anottated 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 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.

2 Modularization of Instances

Let the instance type of a non-overloaded variable be the variable's type. Unless the type of a variable is explicitly annotated, either in a type class or in a type annotation, a variable's type is the anti-unification of instance types defined for the variable in a module.

Consider, for example, ...

3 Ambiguity Rule

4 Anti-unification of instance types

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] Mark Jones. *Qualified Types: Theory and Practice.* PhD thesis, Distinguished Dissertations in Computer Science. Cambridge Univ. Press, 1994.

[3] Peter Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM TOPLAS*, 27(6):1216–1269, November 2005.