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. programmers can overload symbols without declaring their types in type classes.

The type of an overloaded symbol is, if not explicitly defined in a type class, automatically determined from the anti-unification of instance types defined for the symbol in the relevant module. A type class having the overloaded name as its unique member is automatically created from this type.

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 by 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 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 a 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.

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 of the constraints on the expression's type.

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 4.

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 5. 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 6).

2 Preliminaries

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

The notation $\overline{a}^{\,n}$, or simply $\overline{a}$, denotes the sequence $a_1 \cdots a_n$, or $a_1, \ldots, a_n$, or $a_1; \ldots; a_n$, depending on the context where it is used, where $n \geq 0$. When used in a context of a set, it denotes $\{a_1, \ldots, a_n\}$. It can be used with more than one variable; for example, in $\overline{x = e}^{\,n}$, it denotes the sequence $x_1 = e_1, \ldots, x_n = e_n$.

A substitution is a function from type variables to simple type expressions. The identity substitution denoted by id. $\phi(\sigma)$ (or simply $\phi\,\sigma$) represents the capture-free operation of substituting $\phi(\alpha)$ 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 $dom(\phi) = \{\alpha \mid \ \phi(\alpha) \neq \alpha\}$.

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

3 Anti-unification of instance types

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

We say that τ' is less general than τ, written $\tau \leq \tau'$, if there exist ϕ such that $\phi(\tau) = \tau'$.

The *least common generalization* (lcg) of a set of types $\mathbb{T}$ and a type τ holds, written as $\mathtt{lcg}_r(\mathbb{T}, \tau)$, if, for all generalizations τ' of $\mathbb{T}$, we have $\tau \leq \tau'$.

An algorithm for computing the lcg of a finite set of types in presented in Figure 1. The concept was studied by Gordon Plotkin [6,7], that defined a function for constructing a generalization of two symbolic expressions. In Figure 1, we present function *lcg*, that gives the lcg of a finite set of simple types by recursion on the structure of set $\mathbb{T}$, using a function to compute the generalization of two simple types. For two types τ_1 and τ_2 the idea is to recursively traverse the structure of both types using a finite map to store previously generalized types. Whenever we found two different type constructors, we search on the finite map if they have been previously generalized. If this was the case, the generalization is returned. But if these two type constructors aren't in the finite map we insert them using a fresh type variable as their generalization and return this new variable.

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

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

$$lcg'(\{\tau_1\} \cup \mathbb{T}, \phi) = lcg''(\tau_1, \tau', \phi') \quad \texttt{where } (\tau', \phi') = lcg'(\mathbb{T}, \phi)$$

$$\begin{array}{l}
lcg''(C\,\overline{\tau}^{\,n},\ D\,\overline{\rho}^{\,m}, \phi) = \\
\quad \texttt{if } \phi(\alpha) = (C\,\overline{\tau}^{\,n},\ D\,\overline{\rho}^{\,m}) \texttt{ for some } \alpha \texttt{ then } (\alpha, \phi) \\
\quad \texttt{else} \\
\qquad \texttt{if } n \neq m \texttt{ then } (\beta, \phi[\beta \mapsto (C\,\overline{\tau}^{\,n},\ D\,\overline{\rho}^{\,m})]) \\
\qquad\quad \texttt{where } \beta \texttt{ is a fresh type variable} \\
\qquad \texttt{else } (\psi\,\overline{\tau'}^{\,n}, \phi_n) \\
\qquad\quad \texttt{where } (\psi, \phi_0) = \begin{cases} (C, \phi) & \texttt{if } C = D \\ (\alpha, \phi\,[\alpha \mapsto (C, D)]) & \texttt{otherwise, } \alpha \texttt{ is fresh} \end{cases} \\
\qquad\qquad\quad (\tau'_i, \phi_i) = lcg''(\tau_i, \rho_i, \phi_{i-1}), \texttt{ for } i = 1, \ldots, n
\end{array}$$

Figure 1. Least Common Generalization

As an example of the use of *lcg*, consider the following types (of functions *map* on lists and trees, respectively):

$$(a \to b) \ \to \ \texttt{[}a\texttt{]} \ \to \ \texttt{[}b\texttt{]}$$
$$(a \to b) \ \to \ \mathit{Tree} \ \ a \ \to \ \mathit{Tree} \ \ b$$

A call of *lcg* for a set with these types yields type $(a \to b) \to c \ \ a \to c \ \ b$, where c is a generalization of type constructors `[]` and *Tree*.

We have:

Theorem 1 (Soundness of ***lcg*****)** *For all (sets of simple types)* $\mathbb{T}$*, we have that* $\mathit{lcg}(\mathbb{T})$ *yields a generalization of* $\mathbb{T}$*.*

Theorem 2 (Completeness of ***lcg*****)** *For all (sets of simple types)* $\mathbb{T}$*, we have that* $\texttt{lcg}_r(\mathbb{T}, \mathit{lcg}(\mathbb{T}))$ *holds, i.e. if* τ *is a generalization of* $\mathbb{T}$ *then* $\mathit{lcg}(\mathbb{T}) \leq \tau$*.*

Theorem 3 (Compositionality of ***lcg*****)** *For all non-empty (sets of simple types)* $\mathbb{T}, \mathbb{T}'$*, we have that* $\mathit{lcg}(\mathit{lcg}(\mathbb{T}), \mathit{lcg}(\mathbb{T}')) = \mathit{lcg}(\mathbb{T} \cup \mathbb{T}')$*.*

Theorem 4 (Uniqueness of ***lcg*****)** *For all (sets of simple types)* $\mathbb{T}$*, we have that* $\mathit{lcg}(\mathbb{T})$ *is unique, up to variable renaming.*

The proofs use straighforward induction on the number and complexity of elements of $\mathbb{T}$.

4 Modularization of Instances

This paper does not attempt to discuss any major revision to Haskell's module system. We summarize in subsection 4.1 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 — 2is 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.

4.1 Instance visibility control: a summary

Modularization of instance definitions can be 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` $A\ \overline{\tau}$, where $\overline{\tau}$ is a (non-empty) sequence of types and A is a class name. We have:

$$\texttt{module}\ M\ (\texttt{instance}\ A\ \overline{\tau}, \ldots)\ \texttt{where} \ldots$$

specifies that the instance of $\overline{\tau}$ for class D is exported in module M.

$$\texttt{import}\ M\ (\texttt{instance}\ A\ \overline{\tau}, \ldots)$$

specifies that the instance of $\overline{\tau}$ for class A is imported from M, in the module where the import clause occurs.

Alternatively, we can simply give a name to an instance, in an instance declaration, and use that name in import and export clauses (see [4]). However, in this paper we don't need to give a name to an instance, since we only consider instances of undeclared classes, which have a single member, and we can thus use the name of the member as the instance name.

4.2 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. the (first) parameter of function *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.

5 Mini-Haskell with Optional Type Classes

In this section we present a type system for mini-Haskell, where type class declaration is optional. Programmers can overload symbols without declaring their types in type classes. The type of an overloaded symbol is, if not explicitly defined in a type class, based on the anti-unification of instance types defined for the symbol in the relevant module.

Figure 2 shows meta-variable usage and the context-free syntax of mini-Haskell: expressions and their types, modules and programs. Meta-variables can be possibly primed or subscripted. An instance can be specified without specifying a type class.

For simplicity and following common practice, kinds are not considered in type expressions and type expressions which are not simple types are not explicitly distinguished from simple types. Type expression variables are called simply type variables.

As usual, we assume the existence of type constructor $\rightarrow$, that is written as an infix operator $(\tau \rightarrow \tau')$. A type $\forall\, \overline{a}.\, C \Rightarrow \tau$ is equivalent to $C \Rightarrow \tau$ if $\overline{a}$ is empty and, similarly, $C \Rightarrow \tau$ is equivalent to τ if C is empty.

For simplicitly, imported and exported variables and instances must be explicitly indicated, e.g. we do not include notations for exporting and importing all variables of a module.

Multi-parameter type classes are supported. In this papert we do not consider recursivity, neither in let-bindings nor in instance declarations.

A program theory P is a set of axioms of first-order logic, generated from class and instance declarations occurring in the program, of the form $C \Rightarrow \pi$, where C is a set of simple constraints and π is a simple constraint (cf. Figure 2).

Class Name	A, B	
Module Name	M, N	
Type variable	a, b, α, β	
Type constructor	T	
Simple Constraint	π	$::= A\,\overline{\tau}$
Unquantified Constraint	ψ	$::= C \Rightarrow \pi$
Constraint	θ	$::= \forall\,\overline{\alpha}.\,\psi$
Set of Simple Constraints	C, D	
Constrained Type	δ, ϵ	$::= C \Rightarrow \tau$
Simple Type	τ, ρ	$::= \alpha \mid T \mid \tau\,\tau'$
Type	σ	$::= \forall\,\overline{\alpha}.\,\delta$
Program Theory	P, Q	
Variable	x, y, z	
Expression	e	$::= x \mid \lambda x.\,e \mid e\,e' \mid \texttt{let}\ x = e\ \texttt{in}\ e'$
Program	p	$::= \overline{m}$
Module	m	$::= \texttt{module}\,M\,(X)\ \texttt{where}\ \overline{I}; \bar{d}$
Export clause	X	$::= \overline{\iota}$
Import clause	I	$::= \texttt{import}\ M\,(\,\overline{\iota}\,)$
Item	ι	$::= x$
Declaration	d	$::= \mathit{classDecl} \mid \mathit{instDecl} \mid \overline{b}$
Class Declaration	$\mathit{classDecl}$	$::= \texttt{class}\ C \Rightarrow A\,\overline{a}\ \texttt{where}\ \overline{x : \delta}$
Instance Declaration	$\mathit{instDecl}$	$::= \texttt{instance}\ C \Rightarrow A\,\overline{\tau}\ \texttt{where}\ \overline{b} \mid \texttt{instance}\ b$
Binding	b	$::= x = e$

Figure 2. Context-free syntax of mini-Haskell and types

Entailment of a set of constraints C by a program theory P is written as $P \vdash_e C$ (see e.g. [1]).

Typing contexts are indexed by module names. $\Gamma(M)$ gives a function that can be eiter:

- a function on variable names to types, if the variable is not an overloaded name declared as an instance of an undeclared class.
  In this case, $\Gamma(M)(x)$ gives the type of x in module M and typing context Γ.
- otherwise, a function on the instance type to the type of the instance name. In this case, $\Gamma(M)(x, \tau)$ gives the type of x of type τ, in module M and typing context Γ (τ must be the type of x in an instance with an undeclared class, as described below).

A special, empty module name, denoted by `[]`, is used for names exported by modules, to control the scope of names that use import and export clauses. Also, a reserved name γ is used to refer to the current module, being defined and used in the type system and relations to control import and export clauses.

It is not necessary to save multiple instance types for the same variable in a typing context, neither it is necessary to use instance types in typing contexts (they are needed only in the program theory); only the lcg of instance types is used, because of lcg compositionality (theorem 3). When a new instance is declared, if it is an instance of a declared class the type systems guarantees that each field is an instance of the type declared in the type class; otherwise (i.e. it is the single field of an undeclared class), its (new) type is given by the lcg of the existing type (an existing lcg of previous instance types) and the instance type.

A partial order on possibly constrained and possibly quantified types, based on constraint set entailment, is defined in Figure 3. We use $P \vdash_e C$ to abbreviate $P \vdash_e \pi$ for all $\pi \in C$. Note that type ordering disregards constraint set satisfiability. Satisfiability is only important when considering whether a constraint set C can be removed from a constrained type $C, D \Rightarrow \tau$ (C can be removed if and only if overloading for C has been resolved and there exists a single satisfying substitution for C)[1].

$$\frac{P \vdash_e \phi C \quad D \subseteq \phi C \quad \bar{b} \subseteq tv(D) \cup tv(\phi\tau)}{\forall \bar{a}.\, C \Rightarrow \tau \leq_P \forall \bar{b}.\, D \Rightarrow \phi\tau}$$

Figure 3. Partial order on Types

A type system for core-Haskell is presented in Figure 4, using rules of the form $P; \Gamma \vdash_0 e : \delta$, which means that e has type δ in typing context Γ and program theory P.

Rule (LET) performs constraint set simplification before type generalization. Constrait set simplification $\gg_P$ is a relation on constraints, defined as a composition of improvement and context reduction [1].

In rule (APP), the constraints on the type of the result are those that occur in the function type plus not all constraints that occur in the type of the argument but only those that have variables reachable from the set of variables that occur in the simple type of the result or in the constraint set on the function type. This allows, for example, not including constraints on the type of the following expressions, where o is any expression, with a possibly non-empty set of constraints on its type: $\mathit{flip}\ \ \mathit{const}\ \ o$ (where const has type $\forall a, b.\, a \to b \to a$ and flip has type $\forall a, b, c.\, (a \to b \to c) \to b \to a \to c$), which should denote an identity function, and $\mathit{fst}\ (e, o)$, which should have the same denotation as e.

A variable $a \in \mathit{tv}(C)$ is called reachable from, or with respect to, a set of type variables V if $a \in V$ or if $a \in \pi$ for some $\pi \in C$ such that there exists $b \in \mathit{tv}(\pi)$ such that b is reachable. $a \in \mathit{tv}(C)$ is called unreachable if it is not reachable. The set of reachable and unreachable type variables of constraint set C *from* V is denoted by $\mathit{reachableVars}(C, V)$.

$C \oplus_V D$ denotes the constraint set obtained by adding to C constraints from D that have type variables reachable from V:

$$P \oplus_V Q = P \cup \{\psi \in Q \mid \mathit{tv}(\psi) \cap \mathit{reachableVars}(Q, V) \neq \emptyset\}$$

$\mathit{gen}(\delta, \sigma, V)$ holds if $\sigma = \forall\, \overline{\alpha}.\, \delta$, where $\overline{\alpha} = \mathit{tv}(\delta) - V$.

A type system for mini-Haskell, which extends core-Haskell with (optional) type classes, modules and modularized instance declarations, is presented in Figures 5 and 7. Rule (MOD) uses relations ($\vdash_{\Downarrow}$) and ($\vdash^{X}_{\Uparrow}$), which are defined separately, for clarity, in Figures 6 and 7.

The import relation $\Gamma \vdash_{\Downarrow} \overline{I} \rightsquigarrow \Gamma'$ yields a typing context (Γ') from a typing context (Γ) and a sequence of import clauses ($\overline{I}$).

Relation $P; \Gamma \vdash^{X}_{\Uparrow} \bar{d} : (R, P', \Gamma')$ is used for specifying the types a sequence of bindings, from a typing context (Γ), a program theory (P) and a set of exported names (X); it yields a record of elements (R) of the form ($x : \sigma$), together with both i) a new typing context (Γ'), modified to contain elements of X, so that $\Gamma'(\texttt{[]})$ contains the types of each $x \in X$, and ii) a new program theory (P'), updated from class and instance declarations. Relation ($\vdash_0$) is used to check that expressions of core-Haskell that occur in declarations are well-typed.

There must exist a sequence of derivations for typing a sequence of modules that composes a program that starts from an empty typing context, or from a typing context that corresponds to predefined library modules. Recursive modules are not treated in this paper.

$C \oplus_V D$ denotes the constraint set obtained by adding to C constraints from D that have type variables reachable from V:

$$P \oplus_V Q = P \cup \{\psi \in Q \mid \mathit{tv}(\psi) \cap \mathit{reachableVars}(Q, V) \neq \emptyset\}$$

Relation $\oplus$ is also used in Figure 7 for concatenation of record elements: $\{x_1, \ldots, x_n\} \oplus \{y_1, \ldots, y_m\} = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$.

$$\frac{\Gamma(\gamma)(x) = \sigma \quad \sigma \leq_P \delta}{P; \Gamma \vdash_0 x : \delta} \text{ (VAR)}$$

$$\frac{(\Gamma(\gamma), x \mapsto \tau) \vdash_0 e : C \Rightarrow \tau'}{P; \Gamma \vdash_0 \lambda x.\, e : C \Rightarrow \tau \rightarrow \tau'} \text{ (ABS)}$$

$$\frac{\begin{array}{cc} P; \Gamma \vdash_0 e : C \Rightarrow \tau' \rightarrow \tau & P; \Gamma \vdash_0 e' : C' \Rightarrow \tau' \\ V = tv(\tau) \cup tv(C) & (C \oplus_V C') \gg_P D \end{array}}{P; \Gamma \vdash_0 e\, e' : D \Rightarrow \tau} \text{ (APP)}$$

$$\frac{\begin{array}{ll} P; \Gamma \vdash_0 e : C \Rightarrow \tau & C \gg_P D \\ gen(D \Rightarrow \tau, \sigma, tv(\Gamma)) & P; (\Gamma(\gamma), x \mapsto \sigma) \vdash_0 e' : \delta \end{array}}{P; \Gamma \vdash_0 \texttt{let} \ \ x = e \ \ \texttt{in} \ \ e' : \delta} \text{ (LET)}$$

Figure 4. Core-Haskell Type System

$$\frac{\Gamma_0 \vdash_\Downarrow \bar{I} : \Gamma \quad P; \Gamma \vdash_\Uparrow^X \bar{d} : (R, P', \Gamma')\}}{P; \Gamma_0 \vdash \texttt{module}\ M\,(X)\ \texttt{where}\ \bar{I}; \bar{d} : R} \text{ (MOD)}$$

Figure 5. Mini-Haskell module rule

$$\frac{\Gamma'(M)(x) = \begin{cases} \Gamma(\texttt{[]})(x) & \text{if } M = \gamma \text{ and } x = \iota_k, 1 \leq k \leq n \\ \Gamma(M)(x) & \text{otherwise} \end{cases}}{\Gamma \vdash_\Downarrow \texttt{import}\ M\,(\,\bar{\iota}^{\,n}\,) : \Gamma'}$$

$$\frac{\Gamma_0 \vdash_\Downarrow \texttt{import}\ M\,(\,\bar{\iota}\,) : \Gamma \qquad \Gamma \vdash_\Downarrow \overline{I} : \Gamma'}{\Gamma_0 \vdash_\Downarrow \texttt{import}\ M\,(\,\bar{\iota}\,); \overline{I} : \Gamma'}$$

Figure 6. Import relation

$$\frac{\begin{array}{l} Q;\Gamma \vdash^X_{\Uparrow} \bar{d} : (R, Q', \Gamma') \quad Q = P \cup \begin{cases} \{C \Rightarrow A\,\overline{a}\} & \text{if } C \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\ \Gamma(M)(x) = \begin{cases} \delta_k & \text{if } x = x_k, 1 \leq k \leq n, \text{ and } M \in \{\gamma, \texttt{[]}\} \\ \Gamma_0(M)(x) & \text{otherwise} \end{cases} \end{array}}{P;\Gamma_0 \vdash^X_{\Uparrow} \texttt{class}\ C \Rightarrow A\,\overline{a}\ \texttt{where}\ \overline{x : \delta}^{\,n};\bar{d} : (R, Q', \Gamma')}$$

$$\frac{\begin{array}{ll} Q = P \cup \begin{cases} \{\forall\,\overline{a}.\,C \Rightarrow \pi\} & \text{if } C \neq \emptyset \\ \{\forall\,\overline{a}.\,\pi\} & \text{otherwise} \end{cases} & P \vdash_e C \Rightarrow \pi \\ Q;\Gamma \vdash_0 \overline{e}^{\,n} : \overline{\delta}^{\,n} & Q;\Gamma \vdash_{\Uparrow} \bar{d} : (R, Q', \Gamma') \end{array}}{P;\Gamma \vdash^X_{\Uparrow} \texttt{instance}\ C \Rightarrow \pi\ \texttt{where}\ \overline{x = e}^{\,n};\bar{d} : (\overline{x : \delta}^{\,n} \oplus R, Q', \Gamma')}$$

$$\frac{\begin{array}{l} P;\Gamma \vdash_0 e : C \Rightarrow \tau \quad \texttt{lcg}_r(\{\tau\} \cup \Gamma_0(\gamma)(x), \tau') \\ \Gamma(M)(y) = \begin{cases} \tau' & \text{if } y = x, (M = \gamma \text{ or } (M = \texttt{[]}, x \in X)) \\ \Gamma_0(M)(y) & \text{otherwise} \end{cases} \\ P;\Gamma \vdash^X_{\Uparrow} \bar{d} : (R, Q', \Gamma') \end{array}}{P;\Gamma_0 \vdash^X_{\Uparrow} \texttt{instance}\ x = e;\bar{d} : (\{x : C \Rightarrow \tau\} \oplus R, Q', \Gamma')}$$

$$\frac{\begin{array}{l} P;\Gamma_0 \vdash_0 e : \delta \quad \mathit{gen}(\delta, \sigma, \mathit{tv}(\Gamma_0)) \\ \Gamma(M)(y) = \begin{cases} \sigma & \text{if } y = x, (M = \gamma \text{ or } (M = \texttt{[]}, x \in X)) \\ \Gamma_0(M)(y) & \text{otherwise} \end{cases} \\ P;\Gamma \vdash^X_{\Uparrow} \bar{d} : (R, P', \Gamma') \end{array}}{P;\Gamma_0 \vdash^X_{\Uparrow} x = e;\bar{d} : (\{x : \delta\} \oplus R, P', \Gamma')}$$

Figure 7. Mini-Haskell rules for declarations

6 Records with overloaded fields

7 Related Work

8 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.