Optional Type Classes

Rodrigo Ribeiro

Dep. de Computação e Sistemas, Universidade Federal de Ouro Preto, ICEA, João Monlevade, Minas Gerais, Brasil

Carlos Camarão

Dep. de Ciência da Computação, Universidade Federal de Minas Gerais, Av. Antônio Carlos 6627, Belo Horizonte, Minas Gerais, Brasil

Lucília Figueiredo

Dep. de Computação, Universidade Federal de Ouro Preto, ICEB, Campus Universitário Morro do Cruzeiro, Ouro Preto, Minas Gerais, Brasil

Cristiano Vasconcellos

Dep. de Ciência da Computação, Universidade do Estado de Santa Catarina (UDESC), Joinville

Abstract

This paper presents an approach that allows programmers to declare and use overloaded symbols without declaring their types in type classes. The type of an overloaded symbol not explicitly defined in a type class is automatically determined from the anti-unification of the types of visible instances for this symbol in the relevant module. Overloaded symbols not declared in a type class have thus a modular scope.

The approach uses an open-world approach to overloading which means: "overloading resolution is checked when and only when there are unreachable variables in type constraints".

After overloading resolution is checked, either a constraint is removed, if there is a single entailing instance for the constraint, or else ambiguity or unsat-

Email addresses: `rodrigo@decsi.ufop.br` (Rodrigo Ribeiro), `camarao@dcc.ufmg.br` (Carlos Camarão), `luciliacf@gmail.com` (Lucília Figueiredo), `cristiano.vasconcellos@udesc.br` (Cristiano Vasconcellos)

isfiability occurs. Ambiguity means that there are two or more instances that entail the constraint, and unsatisfiability that there are none.

In this approach multi-parameter type classes do not need an extra mechanism, such as functional dependencies or type families, in order to avoid ambiguity, and the additional flexibility to Haskell-style of overloading allows, for example, polymonadic programming, and, together with optional type classes, an easy support of type directed name resolution and overloaded record fields.

Keywords: Ambiguity; Instance modularization; Optional type classes; Constrained polymorphism; Context-dependent overloading; Haskell

1. Introduction

The Haskell approach to overloading [1, 2] is a distinctive feature of the language that has been widely used. In Haskell, the types of overloaded names (or symbols) are provided in *type class* declarations and their definitions are introduced in *class instance* declarations. The type of an expression that uses an overloaded symbol is a constrained polymorphic type, that contains a type class constraint if the overloading is not resolved. For example, consider (where the body of definitions are replaced by ...):

```
class Show a where
  show ::  a → String
instance Show Int where
  show = ...
putStrLn ::  String → IO ()
putStrLn = ...
```

The type of *print*, defined as: *print x* = *putStrLn* (*show x*) is:

$$Show\ a \Rightarrow a \rightarrow IO\ ()$$

where constraint ($Show\ a$) is due to the use of the overloaded function *show*, of type $Show\ a \Rightarrow a \rightarrow String$ (declared in class *Show*).

A type in Haskell is of the form $\forall \overline{a}. C \Rightarrow \tau$, where $\overline{a}$ is a possibly empty set of type variables, C is a possibly empty set of constraints and τ is a simple (unconstrained, unquantified) type. A constraint is a class name followed by a sequence of simple types.

Overloading resolution in Haskell is *context-dependent*, that is, it depends on the context where an expression occurs. For function calls, this means that it depends also on the type of the function result (i.e. overloading resolution in an expression $f\ x$ can be based also on the type required by the context in which $f\ x$ occurs). Also, constants can be overloaded — for example, literals like `1`, `2` etc. can be used to represent fixed and arbitrary precision integers as well as fractional numbers. The possibility of using overloading without requiring overloading resolution in each function call makes overloading more expressive. Functions with types that differ only on the type of the result can be overloaded, for example *read* functions with types $\mathit{String} \to \mathit{Bool}$, $\mathit{String} \to \mathit{Int}$, each taking a string and generating the denoted value in the corresponding type. More importantly, functions can use overloaded functions in the definition of other functions, as well as define instances for distinct type constructors. For example, by using mapping and folding functions, e.g. with types $\mathit{Functor}\ f \Rightarrow (a \to b) \to f\ a \to f\ b$ and $\mathit{Foldable}\ f \Rightarrow (a \to b \to b) \to b \to f\ a \to b$, respectively, mapping and folding functions for instances of f as well as other higher-order polymorphic functions that use mapping and folding functions can be defined.

Another fundamental characteristic of overloading in Haskell, that supports context-dependent overloading resolution, is the reliance on an open-world approach. This implies that overloading resolution needs to be checked only when there exist variables in constraints that occur only in the constraints, i.e. do not occur in the simple (unconstrained part of the) type. The fact that overloading resolution does not need to be checked in every function application is important to defer overloading resolution and also for compile time efficiency. As explained in Section 4, in the presence of multi-parameter type classes (MPTCs), functional dependencies (FDs) [3, 4] or type families (TFs) [5, 6] are used to avoid "ambiguity". With MPTCs, "do not occur in the simple type" is modified in GHC [7], the prevailing Haskell compiler, to "is not uniquely determined from the set of type variables that occur in the simple type". This unique determination is such that, for each type variable a that occurs in the constraints C but not in the simple type τ of a constrained type $C \Rightarrow \tau$, there must exist a FD $b \mapsto a$ for some b in τ (in the case of TFs, a similar unique determination is specified; we use $b \mapsto a$ for a FD, instead of $b \to a$, to avoid confusion with the notation used to denote functional types.)

However, in Haskell the presence of type variables in constraints that are not uniquely determined from type variables in the simple type characterizes that "the type is ambiguous", not that overloading resolution needs to be checked. This guarantees that new instances can be introduced in well-typed Haskell programs without ever introducing a type error. On the other hand, i) it does not consider that the presence of type variables in constraints which are unreachable from the set of type variables in the simple type may be used to check overloading resolution, avoiding the need of using an extra mechanism for dealing with ambiguity problems in the presence of MPTCs (cf. Section 4); and ii) it does not distinguish overloading resolution from ambiguity, missing an opportunity to recover an intuitive common sense meaning of ambiguity as existence of two or more instances that can be used for an overloaded name in an expression, when overloading is resolved; iii) furthermore, the expressivity of overloading obtained allows easy support of, for example, polymonadic programming (Subsection 7.1) and, together with optional type classes, type directed name resolution (Subsection 7.2) and overloaded record fields (Subsection 7.3).

The paper presents minimalist modifications to Haskell's module system that are necessary for overloaded names to have a modular scope, with types annotated or not in type classes. Benefits and drawbacks of instance modularization are discussed in Section 5. Only global instances with a type class could be used in conjunction with modular instances without a type class, but in this paper we consider that all instances have a modular scope.

The mechanism of optional type classes presented allows declarations of overloaded names of the form `instance` $x = e$, without explicitly declaring x's type in a type class. Its type is determined from the anti-unification of visible instance types for x. The algorithm used for computing the type of x is presented in Section 3. Keywords `instance` is required if but only if there exists another definition of x visible in the defining module. The automatic creation of a type class for x simply uses x also as the name of the class. The initial constraint introduced when x is used is $x\,\tau$, where τ is the anti-unification of visible instance types of x. This was preferred to the use of a single parameter type class $x\ a$, or a possibly MPTC $x\,\overline{a}$, where $\overline{a}$ is a sequence of parameters that corresponds to the sequence of type variables that occur in the the anti-unification of visible instance types of x, ordered (say) lexicographically.

The proposed approach is formalized in Section 6, where a type system is presented for a mini-Haskell language with type classes optionally declared.

A type inference algorithm is presented in Section 8. It is proved to be sound and to infer, for a given expression in a given typing context, a type that is a minimal generalization of types derivable in the type system.

Related work is outlined in Section 9 and Section 10 concludes.

A prototype implementation of a type inference algorithm for Haskell supporting overloading without the need of defining a type class is available [8].

2. Preliminaries

This section introduces basic definitions and notations. Meta-variable usage and the syntax of types are given in Figure 1.

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.

Notation $\overline{x}^{\,n}$, or simply $\overline{x}$, is used to denote the sequence $x_1 \cdots x_n$, or $x_1, \ldots, x_n$, or $x_1; \ldots; x_n$, depending on the context where it is used, where $n \geq 0$, and x's can be either type variables, or mappings, or bindings etc. When used in a context of a set, it denotes $\{x_1, \ldots, x_n\}$.

Class Name	A	
Type variable	a, b	
Type constructor	T	
Constraint	π	$::= A\,\overline{\tau}$
Set of Simple Constraints	C	
QuantifiedConstraint	θ	$::= \forall\,\overline{a}.\,C \Rightarrow \pi$
Simple Type	τ, ρ	$::= a \mid T \mid \tau\,\tau'$
Constrained Type	δ	$::= C \Rightarrow \tau$
Type	σ	$::= \forall\,\overline{a}.\,\delta$
Substitution	ϕ	

Figure 1: Syntax of Types

The set of type variables occurring in X is denoted by $tv(X)$, where X can be a type, a constraint, sets of types or constraints, or a typing context.

A substitution ϕ is a function from type variables to simple type expressions. A type variable can be a constructor variable and the image of a substitution can be a type expression that is not a type (for example a type constructor with an arity that is not zero). Kinds are not considered in type expressions, and type expressions which are not simple types are not explicitly distinguished from simple types, but whenever necessary we distinguish the arity of type expressions. The identity substitution is denoted by id. $\phi(\sigma)$ (or simply $\phi\,\sigma$) represents the capture-free operation of substituting $\phi(a)$ for each free occurrence of a 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 $\phi[\overline{a \mapsto \tau}^{\,n}]$ denotes the substitution ϕ' such that $\phi'(b) = \tau_i$ if $b = a_i$, for $i = 1, \ldots, n$, otherwise $\phi(b)$. Also, $[\overline{a \mapsto \tau}] = id[\overline{a \mapsto \tau}^{\,n}]$.

3. Anti-unification of instance types

A simple 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 $i = 1, \ldots, n$. For example, $a_0 \rightarrow a_0$, $a_1 \rightarrow a_2$, and a_3 are generalizations of $\{\mathit{Int} \rightarrow \mathit{Int}, \mathit{Float} \rightarrow \mathit{Float}\}$. A generalization is also called a (first-order) *anti-unification* [9].

We say that τ is less general than τ', written $\tau \leq \tau'$, if there exists a substitution ϕ such that $\phi(\tau') = \tau$. For example, $a_0 \rightarrow a_0 \leq a_1 \rightarrow a_2 \leq a_3$.

The *least common generalization* (lcg) of a set of types S and a type τ holds, written as $\mathtt{lcg}_r(S, \tau)$, if, for all generalizations τ' of S we have $\tau \leq \tau'$.

The concept of least common generalization was studied by Gordon Plotkin [10, 11], who defined a function for constructing a generalization of two symbolic expressions. In Figure 2, we define function *lcg*, which returns a lcg of a finite set of simple types S, by recursion on the structure of S, where $\emptyset$ is an empty finite map. Function lcg_2 computes 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.

In function lcg_2, X is used to represent either a type constructor, constant or variable.

Whenever two type expressions of the same arity $\rho_1 = X_1\, \bar{\tau}_1^n$ and $\rho_2 = X_2\, \bar{\tau}_2^n$ are parameters of lcg_2, it is checked whether there is an entry that maps (X_1, X_2) to some type variable, i.e. whether there has been a previous generalization of X_1, X_2. If this is the case, the previous generalization is returned; otherwise, a fresh type variable is saved as their generalization. Calls to lcg_2 repeat this process for each pair of types $*\tau_1)_i$, $(\tau_2)_i$, for $i = 1, \ldots, n$. If two type expressions of distinct arities $X\, \overline{\tau}^{\,n}$ and $X'\, \overline{\rho}^{\,m}$, where $n \neq m$, are parameters of lcg_2, then lcg_2 is recursively called to yield i) the generalization τ' of $X\, \overline{\tau}^{\,n-1}$ and $X'\, \overline{\rho}^{\,m-1}$ and ii) the generalization τ'' of τ^n and ρ^m, to yield $\tau'\, \tau''$, as illustrated in Examples 2 and 3.

In the definition of *lcg*, the notation used for finite maps ζ is similar to that used for substitutions.

Example 1. As a first example, 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* (for c to be used in $c\, b$, mapping $(\texttt{[]}, \mathit{Tree}) \mapsto c$ is saved in the finite map, so that c can then be reused.

Example 2. As an example where lcg_2 computes the lcg of two types with type constructors that have distinct numbers of type arguments, consider.

$$\begin{array}{lll}
\tau_1 & = a & \to\; \tau_1' \\
\tau_1' & = & \mathit{IO}\;(\mathit{IORef}\;\; a) \\
\tau_2 & = a' & \to\; \tau_2' \\
\tau_2' & = & \mathit{ST}\; s\;(\mathit{STRef}\;\; s\; a')
\end{array}$$

We have, letting $S = \{\tau_1, \tau_2\}$, that $\mathit{lcg}(S) = \mathit{lcg}'(S, \mathit{id}) = \mathit{lcg}_2(\tau_1, \tau_2, \mathit{id})$.

The arities of the outermost type constructor, $(\to)$, of both τ_1 and τ_2, are equal (to 2), and then we have: $\mathit{lcg}_2(\tau_1, \tau_2, \emptyset) = ((\to) a_1 (b (b_1 a_1)))$, where a_1, b, b_1 are fresh type variables, since: $(a_1, \zeta_1) = \mathit{lcg}_2(a, a', \emptyset)$, where $\zeta_1 = \emptyset[(a, a') \mapsto a_1]$, and $(b\;(b_1\; a_1), \zeta_2) = \mathit{lcg}_2(\tau_1', \tau_2', \zeta_1)$.

The computation of $(b\;(b_1\; a_1), \zeta_2) = \mathit{lcg}_2(\tau_1', \tau_2', \zeta_1)$ illustrates a case with distinct arities of the outermost constructors of the two type arguments of lcg_2. We have: $\mathit{lcg}_2(\tau_1', \tau_2', \zeta_1) = (b\;(b_1\; a_1), \zeta_2)$ (where b, b_1 are fresh type variables), $(b, \zeta_1') = \mathit{lcg}_2(\mathit{IO}, \mathit{ST}\;\, s, \zeta_1)$, where $\zeta_1' = \zeta_1[(\mathit{IO}, \mathit{ST}\;\, s) \mapsto b]$ and $(b_1\; a_1, \zeta_2) = \mathit{lcg}_2(\mathit{IORef}\;\, a, \mathit{STRef}\;\, s\; a', \zeta_1')$.

The computation of $\mathit{lcg}_2(\mathit{IORef}\;\, a, \mathit{STRef}\;\, s\; a', \zeta_1')$ involves also a case with distinct arities of type constructors (arity 1 for *IORef* and 2 for *STRef*); we have: $\mathit{lcg}_2(\mathit{IORef}\, a, \mathit{STRef}\, s a', \zeta_1') = (b a_1, \zeta_2)$, where $(b, \zeta_2) = \mathit{lcg}_2(\mathit{IORef}, \mathit{STRef}\, s, \zeta_1')$, $\zeta_2 = \zeta_1'[(\mathit{IORef}, \mathit{STRef}\;\, s) \mapsto b]$ and $(a_1, \zeta_2) = \mathit{lcg}_2(a, a', \zeta_2)$.

The following is another example that illustrates a domain of the finite used in lcg_2 where the type expressions with distinct arities in the domain of the finite map ((T and $T\ Int$) have the same outermost type constructor (T).

Example 3. *Consider parameterised algebraic data types, such as e.g.:*

$$\texttt{data}\ F\ a\ \texttt{=}\ T\ a$$
$$\texttt{data}\ G\ f\ \texttt{=}\ T'\ \texttt{(}f\ \ Int\texttt{)}$$

and let $\tau_1 = F\ (T\ Int)$, $\tau_2 = G\,T$.

We have that $lcg(\{\tau_1, \tau_2\})$ yields the generalization $\tau = cb$, where c and b are fresh type variables. We have: $lcg(\{\tau_1, \tau_2\}) = lcg'(\{\tau_1, \tau_2\}, \emptyset) = lcg_2(\tau_1, \tau_2, \emptyset) = (c\,b, \zeta')$, where $(c, \zeta) = lcg_2(F, G, \emptyset)$ and $(b, \zeta') = lcg_2(T, T\ Int, \zeta)$.

The following theorems specify the behavior of lcg.

Theorem 1 (Soundness of lcg)**.** *For all non-empty sets of simple types S, we have that $lcg(S)$ yields a generalization of S.*

Theorem 2 (Completeness of lcg)**.** *For all non-empty sets of simple types S, we have that $\texttt{lcg}_r(S, lcg(S))$ holds (i.e. $lcg(S)$ is a generalization of S) and, for any τ that is a generalization of S, we have that $lcg(S) \leq \tau$.*

Theorem 3 (Compositionality of lcg)**.** *For all non-empty sets of simple types S, S', we have that $lcg(lcg(S), lcg(S')) = lcg(S \cup S')$.*

Theorem 4 (Uniqueness of lcg)**.** *For all non-empty sets of simple types S, we have that $lcg(S)$ is unique, up to variable renaming.*

The proofs use straighforward induction on the number and structural complexity of elements of S, using lemma 1.

Lemma 1. *For all τ_1, τ_2, ζ, we have that $lcg_2(\tau_1, \tau_2, \zeta) = (\tau, \zeta')$, for some τ, ζ' such that $\phi_1\tau = \tau_1$, $\phi_2\tau = \tau_2$, and ϕ_1, ϕ_2 are the first and second projections of the inverse of ζ', that is, $\phi_1(a) = \rho_1$ and $\phi_2(a) = \rho_2$ hold if and only if $\zeta'(\rho_1, \rho_2) = a$.*

```
$lcg(S) = \tau \quad$ where $(\tau, \zeta) = lcg'(S, \emptyset)$, for some $\zeta$

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

$lcg'(\{\tau_1\} \cup S, \zeta) = lcg_2(\tau_1, \tau', \zeta') \quad$ where $\quad (\tau', \zeta') \quad = lcg'(S, \zeta)$

$lcg_2\ (X\,\overline{\tau}^n,\ X'\,\overline{\rho}^m, \zeta) =$
    if $\zeta(X\,\overline{\tau}^n,\ X'\,\overline{\rho}^m) = a$ for some $a$ then $(a, \zeta)$
    else if $n = m$ then $(\zeta\,\overline{\tau'}^n, \zeta_n)$
                where $\quad (\psi, \zeta_0) \quad = \begin{cases} (T, \zeta) & \text{if } X = X' \\ (a, \zeta\,[(X, X') \mapsto a]) & \text{otherwise } (a \text{ fresh}) \end{cases}$
                      $(\tau'_i, \zeta_i) \quad = lcg_2(\tau_i, \rho_i, \zeta_{i-1})$, for $i = 1, \ldots, n$
        else if $n = 0$ or $m = 0$ then $(a, \zeta[(X\,\overline{\tau}^n,\ X'\,\overline{\rho}^m) \mapsto a])$
                                          where $a$ is a fresh type variable
            else $(\tau'\,\tau'', \zeta'')$
                where $(\tau', \zeta') = lcg_2(X\,\overline{\tau}^{n-1},\ X'\,\overline{\rho}^{m-1}, \zeta)$
                      $(\tau'', \zeta'') = lcg_2(\tau_m, \rho_m, \zeta')$
```

Figure 2: Least Common Generalization

4. Ambiguity Rule

As mentioned in the introduction, Haskell considers ambiguity as a property of a type, and does not distinguish it from a trigger condition to check overloading resolution: a type $\forall\,\overline{a}.\,C \Rightarrow \tau$ is ambiguous in Haskell if there exists a type variable that occurs in the set of constraints (C) but does not occur in τ. In the presence of FDs or TFs, "does not occur in τ" is modified in GHC to "is not uniquely determined from the set of type variables that occur in τ". This unique determination is such that, for each type variable a that occurs in C but not in τ there must exist a FD $b \mapsto a$ for some b in τ (in the case of type familes, a similar unique determination is specified). As mentioned the introduction, we use a different trigger condition to check overloading, based on variable reachability:

Definition 1 (Reachable Variable)**.** *A variable $a \in tv(C)$ is* reachable *from a set of type variables V if $a \in V$ or if $a \in \pi$ for some $\pi \in C$ such that there exists $b \in tv(\pi)$ such that b is reachable. Type variable $a \in tv(C)$ is unreachable if it is not reachable.*

The set of reachable type variables of constraint set C from V *is denoted by $reachableVs(C, V)$.*

The set of unreachable type variables of constraint set C from V *is denoted by $unReachVs(C, V)$.*

For example, in $(A_1\ a\ b, A_2\ a) \Rightarrow b$, type variable a is reachable $\{b\}$, because a occurs in constraint $A_1\ a\ b$, and b is reachable. Similarly, if $C = (A_1\ a\ b, A_2\ b\ c, A_3\ c)$, then c is reachable from $\{a\}$.

The presence of unreachable variables in a constraint $\pi \in C$, on a type $C \Rightarrow \tau$, is a trigger condition for overloading resolution (cf. Theorem 9, Section 6). This is motivated by the fact that there is no context in which an expression with such a type could be placed that could instantiate any of the unreachable variables occurring in π.

The presence of unreachable variables does not necessarily imply ambiguity. Ambiguity is a property of an expression, not of its type. It depends on the context where the expression occurs, and on *entailment* of the constraints on the expression's type. Also, by virtue of Haskell's *open-world* style of overloading, ambiguity can be checked only when overloading is resolved, i.e. only when there exist unreachable variables. When there are no unreachable variables, overloading is yet unresolved.

Entailment of constraints and its algorithmic (functional) counterpart, satisfiability, are well-known in the Haskell world (see e.g. [2, 12, 13]). For completeness, the definition of the entailment relation, used in the type system (Section 6), is defined in Subsection 6.4, and satisfability in 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 transitively implied by the set of class and instance declarations occurring in P [2, 13]. In this case we say that C is entailed by ϕ.

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

When overloading resolution is checked for a constraint π occurring in a constrained type $\delta = C \cup \{\pi\} \Rightarrow \tau$ — in Haskell notation $(C, \pi) \Rightarrow \tau$ —, one of the following holds:

1. π is entailed by a single instance. In this case a type simplification (also called “improvement”) occurs: δ can be simplified to $C \Rightarrow \tau$.
2. π is entailed by two or more instances. In this case we have a type error: ambiguity.
3. π is not entailed (by any instance). In this case we have also a type error: unsatisfiability.

Variables in a single constraint are either all reachable or all unreachable:

Lemma 2. *For all* π, V, *we have that* $unReachVs(\{\pi\}, V) \neq \emptyset$ *if and only if* $unReachVs(\{\pi\}, V) = tv(\pi)$.

Proof: Directly from Definition 1. $\square$

The possibility of a modular control of the visibility of instance definitions conforms to this simplifying change. This is the subject of Section 5.

5. Modularization of Instances

This section presents simple modifications to Haskell's module system that are necessary to allow instances to have a modular scope [14] (we do not attempt to discuss any major revision to Haskell's module system). Essentially, import and export clauses can specify, instead of just names, also `instance` A $\overline{\tau}$, where

$\overline{\tau}$ is a (non-empty) sequence of types and A is a class name:

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

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

`import` M (`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.

In this paper we consider that, in the case of instances without a type class, keyword `instance` must be specified, in an import clause, if there exists more than one visible instance of the overloaded name in the imported module (if there is just one visible instance in the imported module, `instance` may be specified or not); similarly, when exporting, `instance` must be specified if there exists more than one visible instance of the overloaded name in the module.

5.1. Pros and Cons of Instance Modularization

Among the advantages of having instances with modular scope, we cite:

- The types of all names, overloaded or not, and of all expressions in a module depend on the scope of variables imported by the module, and can be inferred if not explicitly annotated.
- It is possible to define and use more than one instance for the same type or for the same type constructor in a program. For example, distinct instances of *Either* for class *Monad* (say one from package *mtl* and another from *transformers*), can be used in a program.
- Use of newtypes to wrap distinct instance definitions can be avoided, if the distinct instances are not used in the same module. For example, newtypes to wrap distinct instances of Monoids for integer types, one for handling addition and one for handling multiplication.

- Modularized instance scopes and optional type classes with a revised ambiguity rule can avoid the use of functions that include additional (-by) parameters, such as e.g. the (first) parameter of function *sortBy* in module *Data.List*.

- Modularized instance scopes with a revised ambiguity rule and optional type classes can be used to support overloaded record fields in a simple way, based on the automatic creation of a type class for each overloaded record field (Section 7.3).

- The need for type-directed name resolution fades with modular instance scopes, a revised ambiguity rule and optional type classes (Section 7.2).

- Modularized instance scopes with a revised ambiguity rule and optional type classes may avoid the use of qualified imports (as used e.g. in the *classy-prelude*, used in e.g. Yesod [15]).

- Compile time consumed because the use of orphan instances can be avoided. Orphan instances are instances defined in a module where neither the definition of the data type nor the definition of the type class occur. Any instance declaration in any module imported directly or indirectly by a module M is visible in Haskell; in principle, a Haskell compiler must then read the interface files of each module imported by M, directly or indirectly, to check if it contains an instance declaration used in M, which can consume significant compile time.

However, instance modularization requires programmers 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. 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.

Consider the canonical ambiguity example in Haskell, of (*show* . *read*), but considering instances without type classes and with a modular scope:

```
module M (myshow, myread, f) where
  myshow :: Int → String
  myshow = ...
  myread :: String → Int
  myread = ...
  f = myshow . myread

module N where
  import M (myshow, myread, f)
  myshow :: Bool → String
  instance myshow = ...
  myread :: String → Bool
  instance myread = ...
  g = f "123"
```

In Haskell, any use of (*show* . *read*) leads to a type error, since the type (*Show a*, *Read a*) ⇒ *String*→ *String* is ambiguous. In our approach, the definition of f in module M is well-typed: there exists in module M a single instance of *myshow* and *myread*. As a result, f has type *String* → *String*. Its use in module N is (then) also well-typed (f 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 M), even though (*myshow* . *myread*) "123" is not well-typed in module N, since (*myshow* . *myread*) is ambiguous in N. Explicit annotation of types of imported names would make module interfaces clearer.

6. 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. Mini-Haskell extends core-Haskell

Module Name	M, N	
Program Theory	P, Q	
Variable	x, y	
Expression	e	$::= x \mid \lambda x.\, e \mid e\, e' \mid \texttt{let}\ x = e\ \texttt{in}\ e'$
Program	p	$::= \overline{m}$
Module	m	$::= \texttt{module}\, M\, (X)\ \texttt{where}\ \overline{I}; \overline{D}$
Export clause	X	$::= \overline{\iota}$
Import clause	I	$::= \texttt{import}\ M\ (X)$
Item	ι	$::= x \mid \texttt{instance}\ A\, \overline{\tau}$
Declaration	D	$::= \mathit{classDecl} \mid \mathit{instDecl} \mid B$
Class Declaration	*classDecl*	$::= \texttt{class}\ C \Rightarrow A\, \overline{a}\ \texttt{where}\ \overline{x : \delta}$
Instance Declaration	*instDecl*	$::= \texttt{instance}\ C \Rightarrow A\, \overline{\tau}\ \texttt{where}\ \overline{B} \mid$ $\texttt{instance}\ B$
Binding	B	$::= x = e$

Figure 3: Context-free syntax of mini-Haskell

expressions (Subsection 6.1) with class and instance declarations, allowing type classes to be optionally declared, and with modules, that can export and import names and instances (Subsection 6.2).

Figure 3 shows the context-free syntax of mini-Haskell: expressions, modules and programs. An instance can be specified without a type class, cf. second option (after `|`) in Instance Declaration in Figure 3.

For simplicity, 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.

MPTCs are supported. In this paper 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 given for quantified constraints (defined in Figure 1).

Typing contexts are indexed by module names. $\Gamma(M)$ gives a function on variable names to types: $\Gamma(M)(x)$ gives the type of x in module M and typing context Γ. The notation $(\Gamma(M), x \mapsto \sigma)$ is used to denote the typing context Γ' that differs from Γ only by mapping x to σ in module M, i.e. : $\Gamma'(M')(x') = \sigma$ if $M' = M$ and $x' = x$, otherwise $\Gamma'(M')(x') = \Gamma(M')(x')$. To avoid introducing a name for instance definitions, the domain of $\Gamma(M)$ (for any module M) can consist of not only normal variable names but also items of the form `instance` $C \Rightarrow A\,\tau$ (where C is a constraint set, A a class name, τ a simple type).

An empty module name, denoted by `[]`, is used for a module of exported names, to control the scope of names in import and export clauses. The reserved name (`self`) is used to refer to the current module, used in the type system and relations to control the scope of names in import and export clauses.

It is not necessary to store 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 system guarantees that each member is an instance of the type declared in the type class; otherwise (i.e. it is the single member 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. We define $st(\Gamma, M, x)$ below to yield the singleton set containing a fresh instance of the type of x in $\Gamma(M)$, if $x \in dom(\Gamma(M))$, where we consider $freshst(\sigma_x)$ yields the simple type in σ_x with type variables renamed to be fresh (for example, $freshst(\forall\,\alpha.\,\alpha)$ yields a fresh type variable α'):

$$st(\Gamma, M, x) = \begin{cases} \emptyset & \text{if } x \notin dom(\Gamma(M)) \\ \{\tau_x\} & \text{otherwise, where } \Gamma(M)(x) = \sigma_x, \tau_x = freshst(\sigma_x) \end{cases}$$

$$\frac{\Gamma(\texttt{self})(x) = (\forall\,\overline{a}.\,C \Rightarrow \tau) \qquad P \vdash_e \phi\,C \qquad dom(\phi) \subseteq \overline{a}}{P;\Gamma \vdash x : \phi(C \Rightarrow \tau)}\ (\textsc{var})$$

$$\frac{P;(\Gamma(\texttt{self}), x \mapsto \tau) \vdash e : C \Rightarrow \tau'}{P;\Gamma \vdash \lambda x.\,e : C \Rightarrow \tau \rightarrow \tau'}\ (\textsc{abs})$$

$$\frac{\begin{array}{c} P;\Gamma \vdash e : C \Rightarrow \tau' \rightarrow \tau \qquad P;\Gamma \vdash e' : C' \Rightarrow \tau' \\ V = tv(\tau) \cup tv(C) \qquad (C \oplus_V C') \gg_P C'' \end{array}}{P;\Gamma \vdash e\,e' : C'' \Rightarrow \tau}\ (\textsc{app})$$

$$\frac{\begin{array}{ll} P;\Gamma \vdash e\!: C \Rightarrow \tau & C \gg_P C'' \\ gen(C'' \Rightarrow \tau, \sigma, tv(\Gamma)) & P;(\Gamma(\texttt{self}), x \mapsto \sigma) \vdash e'\!:\ C' \Rightarrow \tau' \end{array}}{P;\Gamma \vdash \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e' : C' \Rightarrow \tau'}\ (\textsc{let})$$

Figure 4: Core-Haskell Type System

6.1. Core-Haskell

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

The type system uses entailment of a set of constraints C by a program theory P, written as $P \vdash_e C$. Entailment is defined in Subsection 6.4. The type system uses also the constraint set simplification relation, $\gg_P$, which is defined as a composition of the improvement and context reduction relations, defined respectively in 6.5.1 and 6.5.2.

Improvement is also defined in terms of constraint set entailment. It is simply a process of removing the subset of constraints for which overloading is resolved and there exists a single substitution that entails the resolved constraint. In Subsection 8.1 we define constraint set satisfiability, the functional counterpart of the entailment relation.

Rules (VAR) and (ABS) are standard. Rule (VAR) enables constrained types to be derived for a variable, by instantiation of possibly polymorphic constrained types, requiring that instantation yields entailed constraints in the program theory.

Rule (LET) performs constraint set simplification before type generalization.

We define that $gen(\delta, \sigma, V)$ holds if $\sigma = \forall\,\overline{a}.\,\delta$, where $\overline{a} = tv(\delta) - V$; similarly, for constraints, $gen(C \Rightarrow \pi, \theta, V)$ is defined to hold if $\theta = \forall\,\overline{a}.\,C \Rightarrow \pi$, where $\overline{a} = tv(C \Rightarrow \pi) - V$.

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

$$C \oplus_V C' = C \cup \{\pi \in C' \mid tv(\pi) \cap reachableVs(C', V) \neq \emptyset\}$$

A constraint set C' can be removed from a constrained type $C, C' \Rightarrow \tau$ if and only if overloading for C' has been resolved and there exists a single entailing substitution for C' (cf. Section 6.4).

In rule (APP), the use of $\gg_P$ allows constraints on the type of the result to be those that occur in the function type plus 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 (cf. Definition 1). This allows e.g. to eliminate constraints on the type of the following expressions, where o is any expression, with a possibly non-empty set of constraints on its type: *flip const o* (where *const* has type $\forall a, b.\, a \rightarrow b \rightarrow a$ and *flip* has type $\forall a, b, c.\, (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$), which should denote an identity function, and *fst* $(e,\, o)$, which should have the same denotation as e.

We have the following:

Theorem 5 (Substituition)**.** *For all $P; \Gamma \vdash e : C \Rightarrow \tau$ and all substitutions ϕ such that $P \vdash_e \phi C$ holds, we have that $P; \phi\Gamma \vdash e : \phi(C \Rightarrow \tau)$ holds.*

Proof: By a straightforward induction on the structure of e. $\square$.

6.2. Mini-Haskell

Mini-Haskell extends core-Haskell with declarations of modules (Figure 5), import clauses for instances (Figure 6) and declarations of classes, instances and non-overloaded names (Figure 7).

Rule (MOD), in Figure 5, uses relations ($\vdash_{\Downarrow}$) and ($\vdash^{X}_{\Uparrow}$), which are defined separately, for clarity, in Figures 6 and 7).

The import relation $\Gamma \vdash_{\Downarrow} \overline{I} : \Gamma'$ yields a typing context (Γ') from a typing context (Γ) and a sequence of import clauses ($\overline{I}$). It inserts in the scope of the importing module pairs of variable names and their types, that occur in module `[]`, the module of exported names.

Relation $P; \Gamma \vdash^{X}_{\Uparrow} \overline{D} : (P', \Gamma')$ is used for specifying the types of a sequence of bindings, from a typing context (Γ) and a program theory (P); it yields a new typing context (Γ'), so that $\Gamma'(\texttt{[]})$ contains the types of exported names, and a new program theory (P'), updated from class and instance declarations. Relation ($\vdash$) 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 with variables of predefined library modules with their types. Recursive modules are not treated in this paper.

The first and second rules in Figure 7 specify the bindings generated by standard Haskell type classes and instance declarations, respectively. For simplicity, we omit special rules for validity of type class and instance declarations (see [7]), that are not relevant here (for example, that the class hierachy is acyclic).

If an instance with constraints is exported, it is not necessary to insert the constraints for exporting it. The constraints are checked and recovered if necessary when the instance is used.

For example, it is necessary to include `instance` Eq `[`a`]` in an export list of a module that defines:

$$\frac{\Gamma_0 \vdash_{\Downarrow} \bar{I} : \Gamma \qquad P; \Gamma \vdash^{X}_{\Uparrow} \overline{D} : (P', \Gamma')}{P; \Gamma_0 \vdash \texttt{module}\ M\,(X)\ \texttt{where}\ \bar{I}; \overline{D} : (P', \Gamma')}\ (\texttt{MOD})$$

Figure 5: Mini-Haskell module rule

$$\frac{\Gamma'(M)(x) = \begin{cases} \Gamma(\texttt{[]})(x) & \text{if} M = \texttt{self} \text{ and, for some } 1 \leq k \leq n, \\ & x = \iota_k \text{ or } (\iota_k = \texttt{instance}\, A\, \overline{\tau},\ x \text{ member of class } A) \\ \Gamma(M)(x) & \text{otherwise} \end{cases}}{\Gamma \vdash_{\Downarrow} \texttt{import}\ M\,(\,\overline{\iota}^{\,n}\,) : \Gamma'}$$

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

Figure 6: Import relation

```
instance Eq a ⇒ Eq [a] where ...
```

The third rule accounts for instance declarations of an overloaded symbol x whose type is not explicitly specified in a type class. As stated previously, the type τ' of x is the least common generalization of the set of types $\{\tau\} \cup \{\Gamma_0(\texttt{self})(x)\}$, where τ is the type of the expression in the current instance declaration for x and $\Gamma_0(\texttt{self})(x)$ is the type of x in the current type environment (previously computed from other instance declarations for x that are visible in Γ_0). This rule is based on Theorem 3.

In this paper, we use the overloaded name also as the automatically generated class name. The overloaded name type, which is the anti-unification of visible instance types of the overloaded name in the current module, is used as the initial constraint introduced due to the use of the overloaded name.

The fourth rule is similar to rule (LET); it defines how the typing context is updated upon a declaration of a non-overloaded name.

6.3. Use of type system rules

This section presents examples that illustrate the use of Mini-Haskell rules for declarations.

Consider the example of Figure 6.3, where e_1 is assumed to be an expression of type *Char* $\rightarrow$ *Int* and e_2 an expression of type *Int* $\rightarrow$ *Bool*.

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

$$\frac{P \vdash_e \phi(C \Rightarrow \pi) \quad \mathit{gen}(\phi(C \Rightarrow \pi), \theta, \mathit{tv}(\Gamma)) \quad Q = P \cup \{\theta\} \\ Q;\Gamma \vdash e_i : \delta_i \quad \delta_i = \phi(\Gamma(\texttt{[]})(x_i))),\ \text{for } i = 1,\ldots,n \\ Q;\Gamma \vdash^X_{\Uparrow} \overline{D} : (Q',\Gamma')}{P;\Gamma \vdash^X_{\Uparrow} \texttt{instance}\ \phi(C \Rightarrow \pi)\ \texttt{where}\ \overline{x = e}^n ; \overline{D} : (Q',\Gamma')}$$

$$\frac{A \text{ is the class name generated for } x \\ P;\Gamma_0 \vdash e : C \Rightarrow \tau \quad \mathit{gen}(C \Rightarrow A\,\tau, \theta, \mathit{tv}(\Gamma_0)) \quad Q = P \cup \{\theta\} \\ Q;\Gamma \vdash^X_{\Uparrow} \overline{D} : (E, Q',\Gamma') \quad \texttt{lcg}_r(\{\tau\} \cup \mathit{st}(\Gamma_0, \texttt{self}, x), \tau') \\ \Gamma(M)(y) = \begin{cases} A\,\tau' \Rightarrow \tau' & \text{if } y = x, (M = \texttt{self} \text{ or } (M = \texttt{[]}, x \in X)) \\ \Gamma_0(M)(y) & \text{otherwise} \end{cases}}{P;\Gamma_0 \vdash^X_{\Uparrow} \texttt{instance}\ x = e; \overline{D} : (Q',\Gamma')}$$

$$\frac{P;\Gamma_0 \vdash e : C \Rightarrow \tau \;\; C \gg_P C' \;\; \mathit{gen}(C' \Rightarrow \tau, \sigma, \mathit{tv}(\Gamma_0)) \;\; P;\Gamma \vdash^X_{\Uparrow} \overline{D} : (P',\Gamma') \\ \Gamma(M)(y) = \begin{cases} \sigma & \text{if } y = x, (M = \texttt{self} \text{ or } (M = \texttt{[]}, x \in X)) \\ \Gamma_0(M)(y) & \text{otherwise} \end{cases}}{P;\Gamma_0 \vdash^X_{\Uparrow} x = e; \overline{D} : (P',\Gamma')}$$

Figure 7: Mini-Haskell rules for declarations

$$
\begin{array}{l}
\texttt{module}\ A\ (\texttt{instance}\ f\ (\mathit{Char} \to \mathit{Int}),\\
\qquad\qquad \texttt{instance}\ f\ (\mathit{Int} \to \mathit{Bool}),\\
\qquad\qquad g - ::\ (f\ (a \to b)) \Rightarrow a \to b\\
\qquad\quad)\ \texttt{where}\\
\quad f :: \mathit{Char} \to \mathit{Int}\\
\quad \texttt{instance}\ f = e_1\\
\quad f :: \mathit{Int} \to \mathit{Bool}\\
\quad \texttt{instance}\ f = e_2\\
\quad g = f
\end{array}
$$

Figure 8: Example for illustrating Mini-Haskell rules for declarations

Let Γ_0 be the typing context with the (possibly empty) set of used variables of predefined library modules, with their types, and P_0 be the program theory with the set of constraints that corresponds to the (possibly empty) set of used prelude instance definitions. Let also $\overline{D}$ be the body of `module` A and:

$$X = \{\ \texttt{instance}\ f\ (\mathit{Char}\ \to \mathit{Int}), \texttt{instance}\ f\ (\mathit{Int}\ \to \mathit{Bool}),\ g\ \}$$

We have:

$$\frac{P_0; \Gamma_0 \vdash^X_{\Uparrow} \overline{D} : (P, \Gamma)}{P_0; \Gamma_0 \vdash \texttt{module}\ A\,(X)\ \texttt{where}\,\overline{D} : (P, \Gamma)}\ (\texttt{MOD})$$

where, letting $\theta_1 = f\ (\mathit{Char} \to \mathit{Int})$ and $\overline{D_1}$ be the sequence of declarations `instance` f `=` e_2`;` g `=` f (a rewritten version of last three lines in Figure 6.3):

$$\frac{\begin{array}{l}
P_0; \Gamma_0 \vdash e_1 : \mathit{Char} \to \mathit{Int} \quad \mathit{gen}(\theta_1, \theta_1, \mathit{tv}(\Gamma_0)) \quad P_1 = P_0 \cup \{\theta_1\}\\
P_1; \Gamma_1 \vdash^X_{\Uparrow} \overline{D_1} : (P, \Gamma) \quad \texttt{lcg}_r(\{\mathit{Char} \to \mathit{Int}\}, \mathit{Char} \to \mathit{Int})\\
\Gamma_1(M)(y) = \begin{cases} \mathit{Char} \to \mathit{Int} & \text{if } y = f, (M = A \text{ or } M = \texttt{[]})\\ \Gamma_0(M)(y) & \text{otherwise}\end{cases}
\end{array}}{P_0; \Gamma_0 \vdash^X_{\Uparrow} \texttt{instance}\ f{=}e_1; \overline{D_1} : (P, \Gamma)}$$

The derivation of $P_1; \Gamma_1 \vdash^X_{\Uparrow} \overline{D_1} : (P, \Gamma)$ is shown below, letting $\theta_2 = f\,(Int \to Bool)$ and using $\mathtt{lcg}_r(\{Int \to Bool, Char \to Int\}, a \to b)$, where a and b are fresh type variables:

$$\frac{\begin{array}{l} P_1; \Gamma_1 \vdash e_2 : Int \to Bool \quad gen(\theta_2, \theta_2, tv(\Gamma_1)) \quad P = P_1 \cup \{\theta_2\} \\ P; \Gamma_2 \vdash^X_{\Uparrow} g = f : (P, \Gamma) \quad \mathtt{lcg}_r(\{Int \to Bool, Char \to Int\}, a \to b) \text{ where } a, b \text{ fresh} \\ \Gamma_2(M)(y) = \begin{cases} f\,(a \to b) \Rightarrow a \to b & \text{if } y = f, (M = A \text{ or } M = \texttt{[]}) \\ \Gamma_1(M)(y) & \text{otherwise} \end{cases} \end{array}}{P_1; \Gamma_1 \vdash \overline{D_1} : (P, \Gamma)}$$

We have, also, where $\sigma = \forall a, b.\, f\,(a \to b) \Rightarrow (a \to b)$:

$$\frac{\begin{array}{l} P; \Gamma_2 \vdash f : f\,(a \to b) \Rightarrow (a \to b) \quad gen(f\,(a \to b) \Rightarrow (a \to b), \sigma, tv(\Gamma_2)) \\ \Gamma(M)(y) = \begin{cases} \sigma & \text{if } y = f, (M = A \text{ or } M = \texttt{[]}) \\ \Gamma_2(M)(y) & \text{otherwise} \end{cases} \end{array}}{P; \Gamma_2 \vdash g = f : (P, \Gamma)}$$

6.4. Entailment

We define in this appendix constraint set provability, called entailment in Haskell terminology. Entailment of a set of constraints is defined with respect to the set of class and instance declarations that occur in a program, a so-called program theory (cf. [16]).

Definition 2. *A program theory P is a set of axioms of first-order logic generated from class and instance declarations occurring in the program, as follows (where $C \Rightarrow \pi$ is considered syntactically equivalent to π if C is empty):*

- *For each class declaration* `class` $C \Rightarrow TC\ \overline{a}$ `where ...` *the program theory contains the following formula if C is not empty: $\forall\, \overline{a}.\, C \Rightarrow TC\ \overline{a}$.*
- *For each instance declaration* `instance` $C \Rightarrow TC\ \overline{t}$ `where ...` *the program theory contains the following formula: $\forall\, \overline{a}.\, C \Rightarrow TC\ \overline{t}$, where $\overline{a} = tv(\overline{t}) \cup tv(C)$; if C is empty, then the instance declaration is of the*

$$\frac{}{P \vdash_e \emptyset}(\texttt{ent}_0) \qquad \frac{(\forall\,\overline{a}.\,C \Rightarrow \pi) \in P}{P \vdash_e \{(C \Rightarrow \pi)[\overline{a} \mapsto \overline{\tau}]\}}(\texttt{inst}_0)$$

$$\frac{P \vdash_e C \quad P \vdash_e \{C \Rightarrow \pi\}}{P \vdash_e \{\pi\}}(\texttt{mp}_0) \quad \frac{P \vdash_e C \quad P \vdash_e D}{P \vdash_e C \cup D}(\texttt{conj}_0)$$

Figure 9: Constraint Set Entailment

form `instance` $TC\ \bar{t}$ `where ...` *and the program theory contains the formula:* $\forall\,\overline{a}.\ TC\ \bar{t}$.

The property that a set of constraints C is entailed by a program theory P, written as $P \vdash_e C$, is defined in Figure 9. Following [17, 18], entailment is obtained from quantified constraints contained in a program theory P.

Definition 3 (Entailed instances and Entailing Substitutions)**.** $\lfloor C \rfloor_P$ is the set of *entailed instances* of constraint set C with respect to program theory P:

$$\lfloor C \rfloor_P = \{\, \phi(C) \mid P \vdash_e \phi(C)\,\}$$

and the corresponding substitutions as *entailing substitutions*:

$$\mathit{entailingSubs}(C, P) = \{\, \phi \mid P \vdash_e \phi(C)\,\}$$

Example 4. As an example, consider:

$$P = \{\forall a, b.\, D\,a\,b \Rightarrow C\ \texttt{[}a\texttt{]}\ b, D\,\mathit{Bool}\ \texttt{[}\mathit{Bool}\texttt{]}\}$$

We have that $\lfloor C\,a\,a \rfloor_P = \lfloor C\,\texttt{[}\mathit{Bool}\texttt{]}\ \texttt{[}\mathit{Bool}\texttt{]} \rfloor_P$. Both constraints $D\,\mathit{Bool}\ [\mathit{Bool}] \Rightarrow C\,\texttt{[}\mathit{Bool}\texttt{]}\ \texttt{[}\mathit{Bool}\texttt{]}$ and $C\,\texttt{[}\mathit{Bool}\texttt{]}\ \texttt{[}\mathit{Bool}\texttt{]}$ are members of $\lfloor C\,a\,a \rfloor_P$ and also members of $\lfloor C\ \texttt{[}\mathit{Bool}\texttt{]}\ \texttt{[}\mathit{Bool}\texttt{]} \rfloor_P$.

A proof that $P \vdash_e \{C\,\texttt{[}\mathit{Bool}\texttt{]}\ \texttt{[}\mathit{Bool}\texttt{]}\}$ holds can be given from the entailment rules given in Figure 9, since this is the conclusion of rule ($\texttt{mp}_0$) with premises $P \vdash_e \{D\,\mathit{Bool}\,\texttt{[}\mathit{Bool}\texttt{]}\}$ and $P \vdash_e \{D\,\mathit{Bool}\ [\mathit{Bool}] \Rightarrow C\,\texttt{[}\mathit{Bool}\texttt{]}\ \texttt{[}\mathit{Bool}\texttt{]}\}$, and these two premises can be derived by using rule ($\texttt{inst}_0$).

Equality of constraint sets is considered modulo type variable renaming. That is, constraint sets C, D are also equal if there exists a renaming substitution ϕ that can be applied to C to make $\phi\,C$ and D equal.

$$\frac{C \vdash^{P}_{\mathtt{impr}} C' \quad C' \vdash^{P}_{\mathtt{red}} D}{C \gg_P D}$$

Figure 10: Constraint set simplification

$$\frac{\begin{array}{l} C' = \{\pi \mid tv(\pi) \subseteq \mathit{unReachVs}(C, tv(\tau))\} \\ \mathit{entailingSubs}(C', P) = \{\phi\} \end{array}}{C \Rightarrow \tau \vdash^{P}_{\mathtt{impr}} (C - C') \Rightarrow \tau}$$

Figure 11: Constraint Set Improvement

ϕ is a renaming substitution if for all $a \in \mathit{dom}(()S)$ we have that $\phi(a) = b$, for some type variable $b \notin \mathit{dom}(\phi)$.

6.5. Constraint-set Simplification

Relation $\gg_P$ is a simplification relation on constraints, defined as a composition of improvement and context reduction, defined respectively in Subsections 6.5.1 and 6.5.2.

6.5.1. Improvement

Improvement removes constraints with unreachable type variables from a constraint C that occurs on a constrained type $C \Rightarrow \tau$, based on constraint set entailment: improvement consists of removing each constraint in C that has unreachable type variables and for which there exists a single entailing substitution. Improvement is defined in Figure 11. If the set $\mathbb{S}$ of entailed instances of $\mathit{unReachVs}(C, tv(\tau))$ has more than one element, or if it is empty, there is no improved constraint (improvement is a partial relation).

6.5.2. Context Reduction

Context reduction is a process that reduces a constraint π into constraint set D according to a *matching instance* for π in a program theory P: if there exists $(\forall\,\overline{\alpha}.\, C \Rightarrow \pi') \in P$ such that $\phi(\pi') = \pi$, for some ϕ such that $\phi(C)$ reduces to

D; if there is no matching instance for π or no reduction of $\phi(C)$ is possible, then π reduces to a constraint set containing only itself.

As an example of a context reduction, consider an instance declaration that introduces $\forall a.\, Eq\, a \Rightarrow Eq\,\texttt{[}a\texttt{]}$ in program theory P; then $Eq\,\texttt{[}a\texttt{]}$ is reduced to $Eq\ a$.

Context reduction can also occur due to the presence of superclass class declarations, but we only consider the case of instance declarations in this paper, which is the more complex process. The treatment of reducing constraints due to the existence of superclasses is standard; see e.g. [2, 19, 20].

Context reduction uses *matches*, defined as follows:

$$\begin{array}{l} \mathit{matches}\big(\pi, (P, \Phi'), \Delta\big) \text{ holds if} \\ \quad \Delta = \left\{ (\phi(C_0), \pi_0, \Phi') \;\middle|\; \begin{array}{l} (\forall\, \overline{\alpha}.\, C_0 \Rightarrow \pi_0) \in P, \\ \mathit{mgm}(\pi_0 = \pi, \phi),\ \Phi' = \Phi[\pi_0, \pi] \end{array} \right\} \end{array}$$

where *mgm* is analogous to *mgu* but denotes the most general matching substitution, instead of the most general unifier.

The third parameter of *matches* is either empty or a singleton set, since overlapping instances [1] are not considered.

Context reduction, defined in Figure 12, uses rules of the form $C \vdash^{P,\Phi}_{\texttt{red}} D; \Phi'$, meaning that either C reduces to D under program theory P and least constraint value function Φ, causing Φ to be updated to Φ', or $C \vdash^{P,\mathit{Fail}}_{\texttt{red}} C; \mathit{Fail}$. Failure is used to define a reduction of a constraint set to itself.

The least constraint value function is used as in the definition of *sats* to guarantee that context reduction is a decidable relation.

An empty constraint set reduces to itself (`red`). Rule (`conj`) specifies that constraint set simplification works, unlike constraint set satisfiability, by performing a union of the result of simplifying separately each constraint in the constraint set. To see that a rule similar to (`conj`) cannot be used in the case of constraint set satisfiability, consider a simple example, of satisfiability of $C = \{A\ a, B\ a\}$ in $P = \{A\ \mathit{Int}, A\ \mathit{Bool}, B\ \mathit{Int}, B\ \mathit{Char}\}$. Satisfiability of C yields a single substitution where a maps to Int, not the union of computing

$$\frac{}{\emptyset \vdash^{P,\Phi}_{\texttt{red}} \emptyset; \Phi}(\texttt{red}) \qquad \frac{\{\pi\} \vdash^{P,\Phi}_{\texttt{red}} C; \Phi_1 \; D \vdash^{P,\Phi_1}_{\texttt{red}} D'; \Phi'}{\{\pi\} \cup D \vdash^{P,\Phi}_{\texttt{red}} C \cup D'; \Phi'}(\texttt{conj})$$

$$\frac{\mathit{matches}\big(\pi, (P,\Phi), \{(C, \pi', \Phi')\}\big) \quad C \vdash^{P,\Phi'}_{\texttt{red}} D; \Phi''}{\{\pi\} \vdash^{P,\Phi}_{\texttt{red}} D; \Phi''}\ (\texttt{inst})$$

$$\frac{\mathit{matches}\big(\pi, (P,\Phi), \{(C, \pi', \Phi')\}\big) \quad C \vdash^{P,\Phi'}_{\texttt{red}} D; \mathit{Fail}}{\{\pi\} \vdash^{P,\Phi}_{\texttt{red}} \{\pi\}; \mathit{Fail}}\ (\texttt{stop}_0)$$

$$\frac{\mathit{matches}\big(\pi, (P,\Phi), \{(C, \pi', \mathit{Fail})\}\big)}{\{\pi\} \cup C \vdash^{P,\Phi}_{\texttt{red}} \{\pi\} \cup C; \mathit{Fail}}\ (\texttt{stop})$$

Figure 12: Context Reduction

satisfiability for $A\,a$ and $B\,a$ separately.

Rule (`inst`) specifies that if there exists a constraint axiom $\forall\,\overline{\alpha}.\,C \Rightarrow A\,\overline{\tau}$, such that $A\,\overline{\tau}$ matches with an input constraint π, then π reduces to any constraint set D that C reduces to.

Rules ($\texttt{stop}_0$) and (`stop`) deal with failure due to updating of the constraint-head-value function.

7. Examples

This section illustrates the use of polymonads (Subsection 7.1) and, together with overloading without type classes, type directed name resolution (Subsection 7.2) and overloaded record fields (Subsection 7.3).

7.1. Polymonads

Polymonads [21, 22], a generalization of monads, give the familiar monadic bind combinator the more general type:

```
(>>=) :: m1 a → (a → m2 b) → m3 b
```

This allows the composition of computations with three different monads, instead of just one. Consider:

```
module P (instance PolyMonad m1 m2 m3) where
  class (Monad m1,Monad m2) ⇒ Morph m1 m2 where
    morph:: m1 a → m2 a
  class PolyMonad m1 m2 m3 where
    (|>>=|):: m1 a → (a → m2 b) → m3 b
  instance (Morph m1 m3,Morph m2 m3) ⇒ PolyMonad m1 m2 m3 where
    ma |>>=| fmb = morph ma >>= morph . fmb
```

A module can use the polymonad instance from module P by defining instances of *Morph* for type constructors that are instances of *m1*, *m2* and *m3*. It can also import P to export q, defined as follows, which has a constrained type with a type variable, *m3*, that occurs only in the constraints:

```
module Q (q)
  import P (instance PolyMonad a b c)
  q::  (PolyMonad m1 m2 m3, PolyMonad m3 m4 m5) ⇒
          m1 a → (a → m2 b) → (b → m4 c) → m5 c
  q m f g = m |>>=| f |>>=| g
```

Each use of q in a module U for specific ground instances of *Monad m1* and *Monad m2*, or of specific ground instances of *Monad m4* and *Monad m5*, will require the existence of a single instance of *Monad m3*. However, this single instance can be determined since it is the same single instance of *Morph m2 m3*(in the first case), or the same single instance of *Morph m3 m4* (in the second case). Further investigation is needed, but this seems to avoid the necessity of using type checker plugins for implementing polymonads, as reported in [22].

Uses of polymonads illustrate that occurrences of variables only in constraints can occur in practice without implying ambiguity. Whenever there is a type variable in a constraint that is unreachable from the set of type variables in the simple type, there must exist only one instance that entails the constraint where this type variable occur (in this particular case, only one instance that entails the type constructor variable).

7.2. Type directed name resolution

The proposal for type directed name resolution (TDNR) [23] is similar to overloading, but uses the so-called dot notation, as in object-oriented languages, to provide an alternative way to specify which function is intended, based on the type of the value that occurs before the dot. For example, instead of:

```
module U where
  import Button (Button, reset) as B
  import Canvas (Canvas, reset) as C
  f :: Button → Canvas → IO()
  f b c = B.reset b >> C.reset c
```

TDNR proposes the use of dot notation to enable the definition of f to be:

```
f :: Button → Canvas → IO()
f b c = b.reset >> c.reset
```

Optional type classes can avoid module qualification in the definition of f and allows it to be named *reset*:

```
module U where
  import Button (Button,reset)
  import Canvas (Canvas,reset)
  reset :: (Button, Canvas) → IO()
  instance reset (b,c) = reset b >> reset c
```

The type of *reset* in module U is:

```
reset (a → IO()) => a → IO()
```

An overloaded curried instance definition:

```
reset :: Button → Canvas → IO()
instance reset b c = reset b >> reset c
```

leads to *reset* having type: *reset* $(a \to b) \Rightarrow a \to b$, where $a \to b$ can be instantiated to one of the following types:

$$
\begin{aligned}
&\mathit{Button} \to \mathit{IO}\,()\\
&\mathit{Canvas} \to \mathit{IO}\,()\\
&\mathit{Button} \to \mathit{Canvas} \to \mathit{IO}\,()
\end{aligned}
$$

7.3. Records with overloaded fields

In this section we describe how the possibility of overloading symbols without the need of declaring type classes allows an easy support for overloaded record fields. Acccessing of an overloaded record field simply leads to the use of an automatic created instance of an undeclared type class, and similarly for any update of an overloaded record field.

We illustrate this idea below by creating an instance of *get_fieldname* and *update_fieldname* whenever there exists, respectively, an access of and an update to an overloaded record field, where *fieldname* is the name of the overloaded record field.

Consider the following simple example of overloaded record fields:

```
data Person  = Person  { id :: Int, name    :: String }
data Address = Address { id :: Int, address :: String }
```

The overloaded *id* fields of types *Person* and *Address* have types:

```
id :: Person  → Int
id :: Address → Int
```

The following instance declarations without declared type classes can be automatically created:

```
get_id :: Person → Int
instance get_id (Person id _ ) = id


get_id :: Address → Int
instance get_id (Address id _ ) = id
```

If record field updating is used, updating functions are created, as illustrated below. Consider for example that record field updating is used as follows:

$update\_id$ `::` $Person \rightarrow Int \rightarrow Person$

`instance` $update\_id$ `(`$Person\ id\ name$`)` new_id `=` $Person\ new\_id\ name$

$update\_id$ `::` $Address \rightarrow Int \rightarrow Address$

`instance` $update\_id$ `(`$Address\ id\ address$`)` new_id `=` $Address\ new\_id\ address$

Given any expression p of type $Person$, any use of `(`$p\ \{id$ `=` $new\_id\}$`)` could then be translated to `(`$update\_id\ p\ new\_id$`)`. Similarly, given any expression a of type $Address$, any use of $a\ \{id$ `=` $new\_id\}$ could then be translated to $update\_id\ a\ new\_id$.

8. Type inference

This section presents a type inference algorithm for mini-Haskell, and presents theorems of soundness and principal type with respect to the type system.

For this, functional counterparts of relations used in the type system (Section 6) need to be defined. The main one is satisfiability, the counterpart of entailment (Subsection 8.1). We do not consider in this paper algorithms for constraint set simplification; interested readers may consult e.g. [24]. In an abuse of notation the same symbols are used in this section for functions corresponding to the relations of constraint set simplification ($\gg_P$, where P is a program theory) and type generalization (gen).

Partial orders on types, constraints, substitutions, and typing contexts with program theories are defined in Figure 13. Type ordering disregards constraint set entailment, which is important only for considering whether a constraint π can be removed from a constraint C occurring in a constrained type $C \Rightarrow \tau$; π can be removed if and only if overloading for π has been resolved and there exists a single entailing substitution for C, as defined in Figure 11, page 26.

A type inference algorithm for core-Haskell is presented in Figure 14, using rules of the form $P; \Gamma \vdash_i e : (\delta, \phi)$, which means that δ is the least (principal)

$$\frac{}{\sigma \leq \phi\,\sigma} \qquad\qquad \frac{}{\pi \leq \phi\,\pi}$$

$$\frac{\Gamma(x) \leq \Gamma'(x) \text{ for all } x \in \mathit{dom}(\Gamma)}{P;\Gamma \leq P;\Gamma'}$$

$$\frac{\text{there exists } \phi_1 \text{ such that } \phi = \phi_1 \circ \phi'}{\phi \leq \phi'}$$

Figure 13: Partial orders

type derivable for e in typing context $\phi\Gamma$ and program theory P, where $\phi\Gamma \leq \Gamma$ and, whenever $\Gamma' \leq \Gamma$ is such that $P;\Gamma' \vdash_i e : (\delta', \phi')$, we have that $\phi\Gamma \leq \Gamma'$ and $\delta' \leq \phi\delta$ hold. Furthermore, we have that $P;\phi\,\Gamma \vdash_i e : (\delta, \phi')$ holds whenever $P;\Gamma \vdash_i e : (\delta, \phi)$ holds, where $\phi' \leq \phi$ (cf. theorem 6 below).

Example 5. Consider expression x and typing context $\Gamma = \{f : \mathit{Int} \to \mathit{Int}, x : \alpha\}$; we can derive $\Gamma \vdash_i f\ x : (\mathit{Int}, \phi)$, where $\phi = [\alpha \mapsto \mathit{Int}]$. From $\phi\,\Gamma = \{f : \mathit{Int} \to \mathit{Int}, x : \mathit{Int}\}$, we can derive $\phi\,\Gamma \vdash_i f\ x : (\mathit{Int}, \mathit{id})$.

Theorem 6. *If $P;\Gamma \vdash_i e : (\delta, \phi)$ holds then $P;\phi\,\Gamma \vdash_i e : (\delta, \phi')$ holds, where $\phi' \leq \phi$.*

Relation *mgu* is the most general (least) unifier relation [25]: $\mathit{mgu}(\mathcal{T}, \phi)$ is defined to hold between a set of pairs of simple types $\mathcal{T}$ and a substitution ϕ if i) ϕ is a unifier of every pair in $\mathcal{T}$ (i.e. $\phi\tau = \phi\tau'$ for every $(\tau, \tau') \in \mathcal{T}$), and ii) it is the least such unifier (i.e. if ϕ' is a unifier of all pairs in $\mathcal{T}$, then $\phi \leq \phi'$). The relation holds similarly for constraints instead of types.

mgu_I is a function that gives a most general unifier of a set of pairs of simple types (or simple constraints). We define also that $\phi = \mathit{mgu}_I(\tau = \tau')$ is an alternative notation for $\phi = \mathit{mgu}_I(\{(\tau, \tau')\})$. We have:

Theorem 7 (Soundness)**.** *If $P;\Gamma \vdash_i e : (\delta, \phi)$ holds then $P;\phi\Gamma \vdash e : \delta$ holds.*

Theorem 8 (Principal type)**.** *If $P;\Gamma \vdash_i e : (\delta, \phi)$ holds then, for all δ' such that $P;\phi\Gamma \vdash e : \delta'$ holds, we have that $\delta \leq \delta'$.*

A completeness theorem does not hold. The canonical Haskell ambiguity example of expression $e_0 = (\mathit{show}\ .\mathit{read})$ (where *show* has type $\mathit{Show}\ a \Rightarrow a \to$

$$\frac{(\Gamma(\texttt{self})(x) = \forall\,\overline{a}.\,\delta) \in \Gamma \quad \overline{b}\text{fresh}}{P;\Gamma \vdash_i x : (\delta[\overline{a} \mapsto \overline{b}], id)}\,(\texttt{VAR}_i)$$

$$\frac{P;(\Gamma, x : a) \vdash_i e : (C \Rightarrow \tau, \phi) \quad a\,\text{fresh} \quad \tau' = \phi\, a}{P;\Gamma \vdash_i \lambda x.\, e : (C \Rightarrow \tau' \to \tau, \phi)}\,(\texttt{ABS}_i)$$

$$\frac{\begin{array}{ll} P;\Gamma \vdash_i e : (C \Rightarrow \tau_1, \phi_1) & P;\phi_1\Gamma \vdash_i e' : (C' \Rightarrow \tau_2, \phi_2) \\ \phi' = mgu_I(\tau_1 = \tau_2 \to a) & a\,\text{fresh},\ \phi = \phi' \circ \phi_2 \circ \phi_1 \\ \tau = \phi\, a,\ V = tv(\tau) \cup tv(\phi C) & (\phi C \oplus_V\ \phi C') \gg_{P_\Gamma} D \end{array}}{\Gamma \vdash_i e\, e' : (D \Rightarrow \tau, \phi)}\,(\texttt{APP}_i)$$

$$\frac{\begin{array}{ll} \Gamma \vdash_i e\!: (C \Rightarrow \tau, \phi_1) & C \gg_{P_\Gamma} C' \\ gen(\sigma, C' \Rightarrow \tau, tv(\phi_1\Gamma)) & \phi_1\Gamma,\ x\!:\!\sigma \vdash_i e_2\!: (\delta, \phi) \end{array}}{\Gamma \vdash_i \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e' : (\delta, \phi)}\,(\texttt{LET}_i)$$

Figure 14: Type Inference

String and *read* has type *Read* $a \Rightarrow$ *String*$\to a$) is such that, whenever there exist two or more entailing instances for *Show* a and *Read* a, for some a, there exists P and Γ such that $P;\Gamma \vdash e_0 : \mathit{String} \to \mathit{String}$ holds, but there is no δ, ϕ such that $P;\Gamma \vdash e_0 : (\delta, \phi)$ holds.

The greater simplicity obtained by allowing type instantiation to occur in a context-independent way, in a type system for a language with support for context-dependent overloading, has significant counterparts. The disadvantages are that ambiguous expressions are allowed to be well-typed, and there exist several translations for expressions, one of them a principal translation, for a semantics defined inductively on the type system rules.

A declarative specification of type inference, with a unique type derivable for each expression, where type instantiation is restricted to be done only in a context-dependent way, defined by considering functions used in the type inference algorithm as relations, is a possible alternative. In this case, the type inference algorithm is obtained directly from a declarative specification of the type system by transforming relations used into functions. The fact that every element has a unique type is consonant with everyday spoken language. It is straightforward to define, a posteriori, the set of types that are valid instances of the type of an expression. Also, the fact that only a single type can be derived for each expression rules out the possibility of having distinct derivations for an expression's type. Thus, an error message for an expression such as (*show* . *read*), in a context with more than one instance for (*Show* a) and (*Read* a), for some a, whose type is *Show* a, *Read* $a \Rightarrow a$), should be that the expression can not be given a well-defined semantics: distinct meanings of (*show* . *read*) would be obtained from distinct instance types of *show* and *read*.

Type inference for mini-Haskell is obtained by extending type inference for core-Haskell straightforwardly, namely by directly using the rules of Subsection 6.2 (Figures 5, 6 and 7), replacing relations by functions.

The overloading resolution theorem below considers a type inference algorithm that differs from mini-Haskell's by only i) disregarding improvement, and thus not removing any *resolved* constraint, and ii) not allowing constraints in an argument to be removed, using $C \cup C'$ instead of $C \oplus_V C'$ (where $V = tv(\tau) \cup tv(C)$). We use $\vdash_{i1}$ instead of $\vdash_i$ in typing formulas of this (mini-Haskell without improvement and constraint selection by $\oplus$) type inference algorithm. We also define a program context $C[e]$ as any expression that has e as a subexpression.

Theorem 9 (Overloading Resolution). *For all P, Γ, e, C, τ such that $P; \Gamma \vdash_{i1} e : C \Rightarrow \tau$ holds, $\pi \in C$ and $a \in unReachVs(\{\pi\}, tv(\phi(\tau)))$, then, for all $P; \phi\Gamma \vdash_{i1} C[e] : C' \Rightarrow \tau'$ that holds, we have that $\pi \in C'$.*

Proof: By induction on the structure of $C[e]$. $\square$

Informally speaking, theorem 9 shows that there is no program context where an expression can be used that will cause a constraint with an unreachable type variable to be instantiated.

8.1. Satisfiability

This subsection contains a description of constraint set satisfiability, including a discussion of decidability (taken from [26]). Constraint set satisfiability is in general an undecidable problem [27]. It is restricted here so that it becomes decidable, as described below. The restriction is based on a measure of constraints, a measure of the sizes of types in a constraint head, given by a so-called constraint-head-value function. Essentially, the sequence of constraints that unify with a constraint axiom in recursive calls of the function that checks satisfiability of a type constraint is such that either the sizes of types of each constraint in this sequence is decreasing or there exists at least one type parameter position with decreasing size.

The definition of the constraint-head-value function is based on the use of a constraint value $\nu(\pi)$ that gives the number of occurrences of type variables and type constructors in π:

$$\begin{aligned}
\nu(C\,\overline{\tau}) &= \textstyle\sum_{i=1}^{n} \nu(\tau_i)\\
\nu(T) &= 1\\
\nu(\alpha) &= 1\\
\nu(\tau\,\tau') &= \nu(\tau) + \nu(\tau')
\end{aligned}$$

Consider computation of satisfiability of a given constraint set C with respect to program theory P and consider that, during the process of checking satisfiability of a constraint $\pi \in C$, a constraint π' unifies with the head of constraint $\forall\,\overline{a}.C_0 \Rightarrow \pi_0$ in P, with unifying substitution ϕ. Then, for any constraint π_1 that, in this process of checking satisfiability of π, also unifies with π_0, where the corresponding unifying substitution is ϕ_1, the following is required, for satisfiability of π to hold:

1. $\nu(\phi\,\pi')$ is less than $\nu(\phi_1\,\pi_1)$ or, if $\nu(\phi\,\pi') = \nu(\phi_1\pi_1)$, then $\phi\,\pi' \neq \pi''$, for all π'' that has the same constraint value as π' and has unified with π_0 in process of checking for satisfiability of π, or
2. $\nu(\phi\,\pi')$ is greater than $\nu(\phi_1\,\pi_1)$ but then there is a type argument position such that the number of type variables and constructors of constraints

that unify with π_0 in this argument position decreases.

More precisely, constraint-head-value-function Φ associates a pair (I, Π) to each constraint in P, where I is a tuple of constraint values and Π is a set of constraints. Let $\Phi_0(\pi_0) = (I_0, \emptyset)$ for each constraint axiom $\forall\, \overline{a}.\, P_0 \Rightarrow \pi_0 \in P$, where I_0 is a tuple of values filled with any value greater than $\nu(\pi)$ for every constraint π in the program theory; decidability is guaranteed by defining the operation $\Phi[\pi_0, \pi]$ of updating $\Phi(\pi_0) = (I, \Pi)$ as follows, where $I = (v_0, v_1, \ldots, v_n)$ and $\pi = C\, \overline{\tau}$:

$$\Phi[\pi_0, \pi] = \begin{cases} \mathit{Fail} & \text{if } v'_i = -1 \text{ for } i = 0, \ldots, n \\ \Phi' & \text{otherwise} \end{cases}$$

$$\begin{array}{lll} \text{where} & \Phi'(\pi_0) & = ((v'_0, v'_1, \ldots, v'_n), \Pi \cup \{\pi\}) \\ & \Phi'(x) & = \Phi(x) \text{ for } x \neq \pi_0 \end{array}$$

$$v'_0 = \begin{cases} \nu(\pi) & \text{if } \nu(\pi) < v_0 \text{ or} \\ & \nu(\pi) = v_0 \text{ and } \pi \notin \Pi \\ -1 & \text{otherwise} \end{cases}$$

$$\text{for } i = 1, \ldots, n \qquad v'_i = \begin{cases} \nu(\tau_i) & \text{if } \nu(\tau_i) < v_i \\ -1 & \text{otherwise} \end{cases}$$

$\mathit{sats}_1\big(\pi, P, \Delta\big)$ is defined to hold if

$$\Delta = \left\{ (\phi|_{tv(\pi)}, \phi C_0, \pi_0) \;\middle|\; \begin{array}{l} (\forall\, \overline{a}.\, C_0 \Rightarrow \pi_0) \in P, \\ \phi = mgu_I(\pi = \pi_0) \end{array} \right\}$$

The set of satisfying substitutions for C with respect to the program theory P is given by $\mathbb{S}$, such that $C \vdash^{P,\Phi_0}_{\mathtt{sats}} \mathbb{S}$ holds, as defined in Figure 15. The restriction $\phi|_V$ of ϕ to V denotes the substitution ϕ' such that $\phi'(a) = \phi(a)$ if $a \in V$, otherwise a.

The following examples illustrate the definition of constraint set satisfiability as defined in Figure 15. Let $\Phi(\pi).I$ and $\Phi(\pi).\Pi$ denote the first and second components of $\Phi(\pi)$, respectively, and v_i the i-th component of a tuple of constraint values I:

$$\frac{}{C \vdash^{P,Fail}_{\mathtt{sats}} \emptyset}(\mathtt{fail}_1) \qquad \frac{}{\emptyset \vdash^{P,\Phi}_{\mathtt{sats}} \{id\}}(\mathtt{empty}_1)$$

$$\frac{\begin{array}{l}\{\pi\} \vdash^{P,\Phi}_{\mathtt{sats}} \mathbb{S}_0 \\ \mathbb{S} = \{\phi' \circ \phi \mid \phi \in \mathbb{S}_0,\, \phi' \in \mathbb{S}_1,\, \phi(C) \vdash^{P,\Phi}_{\mathtt{sats}} \mathbb{S}_1\}\end{array}}{\{\pi\} \cup C \vdash^{P,\Phi}_{\mathtt{sats}} \mathbb{S}}(\mathtt{conj}_1)$$

$$\frac{\begin{array}{l} sats_1(\pi, P, \Delta) \\ \mathbb{S} = \left\{ \phi' \circ \phi \;\middle|\; \begin{array}{c} (\phi, D, \pi') \in \Delta,\, \phi' \in \mathbb{S}_0, \\ D \vdash^{P,\Phi[\pi',\phi\pi]}_{\mathtt{sats}} \mathbb{S}_0 \end{array} \right\}\end{array}}{\{\pi\} \vdash^{P,\Phi}_{\mathtt{sats}} \mathbb{S}}(\mathtt{inst}_1)$$

Figure 15: Decidable Constraint Set Satisfiability

Example 6. Consider satisfiability of $\pi = Eq\,\texttt{[[}I\texttt{]]}$ in $P = \{Eq\ I,\ \forall\, a.\, Eq\ a \Rightarrow Eq\,\texttt{[}a\texttt{]}\}$, letting $\pi_0 = Eq\,\texttt{[}a\texttt{]}$; we have:

$$\frac{\begin{array}{l} sats_1(\pi, P, \{(\phi|_\emptyset, \{Eq\,\texttt{[}I\texttt{]}\}, \pi_0)\}),\ \ \phi = [a_1 \mapsto \texttt{[}I\texttt{]}] \\ \mathbb{S}_0 = \{\phi_1 \circ id \mid\ \phi_1 \in \mathbb{S}_1,\ \ Eq\,\texttt{[}I\texttt{]} \vdash^{P,\Phi_1}_{\mathtt{sats}} \mathbb{S}_1\}\end{array}}{\pi \vdash^{P,\Phi_0}_{\mathtt{sats}} \mathbb{S}_0}(\mathtt{inst}_1)$$

where $\Phi_1 = \Phi_0[\pi_0, \pi]$, which implies that $\Phi_1(\pi_0) = ((3,3), \{\pi\})$, since $\nu(\pi) = 3$, and a_1 is a fresh type variable; then:

$$\frac{\begin{array}{l} sats_1(Eq\,\texttt{[}I\texttt{]}, P, \{(\phi'|_\emptyset, \{Eq\ I\}, \pi_0)\}),\ \ \phi' = [a_2 \mapsto I] \\ \mathbb{S}_1 = \{\phi_2 \circ id \mid\ \phi_2 \in \mathbb{S}_2,\ \ Eq\ I \vdash^{P,\Phi_2}_{\mathtt{sats}} \mathbb{S}_2\}\end{array}}{Eq\,\texttt{[}I\texttt{]} \vdash^{P,\Phi_1}_{\mathtt{sats}} \mathbb{S}_1}(\mathtt{inst}_1)$$

where $\Phi_2 = \Phi_1[\pi_0, Eq\,\texttt{[}I\texttt{]}]$, which implies that $\Phi_2(\pi_0) = ((2,2), \Pi_2)$, with $\Pi_2 = \{\pi, Eq\,\texttt{[}I\texttt{]}\})$, since $\nu(Eq\,\texttt{[}I\texttt{]}) = 2$ is less than $\Phi_1(\pi_0).I.v_0 = 3$; then:

$$\frac{\begin{array}{l} sats_1\big(Eq\ I, P, \{(id, \emptyset, Eq\ I)\}\big) \\ \mathbb{S}_2 = \{\phi_3 \circ id \mid\ \phi_3 \in \mathbb{S}_3,\ \ \emptyset \vdash^{P,\Phi_3}_{\mathtt{sats}} \mathbb{S}_3\}\end{array}}{Eq\ I \vdash^{P,\Phi_2}_{\mathtt{sats}} \mathbb{S}_2}(\mathtt{inst}_1)$$

where $\Phi_3 = \Phi_2[Eq\, I, Eq\, I]$ and $\mathbb{S}_3 = \{id\}$ by ($\texttt{SEmpty}_1$).

The following illustrates a case of satisfiability involving a constraint π' that unifies with a constraint head π_0 such that $\nu(\pi')$ is greater than the upper bound associated to π_0, which is the first component of $\Phi(\pi_0).I$.

Example 7. Consider satisfiability of $\pi = A\, I\, (T^3\, I)$ in program theory $P = \{A\, (T\, a)\, I, \forall\, a, b.\, A\, (T^2\, a)\, b \Rightarrow A\, a\, (T\, b)\}$. We have, where $\pi_0 = A\, a\, (T\, b)$:

$$\frac{\begin{array}{c} sats_1\big(\pi, P, \{(\phi\,|_\emptyset, \{\pi_1\}, \pi_0)\}\big) \\ \phi = [a_1 \mapsto I, b_1 \mapsto T^2\, I] \\ \pi_1 = A\,(T^2\, I)\,(T^2\, I) \\ \mathbb{S}_0 = \{\phi_1 \circ id \mid \phi_1 \in \mathbb{S}_1,\ \ \pi_1 \vdash^{P,\Phi_1}_{\texttt{sats}} \mathbb{S}_1\} \end{array}}{\pi \vdash^{P,\Phi_0}_{\texttt{sats}} \mathbb{S}_0}\,(\texttt{inst}_1)$$

where $\Phi_1 = \Phi_0[\pi_0, \pi]$, which implies that $\Phi_1(\pi_0).I = (5, 1, 4)$; then:

$$\frac{\begin{array}{c} sats_1\big(\pi_1, P, \{(\phi'\,|_\emptyset, \{\pi_2\}, \pi_0)\}\big) \\ \phi' = [a_2 \mapsto T^2\, I, b_2 \mapsto T\, I] \\ \pi_2 = A\,(T^4\, I)\,(T\, I) \\ \mathbb{S}_1 = \{\phi_2 \circ [a_1 \mapsto T^2\, a_2] \mid \phi_2 \in \mathbb{S}_2,\ \ \pi_2 \vdash^{P,\Phi_2}_{\texttt{sats}} \mathbb{S}_2\} \end{array}}{\pi_1 \vdash^{P,\Phi_1}_{\texttt{sats}} \mathbb{S}_1}\,(\texttt{inst}_1)$$

where $\Phi_2 = \Phi_1[\pi_0, \pi_1]$. Since $\nu(\pi_1) = 6 > 5 = \Phi_1(\pi_0).I.v_0$, we have that $\Phi_2(\pi_0).I = (-1, -1, 3)$.

Again, π_2 unifies with π_0, with unifying substitution $\phi' = [a_3 \mapsto T^4\, I, b_2 \mapsto I]$, and updating $\Phi_3 = \Phi_2[\pi_0, \pi_2]$ gives $\Phi_3(\pi_0).I = (-1, -1, 2)$. Satisfiability is then finally tested for $\pi_3 = A\,(T^6\, I) I$, that unifies with $A\,(T\, a)\, I$, returning $\mathbb{S}_3 = \{[a_3 \mapsto T^5\, I]|_\emptyset\} = \{id\}$. Constraint π is thus satisfiable, with $\mathbb{S}_0 = \{id\}$.

The following example illustrates a case where the information kept in the second component of $\Phi(\pi_0)$ is relevant.

Example 8. Consider the satisfiability of $\pi = A\,(T^2\, I)\, F$ in program theory $P = \{A\, I\,(T^2\, F), \forall\, a, b.\, A\, a\,(T\, b) \Rightarrow A\,(T\, a)\, b\}$ and let $\pi_0 = A\,(T\, a)\, b$. Then:

$$\frac{\begin{array}{c} sats_1(\pi, P, \{(\phi\,|_\emptyset, \{\pi_1\}, \pi_0)\}) \\ \phi = [a_1 \mapsto (T\,I), b_1 \mapsto F] \\ \pi_1 = A\,(T\,I)\,(T\,F) \\ \mathbb{S}_0 = \{\phi_1 \circ id \mid \phi_1 \in \mathbb{S}_1, \quad \pi_1 \vdash^{P,\Phi_1}_{sats} \mathbb{S}_1\} \end{array}}{\pi \vdash^{P,\Phi_0}_{sats} \mathbb{S}_0}\,(\texttt{inst}_1)$$

where $\Phi_1 = \Phi_0[\pi_0, \pi]$, giving $\Phi_1(\pi_0) = ((4,3,1), \{\pi\})$; then:

$$\frac{\begin{array}{c} sats_1(\pi_1, P, \{(\phi'\,|_\emptyset, \{\pi_2\}, \pi_0)\}) \\ \phi' = [a_2 \mapsto I, b_2 \mapsto T\,F], \quad \pi_2 = A\,I\,(T^2\,F) \\ \mathbb{S}_1 = \{\phi_2 \circ id \mid \phi_2 \in \mathbb{S}_2, \quad \pi_2 \vdash^{P,\Phi_2}_{\texttt{sats}} \mathbb{S}_2\} \end{array}}{\pi_1 \vdash^{P,\Phi_1}_{\texttt{sats}} \mathbb{S}_1}\,(\texttt{inst}_1)$$

where $\Phi_2 = \Phi_1[\pi_0, \pi_1]$. Since $\nu(\pi_1) = 4$, which is equal to the first component of $\Phi_1(\pi_0).I$, and π_1 is not in $\Phi_1(\pi_0).\Pi$, we obtain that $\mathbb{S}_2 = \{id\}$ and π is thus satisfiable (since $sats_1(A\,I\,(T^2\,F), P) = \{(id, \emptyset, A\,I\,(T^2\,F))\}$).

Since satisfiability of type class constraints is in general undecidable [27], there exist satisfiable instances which are considered to be unsatisfiable according to the definition of Figure 15. Examples can be constructed by encoding instances of solvable Post Correspondence problems by means of constraint set satisfiability, using G. Smith's scheme [27].

To prove that satisfiability as defined in Figure 15 is decidable, consider that there exist finitely many constraints in program theory P, and that, for any constraint π that unifies with π_0, we have, by the definition of $\Phi[\pi_0, \pi]$, that $\Phi(\pi_0)$ is updated so as to include a new value in its second component (otherwise $\Phi[\pi_0, \pi] = \mathit{Fail}$ and satisfiability yields $\emptyset$ as the set of satisfying substitutions for the original constraint). The conclusion follows from the fact that $\Phi(\pi_0)$ can have only finitely many distinct values, for any π_0.

9. Related Work

Principal type schemes for overloading and subtyping were studied since more than two decades [27, 28, 29]. These first works have proven undecidability of unrestrictive constraint set satisfiability (CS-SAT), by a reduction from the Post Correspondence Problem. The CS-SAT problem was firstly defined in terms of provability in the type system.

Mark Jones defined predicate entailment as a relation, viewing CS-SAT as a problem separated from the type system, but did not discuss decidability [2, 30].

CS-SAT was later defined in terms of anti-unification and least common generalisation in [31], where an algorithm for testing satisfiability used an iteration limit for termination, the limit not being reached in practical cases. CS-SAT was restricted, using a termination criterion based on a measure of the size of types in constraints, to define an algorithm for CS-SAT that always terminates [32], which has been used only in a prototype implementation (available at `http://github.com/rodrigogribeiro/mptc`).

Constraint-handling rules (CHRs) were used to describe the programming language Mercury, where context-dependent overloading is supported [12]. Open-world ambiguity and FDs were defined via CHRs in [12], constituting the basis of GHC's implementation.

Expression ambiguity for context-dependent overloading was firstly presented in [33], as a proposal for the introduction of MPTCs in Haskell without the need for FDs and type families. A comparison between open-world ambiguity and expression ambiguity is presented in [13].

Work on instance modularization with names given to instance definitions was presented in [34]. A more radical change to the module system of Haskell, more in the direction of the module system of SML, was proposed in [35]. The subject of controlling the scope of instances in Haskell is discussed in [14, 36].

10. Conclusion

This paper has presented an approach for allowing programmers to overload symbols without declaring their types in type classes. In this approach, the type of an overloaded symbol is automatically determined from the anti-unification of instance types defined for the symbol in the relevant module.

The paper explores this approach in the presence of instance modularization and an ambiguity rule that is defined differently than in Haskell.

The approach allows, for example, overloaded record fields and type directed name resolution to be supported in a simple way.

References

[1] Simon P. Jones and others, GHC — The Glasgow Haskell Compiler 7.0.4 User's Manual, `http://www.haskell.org/ghc/` (2011).

[2] Mark Jones, Qualified Types: Theory and Practice, Ph.D. thesis, Distinguished Dissertations in Computer Science. Cambridge Univ. Press (1994).

[3] Mark Jones, Type Classes with Functional Dependencies, in: Proc. of ESOP'2000, 2000, pp. 230–244, LNCS 1782.

[4] Mark Jones and Iavor Diatchki, Language and Program Design for Functional Dependencies, SIGPLAN Not. 44 (2) (2008) 87–98.

[5] Manuel Chakravarty, Gabriele Keller and Simon P. Jones, Associated Type Synonyms, in: Proc. of the Tenth ACM SIGPLAN International Conference on Functional Programming, ICFP '05, 2005, pp. 241–253.

[6] Tom Schrijvers, Simon P. Jones, Manuel Chakravarty and Martin Sulzmann, Type Checking with Open Type Functions, SIGPLAN Not. 43 (9) (2008) 51–62.

[7] Glasgow Haskell Compiler home page, `http://www.haskell.org/ghc/`.

[8] Rodrigo Ribeiro, Carlos Camarão, Lucília Figueiredo and Cristiano Vasconcellos, Optional Type Classes for Haskell — On-line repository, `https://github.com/rodrigogribeiro/mptc` (2016).

[9] C.C. Chang and H.J. Keisler, Model Theory, Dover Books on Mathematics, 2012, 3rd ed.

[10] Gordon Plotkin, A note on inductive generalisation, Machine Intelligence 5 (1) (1970) 153–163.

[11] Gordon Plotkin, A further note on inductive generalisation, Machine Intelligence 6 (1971) 101–124.

[12] Peter Stuckey and Martin Sulzmann, A Theory of Overloading, ACM Trans. Program. Lang. Syst. 27 (6) (2005) 1216–1269.

[13] Carlos Camarão, Rodrigo Ribeiro and Lucília Figueiredo, Ambiguity and Constrained Polymorphism, Science of Computer Programming 124 (1) (2016) 1–19.

[14] Marco Silva and Carlos Camarão, Controlling the Scope of Instances in Haskell, in: Proc. of SBLP'2011, 2011.

[15] Michael Snoyman, Developing Web Applications with Haskell and Yesod, O'Reilly Media, Inc., 2012.

[16] Martin Sulzmann, Gregory Duck, Simon P. Jones and Peter Stuckey, Understanding functional dependencies via constraint handling rules, Journal of Functional Programming 17 (1) (2007) 83–129.

[17] Manuel Chakravarty, Gabriele Keller, Simon P. Jones and Simon Marlow, Associated types with class, ACM SIGPLAN Notices 40 (1) (2005) 1–13.

[18] Manuel Chakravarty, Gabriele Keller and Simon P. Jones, Associated type synonyms, ACM SIGPLAN Notices 40 (9) (2005) 241–253.

[19] Cordelia Hall, Kevin Hammond, Simon P. Jones and Philip Wadler, Type classes in Haskell, ACM TOPLAS 18 (2) (1996) 109–138.

[20] Karl-Filip Faxén, A static semantics for Haskell, Journal of Functional Programming 12 (5) (2002) 295–357.

[21] Michael Hicks, Gavin M. Bierman, Nataliya Guts, Daan Leijen and Nikhil Swamy, Polymonadic Programming, in: Proc. 5th Workshop on Mathe-

matically Structured Functional Programming, MSFP@ETAPS 2014, 2014, 2014, pp. 79–99.

[22] Jan Bracker and Henrik Nilsson, Polymonad programming in haskell, in: Proc. of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL '15, 2015, 2015, pp. 3:1–3:12.

[23] Simon P. Jones, Type directed name resolution, available (July 2017) at `https://prime.haskell.org/wiki/TypeDirectedNameResolution/`.

[24] Dimitrious Vytiniotis, Simon P. Jones, Tom Schrijvers and Martin Sulzmann, OutsideIn(X): Modular Type Inference with Local Assumptions, Journal of Functional Programming 21 (4–5) (2011) 333–412.

[25] John A. Robinson, A machine oriented logic based on the resolution principle, Journal of the ACM 12 (1) (1965) 23–41.

[26] Rodrigo Ribeiro and Carlos Camarão, Ambiguity and Context-dependent Overloading, Journal of the Brazilian Computer Society 19 (3) (2013) 313–324.

[27] Geoffrey Smith, Polymorphic type inference for languages with overloading and subtyping, Ph.D. thesis, Cornell University (1991).

[28] Dennis Volpano and Geoffrey Smith, On the Complexity of ML Typability with Overloading, in: Proc. of the ACM Symposium on Functional Programming Computer Architecture., no. 523 in LNCS, 1991, pp. 15–28.

[29] Geoffrey Smith, Principal type schemes for functional programs with overloading and subtyping, Science of Computer Programming 23 (2-3) (1994) 197–226.

[30] Mark Jones, Simplifying and Improving Qualified Types, in: Proc. FPCA'95: ACM Conference on Functional Programming and Computer Architecture, 1995, pp. 160–169.

[31] Carlos Camarão, Lucília Figueiredo and Cristiano Vasconcellos, Constraint-set satisfiability for Overloading, in: ACM Press Conf. Proc. of Principles and Practice of Declarative Programming (PPDP'04), 2004, pp. 67–77.

[32] Rodrigo Ribeiro, Carlos Camarão and Lucília Figueiredo, Terminating Constraint Set Satisfiability and Simplification Algorithms for Context-Dependent Overloading, Journal of the Brazilian Computer Society 19 (4) (2013) 423–432.

[33] Carlos Camarão, Rodrigo Ribeiro, Lucília Figueiredo and Cristiano Vasconcellos, A Solution to Haskell's Multi-paramemeter Type Class Dilemma, in: Proc. of SBLP'2011, 2011, pp. 5–18.

[34] Wolfram Kahl and Jan Scheffczyk, Named Instances for Haskell Type Class, in: Proc. of 2001 ACM SIGPLAN Haskell Workshop, 2001, pp. 71–99.

[35] Derek Dreyer, Robert Harper, Manuel Chakravarty and Gabriele Keller, Modular Type Classes, SIGPLAN Not. 42 (1) (2007) 63–70.

[36] Martin Sulzmann and Meng Wang, Modular Generic Programming with Extensible Superclasses, in: Proc. of 2006 ACM SIGPLAN Workshop on Generic Programming, ACM, 2006, pp. 55–65.