J Braz Comput Soc (2013) 19:313–324
DOI 10.1007/s13173-013-0103-0

ORIGINAL PAPER

Ambiguity and context-dependent overloading

Rodrigo Ribeiro · Carlos Camarão

Received: 1 October 2012 / Accepted: 7 February 2013 / Published online: 15 March 2013

Abstract This paper discusses ambiguity in the context of languages that support context-dependent overloading, such as Haskell. A type system for a Haskell-like programming language that supports context-dependent overloading and follow the Hindley-Milner approach of providing context-free type instantiation, allows distinct derivations of the same type for ambiguous expressions. Such expressions are usually rejected by the type inference algorithm, which is thus not complete with respect to the type system. Also, Haskell's open world approach considers a definition of ambiguity that does not conform to the existence of two or more distinct type system derivations for the same type. The article presents an alternative approach, where the standard definition of ambiguity is followed. A type system is presented that allows only context-dependent type instantiation, enabling only one type to be derivable for each expression in a given typing context: the type of an expression can be instantiated only if required by the program context where the expression occurs. We define a notion of greatest instance type for each occurrence of an expression, which is used in the definition of a standard dictionary-passing semantics for core Haskell based on type system derivations, for which coherence is trivial. Type soundness is obtained as a result of disallowing all ambiguous expressions and all expressions involving unsatisfiability in the use of overloaded names. Following the standard definition of ambiguity, satisfiability is tested—i.e., "the world is closed" —if only if overloading is (or should have been) resolved, that is, if and only if there exist unreachable variables in the constraints on types of expressions. Nowadays, satisfiability is tested in Haskell, in the presence of multi-parameter type classes, only upon the presence of functional dependencies or an alternative mechanism that specifies conditions for closing the world, and that may happen when there exist or not unreachable type variables in constraints. The satisfiability trigger condition is then given automatically, by the existence of unreachable variables in constraints, and does not need to be specified by programmers, using an extra mechanism.

Keywords Ambiguity · Type systems · Semantics

R. Ribeiro (✉) · C. Camarão
Departamento de Ciência da Computação, Instituto de Ciências Exatas, Universidade Federal, de Minas Gerais, Brazil
e-mail: rodrigogribeiro@decea.ufop.br

C. Camarão
e-mail: camarao@dcc.ufmg.br

1 Introduction

Parametric polymorphism allows instantiation of quantified variables α, in quantified types $\forall\alpha.\sigma$, to *all* types $[\alpha := \tau]\sigma$, that is, every type generated from σ by replacing free occurrences of type variable α in σ by an arbitrary type τ (the notion of free and bound variables is well known; see, e.g., [1,2]). Type τ is restricted in ML and Haskell to be any simple (unquantified) type, characterizing an important restriction of the so-called ML-style or Let-polymorphism. *Constrained polymorphism* allows instantiation of quantified type variables to be restricted to *some*, instead of being possible to occur for all simple types. The set of types to which a type variable can be instantiated depends on the types of definitions of overloaded names (or symbols) that exist in the relevant context.

Constrained polymorphism is supported in programming languages like Haskell by *context-dependent overloading*, which is a form of overloading in which the decision of which definition of an overloaded name is used depends on the context where this name is used. In other words, in any

expression $f\ e$, the decision of which f is called or which e is applied depends not only on the types of f and e, but also on the context in which $f\ e$ is used.

In such systems, ambiguity becomes a concern. The existence of ambiguous expressions prevents a coherent semantics to be defined by induction on type system derivations, where coherence establishes a single well-defined meaning for each expression. That is, a *coherent* semantics is such that, for any well-typed expression e:

> "if Δ and Δ' are derivations of typing formulas $\Gamma \vdash e : \sigma$ and $\Gamma' \vdash e : \sigma$, respectively, and Γ and Γ' give the same type to every x free in e, then
>
> $$[\![\Gamma \vdash e : \sigma]\!]_\eta = [\![\Gamma' \vdash e : \sigma]\!]_\eta$$
>
> where the meanings are defined using Δ and Δ', respectively." [1]

An expression e of type σ for which there exist two distinct syntax-driven derivations (Δ and Δ', for which distinct semantic values might be assigned to e) is called ambiguous, in a typing context that gives the same type to every x free in e as Γ and Γ'. The restriction to syntax-driven derivations avoids differences that are neither related to the syntax of terms nor to the used typing information.

A type system for a Haskell-like programming language, that supports context-dependent overloading and follow the Hindley-Milner approach of providing context-free type instantiation, allows distinct derivations of the same type to be derivable for some expressions, which are then ambiguous. Such expressions are usually rejected by the type inference algorithm, which becomes then not complete with respect to the type system. This article addresses this issue by considering an alternative approach that disallows context-independent type instantiation for type systems that support context-dependent overloading.

Ambiguity in the presence of context-dependent overloading is discussed, in the traditional way of allowing context-independent type instantiation and by characterizing ambiguity as a syntactic property of constrained types, by Jones [3] and by Stuckey and Sulzmann [4]. The unfortunate lack of completeness of type inference algorithms and incoherence of semantics definitions are reported by Vytiniotis et al. [5]. Faxén [7] expresses wishes for a deterministic way to outlaw ambiguity in the type system, at the same time recognizing the need for the description to remain more abstract than that obtained directly from the type inference algorithm. In our view, the abstract view is provided by the fact that the type system is defined in terms of relations, not functions, and the type inference algorithm can be obtained by transforming these relations into functions.

presents an alternative approach to type inference in the presence of Haskell-style constrained type classes. The key feature of the type system is that it allows only context-dependent instantiation, so an expression, if typeable, has a unique type derivation. We can therefore define the semantics over these type-derivations, thereby ensuring coherence. The declarativeness of the specification of the type system is, in our view, equivalent to it being given by a relation (between typing contexts, expressions and polymorphic constrained types), not as a function. In order to transform it into a type inference algorithm, it suffices to transform all used relations (used in the definition of such a relation) into functions.

The fact that the type system allows only context-dependent type instantiation eliminates the problem of the lack of principal type caused by user-specified type signatures, reported by Faxén [6, Sect. 3].

The article also presents an alternative approach for dealing with ambiguity in the context of Haskell's open world approach. In Haskell, an expression is considered as ambiguous without conformance to the existence of two or more distinct derivations of the same type for this expression. Ambiguity is considered in Haskell as a syntactic condition on type expressions, conflicting with the standard semantically-related definition given above. This occurs because Haskell uses an open world approach to overloading, according to which new definitions of overloaded names might be inserted without altering the typability of expressions. This paper presents an alternative approach where there is no conflict with the standard semantic definition of ambiguity, built upon the distinction between ambiguity and resolved overloading. In this approach, the possibility of inserting new definitions (i.e., openness of the world) is restricted to cases when overloading is not resolved. When overloading is resolved, existing definitions of overloaded names are then considered, in order to check ambiguity.

With the purpose of clarifying these issues, namely that ambiguity is distinct from resolved overloading, and that ambiguity should be considered if and only if (or when and only when) overloading is resolved, we present and discuss examples in the next section which consider ambiguity in the presence of multi-parameter type classes, where ambiguity becomes more relevant. In Sect. 3 we define a mini-language called core Haskell that supports context-dependent overloading and multi-parameter type classes, and define a type system that avoids ambiguous expressions to be well-typed. In Sect. 4 we define a semantics by induction on core Haskell's type system derivations. Section 5 concludes.

2 Ambiguity and overloading resolution

Example 1 Consider expression $f\ o$, used in a context where f and o have the following types:

$f{::}Int \to Int$
$f{::}Int \to Float$
$f{::}Float \to Float$
$o{::}Int$
$o{::}Float$

In Haskell extended with multi-parameter type classes, referred to as Haskell+MPTC in the sequel, this can be achieved by considering that there exist declarations of type classes $F\ a\ b$ and $O\ a$ where f and o have been annotated in these classes with types $a \to b$ and a, respectively, and there exist also instance definitions $F\ Int\ Int$, $F\ Int\ Float$, $F\ Float\ Float$, $O\ Int$ and $O\ Float$.

In Haskell+MPTC, expression $f\ o$ is not ambiguous, because overloadings of f and o have not been resolved.

The main purpose of this example is to show that we can have, if context-independent type instantiation is allowed, two derivations of the same type ($Float$) with distinct semantics, for an expression that is not ambiguous: one derivation for f of type $Int \to Float$, and another for f of type $Float \to Float$.

This example also illustrates that this occurs despite the fact that $f\ o$ has type $(F\ a\ b, O\ a) \Rightarrow b$, where type variable a occurs in the constraints but not in the simple type (b), considering that f is a member of type class F with two parameters a, b, having (implicitly quantifiable) type $F\ a\ b \Rightarrow a \to b$, and o is a member of type class O with (implicitly quantifiable) type $O\ a \Rightarrow a$.

In Haskell, a syntactic condition on type expressions that characterizes overloading whose resolution cannot be further deferred (i.e., must have occurred), which we call *overloading resolution condition* characterizes also "type ambiguity". It is a syntactic condition, that conflicts with the standard definition of ambiguity, based on the existence of distinct type system derivations of the same type for an expression.

The overloading resolution condition used in Haskell (Haskell 98 or Haskell 2010), which supports only single parameter type classes, has been changed in Haskell implementations that support multi-parameter type classes. In the case of single parameter type classes, the overloading resolution condition for constrained type $P \Rightarrow \tau$ is simply $tv(P) \not\subseteq tv(\tau)$ (i.e., there is a type variable that occurs in P but not in τ). In the case of multi-parameter type classes, the condition considers so-called reachable type variables. A type variable occurring in P is reachable, from a set of type variables $tv(\tau)$ in a constrained type $P \Rightarrow \tau$, if it occurs in τ or if it occurs in a constraint in P where another reachable type variable occurs (if a type variable is not reachable, it is, of course, unreachable). In the example above, type variable a in $(F\ a\ b, O\ a) \Rightarrow b$ occurs in the set of constraints ($\{F\ a\ b, O\ a\}$) but not in the simple type (b). This does not characterize that overloading must have been resolved, because type variable a is reachable, since it occurs in constraint $F\ a\ b$, where another reachable type variable (b) occurs. This idea, used nowadays in Haskell implementations that support multi-parameter type classes, appeared firstly in [8], as far as we know.

If used in a program context that requires $f\ o$ to be of type Int—in an expression such as, for example, $f\ o$ + (1::Int)—overloadings of f and o in $f\ o$ are resolved, with f and o having types $Int, \to Int$ and Int, respectively.

If used in a program context that requires $f\ o$ to be of type $Float$, overloadings of f and o in $f\ o$ cannot be resolved, and then, in the context of this example, we have ambiguity. If used in a program context that requires $f\ o$ to be of a type τ distinct from Int and $Float$, overloadings of f and o in $f\ o$ cannot be resolved either, but we have then, in the context of this example, unsatisfiability, since we cannot have a derivation of such type τ for e in this context.

Example 2 Let e_0 be the expression *show.read* (where "." denotes function composition, so e_0 can be written also as $(\lambda x \text{ -> } show\ (read\ x))$. In Haskell, this expression is considered as ambiguous, irrespective of the context in which it occurs (see, e.g., [3,5]).

When used in a context with two or more instance definitions, of classes $Show$ and $Read$, that give functions $show$ and $read$ types, say, $show : Int \to String, show : Bool \to String, read : String \to Int$ and $read : String \to Bool$, expression e_0 is ambiguous: there exist two distinct derivations of type $String \to String$ for e_0 (one using *read* and *show* with types $read : String \to Int$and $show : Int \to String$, and the other using *read* and *show* with types $read : String \to Bool$and $show : Bool \to String$, respectively), and each one would give it distinct meanings.

However, if e_0 is used in a context with a single instance for both *show* and *read*, say $show : Int \to String$and $read : String \to Int$, then there is no ambiguity, since there exists only one derivation for e_0 of type $String \to String$. If there are no definitions of *show* or no definitions of *read* in the typing context, again there is an error, but of unsatisfiability, not ambiguity.

Example 3 Consider expression `[] == []`, where `(==)` is overloaded and `[]` denotes an empty list.

The expression is ambiguous (according to the standard definition), if and only if there exist two or more distinct type system derivations—each given `(==)` distinct types, say $[T_1] \to [T_1] \to Bool$ and $[T_2] \to [T_2] \to Bool$—that may thus assign distinct meanings to `([] == [])` : : *Bool* in the relevant context. We could have instances of `(==)` for types $[T_1]$ and $[T_2]$ for which `([] == [])` are assigned (unexpectedly) semantic value *False* if `[]` has type $[T_1]$, and *True* if `[]` has type $[T_2]$.

In Haskell this is not possible without overlapping instances or without hiding the Haskell prelude definition of `(==)` for lists.

Example 4 Let e_1 be the expression *fst* (*True, o*), where o is overloaded (or is an expression with a constrained type). This expression has type *Bool*. Overloading of o is not resolved, but it need not be for e_1 to be well-typed (and evaluated, as equal to *True*).

The ambiguity or not of e_1 in GHC [9] and its interactive interpreter counterpart GHCi, the most widely used implementations of Haskell, depends on which o is used and on whether the compiler, GHC, or an interactive session of the interpreter, GHCi, is used. In GHC and in non-interactive sessions of GHCi, e_1 is not well-typed. In an interactive session (i.e., if it is typed at the GHCi's prompt), if o has only constraints on some particular type classes (namely *Eq*, *Ord*, *Num* and *Show*), it has type *Bool*.

3 Core language

We use a context-free syntax of core Has-kell expressions, given in Fig. 1, where meta-variable x represents a variable. We use meta-variables x, y, z for variables, and e for expressions, possibly primed or subscripted. The language of terms is called here core Has-kell (not core ML) because expressions occur in a global context with information about overloaded symbols.

A context-free syntax of constrained types is presented in Fig. 2, where meta-variable usage is also indicated. 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. Also, type expression variables are called simply type variables.

$$\text{Expressions} \quad e ::= x \mid \lambda x.\, e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e$$

Fig. 1 Context-free syntax of core Haskell expressions

Class Name	C, D	
Type variable	α, β	
Type constructor	T	
Simple Constraint	π	$::= C\,\overline{\tau}$
Set of Simple Constraints	P, Q	
Constraint	ϕ	$::= P \Rightarrow \pi$
Closed Constraint	θ	$::= \forall\overline{\alpha}.\,\phi$
Overloading Environment		
Set of Closed Constraints	Θ	
Simple Type	τ	$::= \alpha \mid T \mid \tau\ \tau$
Constrained Type	ρ	$::= P \Rightarrow \tau$
Type	σ	$::= \forall\overline{\alpha}.\,\rho$

Fig. 2 Types, constraints and meta-variable usage

$\overline{x}$ denotes the sequence $x_1, \ldots, x_n$, where $n \geq 0$. When used in the context of a set, it denotes the corresponding set of elements in the sequence ($\{x_1, \ldots, x_n\}$).

We assume for simplicity that overloaded definitions are predefined, and form a global overloading environment (cf., e.g., [3,10,11]). The global overloading environment is always a fixed set of closed constraints, being an unchanged part of typing contexts. We write Θ_Γ to mean a fixed, global overloading environment that is assumed to be a part of typing context Γ.

A substitution, denoted by meta-variable S, possibly primed or subscripted, is a function from type variables to simple type expressions. The identity substitution is denoted by id. $S\sigma$ represents the capture-free operation of substituting $S(\alpha)$ for each free occurrence of type variable α in σ. $S\theta$ and sets of types and constraints are defined analogously. Symbol $\circ$ denotes function composition, and $dom(S) = \{\alpha \mid S(\alpha) \neq \alpha\}$.

$S[\overline{\alpha} \mapsto \overline{\tau}]$ denotes updating of S, that is, the substitution S' such that $S'(\beta) = \tau_i$ if $\beta = \alpha_i$, for $i = 1, \ldots, n$, otherwise $S(\beta)$. We use this function updating notation for other functions other than substitutions. Also, $[\overline{\alpha} \mapsto \overline{\tau}] = id[\overline{\alpha} \mapsto \overline{\tau}]$.

The restriction $S|_V$ of S to V denotes the substitution S' such that $S'(\alpha) = S(\alpha)$ if $\alpha \in V$, otherwise α.

A substitution S is said to be more general than a substitution S', written $S \leq S'$, if there is a substitution S_1 such that $S' = S_1 \circ S$.

We use:

$$\Gamma(x) = \{\sigma \mid (x : \sigma) \in \Gamma, \text{for some } \sigma\}$$
$$\Gamma, x : \sigma = (\Gamma \ominus x) \cup \{x : \sigma\}$$
$$\Gamma \ominus x = \Gamma - \{(x : \sigma) \in \Gamma\}$$

A type system for core Haskell is presented in Fig. 3, using rules of the form $\Gamma \vdash e : (\phi, S)$, which means that e has type

$$\frac{(x : \forall\overline{\alpha}.\,\phi) \in \Gamma \quad \overline{\beta}\ \text{fresh}}{\Gamma \vdash x : ([\alpha \mapsto \beta]\phi, id)}\ (\textsc{var})$$

$$\frac{\Gamma, x : \alpha \vdash e : (P \Rightarrow \tau, S) \quad \alpha\ \text{fresh} \quad \tau' = S\,\alpha}{\Gamma \vdash \lambda x.\, e : (P \Rightarrow \tau' \to \tau, S)}\ (\textsc{abs})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (P_1 \Rightarrow \tau_1, S_1) \quad S_1\Gamma \vdash e_2 : (P_2 \Rightarrow \tau_2, S_2) \\ mgu(S_2\tau_1 = \tau_2 \to \alpha, S'),\ \alpha\ \text{fresh} \\ S = S' \circ S_2 \circ S_1,\ \tau = S\alpha, V = tv(\tau) \\ P = SP_1 \oplus_V SP_2,\ P|^*_V \gg_\Theta Q \\ P_u = P - P|^*_V,\ P_u \gg_\Theta Q_u,\ Q_u = \emptyset\end{array}}{\Gamma \vdash e_1\ e_2 : (Q \Rightarrow \tau, S)}\ (\textsc{app})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\phi_1, S_1)\ \ gen(\sigma, \phi_1, tv(S_1\Gamma)) \\ S_1\Gamma, x{:}\sigma \vdash e_2 : (\phi, S)\end{array}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : (\phi, S)}\ (\textsc{let})$$

Fig. 3 Type system

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

$$\frac{\Theta_\Gamma = \Theta_{\Gamma'} \quad \Gamma(x) \leq \Gamma'(x) \text{ for all } x \in dom(\Gamma)}{\Gamma \leq \Gamma'}$$

Fig. 4 Partial order on types, constraints and typing contexts

ϕ in typing context Γ. We have that $S\Gamma \vdash e : (\phi, S')$ holds whenever $\Gamma \vdash e : (\phi, S)$ holds, where $S' \leq S$. Program contexts in which e occurs may instantiate e's type, as stated in Theorem (1) below.

Example 5 As an example of a program context requiring type instantiation, which occurs as a result of function application, consider expression x and typing context $\Gamma = \{f : Int \to Int, x : \alpha\}$; we can derive $\Gamma \vdash f\,x : (Int, S)$, where $S = [\alpha \mapsto Int]$. From $S\Gamma = \{f : Int \to Int, x : Int\}$; we can derive $S\,\Gamma \vdash e : (Int, id)$.

In general, we have the following, where a program context $C[e]$ is an expression which has e as a subexpression, and orderings on types and typing contexts are as defined in Fig. 4.

Theorem 1 *If $\Gamma \vdash e : (\phi, S)$ holds then $S\Gamma \vdash e : (\phi, S_0)$ holds, where $S_0 \leq S$.*

Furthermore, for all program contexts $C[e]$ in which e occurs and all typing contexts Γ' such that $\Gamma \leq \Gamma'$ and $\Gamma' \vdash C[e] : (\phi', S')$ is derivable, for some ϕ', S', we have that $S \leq S'$.

For each expression e, there is a unique type ϕ derivable for e in a typing context Γ. However, expression e can have though a set of instance-types, in program contexts that require instantiation of ϕ in Γ (in fact, in all typing contexts Γ' such that $\Gamma \leq \Gamma'$, cf. Theorem 1). Consider the following example where `B` and `C` represents abbreviations of *Bool* and *Char*, respectively.

Example 6 Let $((==) : \forall a.Eq : a \Rightarrow a \to a \to \texttt{B}) \in \Gamma$, $\{Eq\ \texttt{B}, Eq\ \texttt{C}\} \subseteq \Theta_\Gamma$ and $e = ((==)\,True, (==)\ \texttt{'*'})$. Then $\Gamma \vdash e : ((\texttt{B} \to \texttt{B}, \texttt{C} \to \texttt{B}), S)$ is derivable, where $S = [a \mapsto \texttt{B}, b \mapsto \texttt{C}]$, and a, b are fresh type variables. Instance-types of `(==)` in program contexts `(==)` *True* and `(==) '*'` are respectively $\texttt{B} \to \texttt{B} \to \texttt{B}$ and $\texttt{C} \to \texttt{C} \to \texttt{B}$.

Instance-types are formally defined as follows.

Definition 1 Given expression e and typing context Γ, we have that $S'\phi$ is an *instance-type for e in Γ* if $\Gamma \vdash e : (\phi, S)$ and $\Gamma' \vdash C[e] : (\phi', S')$ hold, where $\Gamma' \leq \Gamma$.

Furthermore, $S'\phi$ is a greatest (most specific) instance-type for an *occurrence* of e in Γ if $S'\phi$ is an instance-type for e in Γ and there is no instance-type $S_1\phi$ distinct from $S'\phi$ for e in Γ such that $S_1 \leq S$.

Distinct occurrences of an expression can have distinct greatest instance-types. For example, the instance-types given in Example 6 are greatest instance-types for the corresponding occurrence of `(==)`.

mgu is the most general unifier relation [12–14]: $mgu(\mathcal{T}, \mathcal{S})$ is defined to hold between a set of pairs of simple types or a set of constraints $\mathcal{T}$ and a substitution S if the following hold: i) $S\tau = S\tau'$ for every $(\tau, \tau') \in \mathcal{T}$ (analogously, $S\pi = S\pi'$ for every $(\pi, \pi') \in \mathcal{T}$), and if S' is a unifier of $\mathcal{T}$, then $S' \leq S$.

When the parameter of *mgu* is a singleton set, following common practice it is written simply as an equality; e.g., $mgu(\pi = \pi', S)$ is written instead of using a set notation like this: $mgu\left(\{(\pi, \pi')\}, S\right)$.

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

The set of constraints $P|^*_V$ denotes the subset of constraints of P with reachable type variables with respect to the set of type variables V [8]. A type variable $\alpha \in P$ is called *reachable* with respect to a set of type variables V if $\alpha \in V$ or $\alpha \in tv(\pi)$ and there exists $\beta \in tv(\pi)$ such that β is reachable (otherwise it is an unreachable type variable). Reachability is considered always with respect to $V = tv(\tau)$ for a constraint set P that occurs on a constrained type $P \Rightarrow \tau$. For example, type variables a, b are reachable and c is unreachable in constraint set $\{C\ a\ b, D\ c\}$ of constrained type $\{C\ a\ b, D\ c\} \Rightarrow a$. Reachability is defined in Fig. 5.

Given any set of type variables V, the constraints of a constraint set P can be partitioned into two disjoint subsets $P|^*_V$ and $P - P|^*_V$, the first containing constraints with at least one reachable type variable and the second constraints with only unreachable type variables.

$P \oplus_V Q$ denotes the constraint set obtained by adding from Q only constraints with type variables reachable from V, i.e., $P \oplus_V Q = P \cup Q|^*_V$ [8,15]. This takes into account that, in an application of a function with type, say $\tau_1 \to \tau$, to an expression with type $P \Rightarrow \tau_1$, it is not always adequate to include in the constraint set of the result all constraints from Q. This occurs because constraints in P may refer to disregarded, non-selected parts of the argument. Consider for example expression $fst(True, o)$ (cf., Sect. 2), where o has any type with non-empty constraints. The type of this expression should be *Bool*, that is, constraints on the type of o should not be part of the set of constraints on the type of $fst(True, o)$.

$$P|^*_V = \begin{cases} P|_V & \text{if } tv(P) \subseteq V \\ P|^*_{tv(P|_V)} & \text{otherwise} \end{cases}$$

$$P|_V = \{\pi \in P \mid tv(\pi) \cap V \neq \emptyset\}$$

Fig. 5 Constraints reachable from a set of type variables

Relation $\gg_\Theta$ is a simplification relation on constraints, defined as a composition of improvement and context reduction, defined respectively in Sects. 3.3 and 3.4. Firstly, the more basic relations of entailment and satisfiability are defined, in Sects. 3.1 and 3.2, respectively, which are used in the definitions of improvement and context reduction.

Q_u in rule (APP) represents the set of constraints with unreachable type variables (subscript u in Q_u is an abbreviation of *unreachable*). The side-condition of rule APP expresses that Q_u should be empty. An empty set of constraints Q_u is obtained after checking satisfiability on the set of constraints P_u, if P_u has unreachable variables, and after removing these constraints with unreachable variables, by context reduction, if there exists a single satisfying solution for such constraints (cf., Definition of $\gg_\Theta$ in Fig. 6).

The article proposes to treat ambiguity by following a standard definition of ambiguity, that consists in: test satisfiability (i.e., "close the world") if overloading is resolved (or should have been resolved), that is, if there exist unreachable variables in the constraints. Nowadays, satisfiability is tested in Haskell, in the presence of multi-parameter type classes, only upon the presence of functional dependencies (or a similar mechanism), that closes the world when there exist *or not* unreachable type variables. Our treatment of ambiguity thus restricts the cases where satisfiabilty is tested, in case, say, a mechanism such as that of functional dependencies is used, and allows to avoid the use of such mechanism (of functional dependencies, or a similar one). In the latter case, the satisfiability trigger condition becomes the existence of unreachable variables, which may then be instantiated if there exists a single satisfying substitution.

The type system uses relations (mgu, gen, $\gg_\Theta$). The facts that it is syntax-directed and type instantiation occurs only if required by a program context allow a sound and complete type inference algorithm to be obtained by transforming these relations into computable functions.

The fact that the type system does not allow context-free type instantiation and allow the derivation of a single type for an expression in a given typing context makes it look closer to a type inference algorithm. Context-dependent notions of instance-types and most specific instance-type for each occurrence of an expression, in a given typing context, are introduced and used in the paper, instead of the standard context-independent notion of principal type.

Typability of function application $f\ e$ in this type system considers $f\ e$ to be well-typed, where f has type τ_1 and e has type τ_2, if there exists types $\tau \to \tau'$ and τ that are respective subtypes of τ_1 and τ_2, where subtyping is simply a matching relation, as defined in Fig. 4.

Fig. 6 Constraint set simplification

$$\frac{P \vdash^{\Theta}_{\mathtt{impr}} P' \quad P' \vdash^{\Theta}_{\mathtt{red}} Q}{P \gg_\Theta Q}$$

$$\frac{}{\Theta \vdash \emptyset}\ (\mathtt{ENT}_0) \qquad \frac{(\forall \overline{\alpha}.\, P \Rightarrow \pi) \in \Theta}{\Theta \vdash \{[\overline{\alpha} \mapsto \overline{\tau}](P \Rightarrow \pi)\}}\ (\mathtt{INST})$$

$$\frac{\Theta \vdash P \quad \Theta \vdash \{P \Rightarrow \pi\}}{\Theta \vdash \{\pi\}}\ (\mathtt{MP}) \qquad \frac{\Theta \vdash P \quad \Theta \vdash Q}{\Theta \vdash P \cup Q}\ (\mathtt{CONJ})$$

Fig. 7 Constraint set entailment

3.1 Entailment

The property that a set of constraints P can be proven from (are entailed by) constraints in an overloading environment Θ, written as $\Theta \vdash P$, is defined in Fig. 7. Following [11,16], entailment is obtained from closed constraints only, contained in a fixed set of constraints Θ.

3.2 Satisfiability

Following [17], $\lfloor P \rfloor_\Theta$ is used to denote the set of satisfiable instances of constraint set P with respect to Θ:

$$\lfloor P \rfloor_\Theta = \{SP \mid \Theta \vdash SP\}$$

Example 7 As an example, consider:

$$\Theta = \{\forall a, b.Dab \Rightarrow C\ [a]\ b,\ D\,Bool\ [Bool]\}$$

Then, we have that $\lfloor C\ a\ a \rfloor_\Theta = \lfloor C\ [Bool]\ [Bool] \rfloor_\Theta$. Both constraints $D\ Bool\ [Bool] \Rightarrow C\ [Bool]\ [Bool]$ and $C\ [Bool]\ [Bool]$ are members of $\lfloor C\ a\ a \rfloor_\Theta$ (and of $\lfloor C\ [Bool]\ [Bool] \rfloor_\Theta$). A proof that $\Theta \vdash \{C\ [Bool] : [Bool]\}$ holds can be given from the entailment rules given in Fig. 7, since this is the conclusion of rule (MP) with premises $\Theta \vdash \{D\ Bool\ [Bool]\}$ and $\Theta \vdash \{D\ Bool\ [Bool] \Rightarrow C\ [Bool]\ [Bool]\}$, and these two premises can be derived by using rule (INST).

Equality of constraint sets is considered modulo type variable renaming. That is, constraint sets P, Q are also equal if there exists a renaming substitution S that can be applied to P to make $S\ P$ and Q equal. S is a renaming substitution if for all $\alpha \in dom(S)$ we have that $S(\alpha) = \beta$, for some type variable $\beta \notin dom(S)$.

If $SP \in \lfloor P \rfloor_\Theta$ then S is called a satisfying substitution for P.

Constraint set satisfiability is in general an undecidable problem [18]. It is restricted and redefined here by using a constraint-head-value finite function, in order to obtain decidability, as described below.

Constraint set satisfiability and simplification both use the same termination criterion, which is based on a measure of the sizes of types in type constraints, given the the constraint-head-value function. The sequence of constraints that unify with a constraint axiom in recursive calls of the function that checks satisfiability or simplification 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.

Constraint set satisfiability is defined so that we can obtain a sound and complete type inference algorithm, by just transforming the relations defined in the type system into functions.

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, defined as follows:

$$\begin{aligned} \nu(C\ \tau_1 \ldots \tau_n) &= \textstyle\sum_{i=1}^{n} \nu(\tau_i) \\ \nu(T) &= 1 \\ \nu(\alpha) &= 1 \\ \nu(\tau\ \tau') &= \nu(\tau) + \nu(\tau') \end{aligned}$$

Consider computation of satisfiability of a given constraint set P with respect to constraint axioms Θ. Consider that, for checking satisfiability of a constraint $\pi \in P$, a constraint π' unifies with the head of constraint $\forall\overline{\alpha}.P_0 \Rightarrow \pi_0 \in \Theta$, with unifying substitution S, and suppose also that satisfiability of π requires also that some constraint π_1 unifies with π_0, giving corresponding unifying substitution S_1. We require the following in order for satisfiability of π to hold:

1. $\nu(S\pi')$ is less than $\nu(S_1\pi_1)$ or, if $\nu(S\pi') = \nu(S_1\pi_1)$, then $S\pi' \neq \pi''$, for any π'' that has the same constraint value as π' and unification with π_0 is required for satisfiability of π to hold, or
2. $S\pi$ is such that there is a type argument position $0 \leq i \leq n$ such that the number of type variables and constructors, in this argument position, of constraints that unify with π_0 is always decreasing.

More precisely, constrain-head-value-function Φ associates a pair (I, Π) to each constraint $(\forall\overline{\alpha}.P_0 \Rightarrow \pi_0) \in \Theta$, 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 $\forall\overline{\alpha}.\, P_0 \Rightarrow \pi_0 \in \Theta$, where I_0 is a tuple of $n+1$ values equal to ∞, a large enough constraint value defined so that $\infty > \nu(\pi)$ for any constraint $\pi \in \Theta$.

Decidability is guaranteed by defining the operation of updating $\Phi(\pi_0) = (I, \Pi)$, denoted by $\Phi[\pi_0, \pi]$, as follows, where $I = (v_0, v_1, \ldots, v_n)$ and $\pi = C\ \tau_1\ \ldots\ \tau_n$:

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

where

$$\begin{aligned} \Phi'(\pi_0) &= ((v_0', v_1', \ldots, v_n'), \Pi \cup \{\pi\}) \\ \Phi'(x) &= \Phi(x) \text{ for } x \neq \pi_0 \end{aligned}$$

$$\frac{}{P \vdash_{\mathtt{sats}}^{\Theta, Fail} \emptyset}\ (\mathtt{SatFail}_1) \qquad \frac{}{\emptyset \vdash_{\mathtt{sats}}^{\Theta,\Phi} \{id\}}\ (\mathtt{SatEmpty}_1)$$

$$\frac{\{\pi\} \vdash_{\mathtt{sats}}^{\Theta,\Phi} \mathbb{S}_0 \quad \mathbb{S} = \{S'S \mid S \in \mathbb{S}_0,\ S' \in \mathbb{S}_1,\ \ SP \vdash_{\mathtt{sats}}^{\Theta,\Phi} \mathbb{S}_1\}}{\{\pi\} \cup P \vdash_{\mathtt{sats}}^{\Theta,\Phi} \mathbb{S}}\ (\mathtt{SatConj}_1)$$

$$\frac{sats_1(\pi, \Theta, \Delta) \quad \mathbb{S} = \left\{ S'S \;\middle|\; \begin{array}{c} (S, Q, \pi') \in \Delta,\ S' \in \mathbb{S}_0, \\ Q \vdash_{\mathtt{sats}}^{\Theta, \Phi[\pi', S\pi]} \mathbb{S}_0 \end{array} \right\}}{\{\pi\} \vdash_{\mathtt{sats}}^{\Theta,\Phi} \mathbb{S}}\ (\mathtt{SatInst}_1)$$

Fig. 8 Decidable constraint set satisfiability

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

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

Let $sats_1(\pi, \Theta, \Delta)$ hold if

$$\Delta = \left\{ (S|_{tv(\pi)}, SP_0, \pi_0) \;\middle|\; \begin{array}{l} (\forall\overline{\alpha}.\ P_0 \Rightarrow \pi_0) \in \Theta, \\ mgu(\pi = \pi_0, S) \text{ holds} \end{array} \right\}$$

The set of satisfying substitutions for constraint set P with respect to the set of constraint axioms Θ is given by $\mathbb{S}$, such that $\mathrm{P} \vdash_{\mathtt{sats}}^{\Theta,\Phi_0} \mathbb{S}$ holds, as defined in Fig. 8.

The following examples illustrate the definition of constraint set satisfiability as defined in Fig. 8. Let $\Phi(\pi).I$ and $\Phi(\pi).\Pi$ denote the first and second components of $\Phi(\pi)$, respectively.

Example 8 Consider satisfiability of $\pi = Eq\,[\,[\,\mathtt{I}\,]\,]$ in $\Theta = \{Eq\ \mathtt{I},\ \forall a.\, Eq\ a \Rightarrow Eq\,[a]\}$, letting $\pi_0 = Eq\,[a]$; we have:

$$\frac{sats_1\,(\pi, \Theta, \{(S|_{\emptyset}, Eq\,[\,\mathtt{I}\,], \pi_0)\}),\ \ S = [a_1 \mapsto [\,\mathtt{I}\,]] \quad \mathbb{S}_0 = \{S_1 \circ id \mid\ S_1 \in \mathbb{S}_1,\ \ Eq\,[\,\mathtt{I}\,] \vdash_{\mathtt{sats}}^{\Theta,\Phi_1} \mathbb{S}_1\}}{\pi \vdash_{\mathtt{sats}}^{\Theta,\Phi_0} \mathbb{S}_0}$$

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:

$$\frac{sats_1(Eq\,[\,\mathtt{I}\,], \Theta, \{(S'|_{\emptyset}, \{Eq\ \mathtt{I}\}, \pi_0)\}),\ \ S' = [a_2 \mapsto \mathtt{I}] \quad \mathbb{S}_1 = \{S_2 \circ id \mid\ S_2 \in \mathbb{S}_2,\ \ Eq\ \mathtt{I} \vdash_{\mathtt{sats}}^{\Theta,\Phi_2} \mathbb{S}_2\}}{Eq\,[\,\mathtt{I}\,] \vdash_{\mathtt{sats}}^{\Theta,\Phi_1} \mathbb{S}_1}$$

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

$$\frac{\begin{array}{l} sats_1\big(Eq\ \mathtt{I}, \Theta, \{(id, \emptyset, Eq\ \mathtt{I})\}\big) \\ \mathbb{S}_2 = \{S_3 \circ id \mid\ S_3 \in \mathbb{S}_3,\quad \emptyset \vdash^{\Theta,\Phi_3}_{\mathtt{sats}} \mathbb{S}_3\} \end{array}}{Eq\ \mathtt{I} \vdash^{\Theta,\Phi_2}_{\mathtt{sats}} \mathbb{S}_2}$$

where $\Phi_3 = \Phi_2[Eq\ \mathtt{I}, Eq\ \mathtt{I}]$ and $\mathbb{S}_3 = \{id\}$ by $(\mathtt{SEmpty_1})$.

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 9 Consider the satisfiability of $\pi = C\ \mathtt{I}\ (T^3\ \mathtt{I})$ in $\Theta = \{C\ (T\ a)\ \mathtt{I}, \forall a, b.\, C\ (T^2\ a)\ b \Rightarrow C\ a\ (T\ b)\}$. We have, where $\pi_0 = C\ a\ (T\ b)$:

$$\frac{\begin{array}{l} sats_1\big(\pi, \Theta, \{(S\mid_\emptyset, \{\pi_1\}, \pi_0)\}\big) \\ S = [a_1 \mapsto \mathtt{I}, b_1 \mapsto T^2\ \mathtt{I}] \\ \pi_1 = C\ (T^2\ \mathtt{I})\ (T^2\ \mathtt{I}) \\ \mathbb{S}_0 = \{S_1 \circ id \mid\ S_1 \in \mathbb{S}_1,\quad \pi_1 \vdash^{\Theta,\Phi_1}_{\mathtt{sats}} \mathbb{S}_1\} \end{array}}{\pi \vdash^{\Theta,\Phi_0}_{\mathtt{sats}} \mathbb{S}_0}$$

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

$$\frac{\begin{array}{l} sats_1\big(\pi_1, \Theta, \{(S'\mid_\emptyset, \{\pi_2\}, \pi_0)\}\big) \\ S' = [a_2 \mapsto T^2\ \mathtt{I}, b_2 \mapsto T\ \mathtt{I}] \\ \pi_2 = C\ (T^4\ \mathtt{I})\ (T\ \mathtt{I}) \\ \mathbb{S}_1 = \{S_2 \circ [a_1 \mapsto T^2\ a_2] \mid S_2 \in \mathbb{S}_2,\quad \pi_2 \vdash^{\Theta,\Phi_2}_{\mathtt{sats}} \mathbb{S}_2\} \end{array}}{\pi_1 \vdash^{\Theta,\Phi_1}_{\mathtt{sats}} \mathbb{S}_1}$$

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 $S' = [a_3 \mapsto T^4\ \mathtt{I}, b_2 \mapsto \mathtt{I}]$, 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 $\pi_3 = C\ (T^6\ \mathtt{I})\mathtt{I}$, that unifies with $C\ (T\ a)\ \mathtt{I}$, returning $\mathbb{S}_3 = \{[a_3 \mapsto T^5\ \mathtt{I}]|_\emptyset\} = \{id\}$. 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 10 Consider the satisfiability of $\pi = C\ (T^2\ \mathtt{I})\ \mathtt{F}$ in $\Theta = \{C\ \mathtt{I}\ (T^2\ \mathtt{F}), \forall a, b.\, C\ a\ (T\ b) \Rightarrow C\ (T\ a)\ b\}$ and let $\pi_0 = C\ (T\ a)\ b$. Then:

$$\frac{\begin{array}{l} sats_1(\pi, \Theta, \{\big(S\mid_\emptyset, \{\pi_1\}, \pi_0\big)\}) \\ S = [a_1 \mapsto (T\ \mathtt{I}), b_1 \mapsto \mathtt{F}] \\ \pi_1 = C\ (T\ \mathtt{I})\ (T\ \mathtt{F}) \\ \mathbb{S}_0 = \{S_1 \circ id \mid\ S_1 \in \mathbb{S}_1,\quad \pi_1 \vdash^{\Theta,\Phi_1}_{\mathtt{sats}} \mathbb{S}_1\} \end{array}}{\pi_1 \vdash^{\Theta,\Phi_0}_{\mathtt{sats}} \mathbb{S}_0}$$

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

$$\frac{\begin{array}{l} sats_1(\pi_1, \Theta, \{\big(S'\mid_\emptyset, \{\pi_2\}, \pi_0\big)\}) \\ S' = [a_2 \mapsto \mathtt{I}, b_2 \mapsto T\ \mathtt{F}], \quad \pi_2 = C\ \mathtt{I}\ (T^2\ \mathtt{F}) \\ \mathbb{S}_1 = \{S_2 \circ id \mid\ S_2 \in \mathbb{S}_2,\quad \pi_2 \vdash^{\Theta,\Phi_2}_{\mathtt{sats}} \mathbb{S}_2\} \end{array}}{\pi_1 \vdash^{\Theta,\Phi_1}_{\mathtt{sats}} \mathbb{S}_1}$$

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 $sats_1(C\ \mathtt{I}\ (T^2\ \mathtt{F}), \Theta) = \{(id, \emptyset, C\ \mathtt{I}\ (T^2\ \mathtt{F})\}$).

Since satisfiability of type class constraints is in general undecidable [18], there exist satisfiable instances which are considered to be unsatisfiable according to the definition of Fig. 8. Examples can be constructed by encoding instances of solvable post correspondence problems by means of constraint set satisfiability, using Smith's scheme [18].

To prove that satisfiability as defined in Fig. 8 is decidable, consider that there exist finitely many constraints in Θ, 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 $\Phi[\pi_0, \pi] = Fail$ and satisfiability yields $\emptyset$ as the set of satisfying solutions 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.

3.3 Improvement

Improvement is a satisfiability preserving relation: improvement of constraint set P is the process of finding a least general substitution S such that $S\ P$ preserves the set of satisfiable instances of P [3].

In this paper, improvement is used to remove unreachable type variables for resolving overloading, when overloading resolution cannot be further deferred, and for detecting ambiguity or unsatisfiability, if unreachable type variables cannot be removed (that is, overloading resolution is not possible). For any constrained type $P \Rightarrow \tau$, improvement is tested only upon the presence of unreachable type variables, that is, if $P_u = P - P|^*_{tv(\tau)} \neq \emptyset$.

This is a consequence of the side-condition $(Q_u = \emptyset)$ in rule (APP), Fig. 3.

If the set $\mathbb{S}$ of satisfiable instances of P_u has more than one element, we have ambiguity; if $\mathbb{S}$ is empty, we have unsatisfiability; otherwise, if $\mathbb{S}$ is a singleton $\{S\}$, then P is improved to $S\ P$, which can is then reduced to a set of constraints without unreachable type variables (that is, the set of constraints in $S\ P_u$ can be removed, since overloading is resolved).

Improvement is defined in Fig. 9, where Φ_0 is as defined in Sect. 3.2, page 16.

$$\frac{P \vdash^{\Theta,\Phi_0}_{\mathtt{sats}} \mathbb{S} \quad Q = S\ P \text{ if } \mathbb{S} = \{S\},\ \text{otherwise } P}{P \vdash^{\Theta}_{\mathtt{impr}} Q}$$

Fig. 9 Constraint set improvement

3.4 Context reduction

Informally speaking, context reduction is a process that reduces a constraint π into Q if there is a *matching instance* for π in Θ, that is, there exists $(\forall\overline{\alpha}.\, P \Rightarrow \pi') \in \Theta$ such that $S\pi' = \pi$, for some S, and $S\,P$ reduces to Q. If there is no matching instance for π or no reduction of $S\,P$ is possible, then π reduces to itself. Note that constraint sets can be reduced into larger constraint sets.

As an example of a context reduction, consider an instance declaration that introduces $\forall a.\, Eq\,a \Rightarrow Eq\ [a]$ in Θ; then $Eq\ [a]$ 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., [3,7,10].

Context reduction uses *matches*, defined as follows:

$$matches(\pi, (\Theta, \Phi'), \Delta) \text{ holds if}$$
$$\Delta = \left\{ (SP_0, \pi_0, \Phi') \;\middle|\; \begin{array}{l} (\forall\overline{\alpha}.\, P_0 \Rightarrow \pi_0) \in \Theta, \\ mgm(\pi_0 = \pi, S),\ \Phi' = \Phi[\pi_0, \pi] \end{array} \right\}$$

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 [19] are not considered.

Context reduction, defined in Fig. 10, uses rules of the form $P \vdash^{\Theta,\Phi}_{\mathrm{red}} Q; \Phi'$, meaning that either P reduces to Q under the set of closed constraints Θ and least constraint value function Φ, causing Φ to be updated to Φ', or $P \vdash^{\Theta,Fail}_{\mathrm{red}} P; Fail$.

Failure implies that a constraint set is updated 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$_0$). Rule (CONJ) specifies that constraint set simplification works, unlike constraint set satisfiability, by performing a union of the result of simplifying each constraint in the constraint set, separately.

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 $P = \{C\ a, D\ a\}$ in $\Theta = \{C\ Int, C\ Bool, D\ Int, D\ Char\}$. The results of computing satisfiability of P yields a single substitution where a maps to Int, not the union of computing satisfiability for $C\ a$ and $D\ a$ separately.

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

Rules (STOP$_0$) and (STOP) deal with failure due to updating of the constraint-head-value function.

$$\frac{}{\emptyset \vdash^{\Theta,\Phi}_{\mathrm{red}} \emptyset; \Phi}\ (\mathrm{RED}_0) \qquad \frac{\{\pi\} \vdash^{\Theta,\Phi}_{\mathrm{red}} P; \Phi_1 \quad Q \vdash^{\Theta,\Phi_1}_{\mathrm{red}} Q'; \Phi'}{\{\pi\} \cup Q \vdash^{\Theta,\Phi}_{\mathrm{red}} P \cup Q'; \Phi'}\ (\mathrm{CONJ})$$

$$\frac{matches(\pi, (\Theta, \Phi), \{(P, \pi', \Phi')\}) \quad P \vdash^{\Theta,\Phi'}_{\mathrm{red}} Q; \Phi''}{\{\pi\} \vdash^{\Theta,\Phi}_{\mathrm{red}} Q; \Phi''}\ (\mathrm{INST})$$

$$\frac{matches(\pi, (\Theta, \Phi), \{(P, \pi', \Phi')\}) \quad P \vdash^{\Theta,\Phi'}_{\mathrm{red}} Q; Fail}{\{\pi\} \vdash^{\Theta,\Phi}_{\mathrm{red}} \{\pi\}; Fail}\ (\mathrm{STOP}_0)$$

$$\frac{matches(\pi, (\Theta, \Phi), \{(P, \pi', Fail)\})}{\{\pi\} \cup P \vdash^{\Theta,\Phi}_{\mathrm{red}} \{\pi\} \cup P; Fail}\ (\mathrm{STOP})$$

Fig. 10 Context reduction

4 Semantics

A type class declaration defines overloaded names, also called class *members*, with corresponding types, and an instance declaration gives a value for each class member, referred to as a *member value* (sometimes also referred to in the literature as a "member function").

The semantics of core Haskell, given in Fig. 11, is based on the application of (so-called) *dictionaries* to overloaded names a standard core Haskell semantics [3,10,20]. A *dictionary* is a tuple that corresponds to an instance declaration, and contains values that correspond to the definitions given in the instance declaration for each class member. A dictionary of a superclass contains also a pointer to a dictionary of each of its subclasses, but the treatment of superclasses is standard and is omitted in this paper (see, e.g., [3,7,10]).

Figure 11 defines the semantics of core Haskell by induction on type system rules, with greatest instance-types of variables explicitly annotated, that is, typing formulas for variables have the form $\Gamma \vdash x :: \phi$ where ϕ is the greatest instance-type of this occurrence of x in typing context Γ (cf. Definition 1). The translation of the types of expressions are also defined in Fig. 11.

For each class declaration $\texttt{class}\ P \Rightarrow \overline{\alpha}\ \texttt{where}\ \overline{x} :: \overline{\tau}$, a sequence of selection functions is generated, one for each overloaded name in $\overline{x}$. The selection function corresponding to x_i simply selects the i-th component of the tuple parameter $(\ldots, x_i, \ldots)$ (if $n = 1$, selection is done by the identity function).

For example, class *Eq* generates a pair of selection functions `(==)` and `(/=)`, defined as equal to *fst* and *snd*. Module scope visibility rules of these generated names are not considered in this paper. See also Example 11 below.

Let $\overline{P}$ denote a sequence of constraints in P in a standard, say lexicographical order.

Each instance declaration $\texttt{instance}\ P \Rightarrow \pi\ \texttt{where}\ \overline{x} = \overline{e}$ of a class C generates a dictionary d_π. Each component

$$\frac{(x:\sigma)\in\Gamma}{[\![\Gamma\vdash x::P\Rightarrow\tau]\!]_\eta=\eta(x,P\Rightarrow\tau,\Gamma)\ ::\ \tau}\ (\textsc{var})$$

$$\frac{[\![\Gamma,x:\alpha\vdash e:(P\Rightarrow\tau,S)]\!]_\eta=\mathbf{e}\ ::\ \tau\quad\alpha\text{ fresh}\quad\tau'=S\alpha}{[\![\Gamma\vdash\lambda x.\,e:(P\Rightarrow\tau'\to\tau,S)]\!]_\eta=\lambda x.\,\mathbf{e}\ ::\ \tau'\to\tau}\ (\textsc{abs})$$

$$\frac{\begin{array}{c}[\![\Gamma\vdash e_1:(P_1\Rightarrow\tau_1,S_1)]\!]_\eta=\mathbf{e}_1\ ::\ S\tau_1\\ [\![S_1\Gamma\vdash e_2:(P_2\Rightarrow\tau_2,S_2)]\!]_\eta=\mathbf{e}_2\ ::\ S\tau_2\\ mgu_I(S_2\tau_1=\tau_2\to\alpha,S')\quad\alpha\text{ fresh}\\ S=S'\circ S_2\circ S_1,\quad\tau=S\alpha,\quad V=tv(\tau)\\ P=SP_1\oplus_V SP_2,\quad P|^*_V\gg_\Theta Q\\ P_u=P-P|^*_V\quad P_u\gg_\Theta Q_u\quad Q_u=\emptyset\end{array}}{[\![\Gamma\vdash e_1\ e_2:(Q\Rightarrow\tau,S)]\!]_\eta=\mathbf{e}_1\ \mathbf{e}_2\ ::\ \tau}\ (\textsc{app})$$

$$\frac{\begin{array}{c}\rho=P\Rightarrow\tau\\ [\![\Gamma\vdash e_1:(P_1\Rightarrow\tau_1,S_1)]\!]_\eta=\mathbf{e}_1{::}\tau_1\\ [\![S_1\Gamma,\ x{:}\sigma\vdash e_2:(P\Rightarrow\tau,S)]\!]_{\eta'}=\mathbf{e}_2{::}\tau\\ gen(\sigma,P_1\Rightarrow\tau_1,tv(S_1\Gamma)),\qquad\overline{v}=vSeq(P_1)\\ gen(\sigma',P\Rightarrow\tau,tv(S(S_1\Gamma))),\ \eta'=\eta\dagger(P_1\mapsto\overline{v})\end{array}}{[\![\Gamma\vdash\mathtt{let}\ x=e_1\ \mathtt{in}\ e_2:\rho]\!]_\eta=\mathbf{let}\ x=\mathbf{e}_1\ \mathbf{in}\ \lambda\overline{v}.\,\mathbf{e}_2\ ::\ \tau}\ (\textsc{let})$$

Fig. 11 Core Haskell semantics

in d_π is a function that takes one dictionary for each constraint in the (possibly empty) sequence $\overline{P}$ and yields the translation of e_i, the value bound by x_i in the instance declaration. The instance declaration makes values $\eta(S\pi)$ and $\eta(x_i, S\tau_i)$ to be equal to d_π, for all substitutions S, where τ_i is the simple type in the type of x_i.

Let $\eta\dagger(P\mapsto\overline{v})$ be equal to $\eta[\pi_1\mapsto v_1,\ldots,\pi_n\mapsto v_n]$, where $P=\{\pi_1,\ldots,\pi_n\}$.

$vSeq(\overline{P})$ denotes a sequence of fresh variables v_i, one for each π_i in the sequence $\overline{P}$.

mgu_I is a functional counterpart of the most general unifier relation (mgu).

We have that $\eta(x, P\Rightarrow\tau, \Gamma)$ gives the semantics of possibly overloaded name x, with instance-type $P\Rightarrow\tau$ and quantified type σ; η is overloaded to be used also on unqualified constraints (as in $\eta(\pi)$) and to yield dictionaries (as in $\eta(x,\tau)$). In the translation, x represents a selection function, $\eta(x,\tau)$ a dictionary, and $\overline{w}$ a sequence of arguments of the selected function, where arguments are themselves dictionaries:

$$\eta(x,P\Rightarrow\tau,\Gamma)=\begin{cases}x & \text{if } P_0=\emptyset\\ x\ \eta(x,\tau)\ \overline{w} & \text{otherwise}\end{cases}$$

where: $\forall\overline{\alpha}.\,P_0\Rightarrow\tau_0=\Gamma(x)$,

$S=mgu_I(\tau,\tau_0)$,

$\pi_1\ldots\pi_n=\overline{P_0}$,

for $i=1,\ldots,n: v_i=\eta(\pi_i)$,

$$w_i=\begin{cases}v_i & \text{if } \pi_i\in P\\ \eta(S\pi_i) & \text{otherwise}\end{cases}$$

Theorem 2 *For any derivations Δ, Δ' of typing formulas $\Gamma\vdash e:\phi$ and $\Gamma'\vdash e:\phi$, respectively, where Γ and Γ' give the same type to every x free in e, we have*

$$[\![\Gamma\vdash e:\phi]\!]_\eta=[\![\Gamma'\vdash e:\phi]\!]_\eta$$

where the meanings are defined using Δ and Δ', respectively.

Proof Since Γ and Γ' give the same type to every x free in e and the type system rules are syntax-directed, Δ and Δ' are the same. □

Consider the following Haskell program extract:

Example 11

```
class TEq a where
  teq:: a → a → (Bool,String)
instance TEq Int where
  teq i i' = (i==i',show i ++ " " ++ show i')
instance (TEq a, Show a) ⇒ TEq [a] where
  teq []     []     = (True,"")
  teq (a:x) (b:y) = let (ab,sab) = teq a b
                        (xy,sxy) = teq x y
                    in (ab && xy, sab ++ sxy)
  teq _      _     = (False,"")

teqww x=(teq [[x]],teq([1,2,3]::[Int])) --(1)
```

The translation of the first occurrence of *teq* in line `(1)` above is equal to $teq\ d_{TEqL}\ v_1\ v_2$, where *teq*'s translation is the identity function, $\mathbf{teq}_L$ is a function that receives the two dictionary arguments v_1 and v_2 passed to $teqww$ and yields the translation of function *teq* for lists defined above. The translation is given with respect to environment $\eta_0\dagger(P\mapsto\overline{v})$, where $P=\{Showa, TEqa\}$, $\overline{v}$ is the sequence $v_1\ v_2$, and η_0 is such that $\eta_0(teq,\tau)=d_{TEqL}$, where $\tau=$ `[[a]]` $\to$ `[[a]]` $\to(Bool, String)$, and d_{TEqL} is a dictionary with just one component $\mathbf{teq}_L$.

We have also that $\eta_0(TEqInt)$ is equal to a dictionary with just one member (say, d_{TEqInt}), and similarly for $\eta_0(ShowInt)$. The translation of the second occurrence of *teq* in line `(1)` above is equal to:

$$teq\ d_{TEqL}\ d_{TEqInt}\ d_{ShowInt}$$

Such use of dictionaries and the ensuing selection of member values at run-time can be avoided by passing values that correspond to overloaded names that are in fact used. A common case is that of a list equality function, that can receive an equality function for list elements, instead of a dictionary containing also an unused inequality function. Passing a dictionary to select at run-time the used equality function is unnecessary and inefficient. Full laziness and common subexpression elimination are techniques used to avoid repeated construction of dictionaries at run-time [3,7]. This and related implementation issues are however outside of the scope of this paper and are left for further work (see also [21,22]).

Note that constraints on types of expressions are considered in the semantics only in the cases of polymorphic and constrained overloaded variables. Consider for example expression *eqStar* given by:

$$\texttt{let}\ eq = \lambda x.\lambda y.\ (\texttt{==})\ x\ y\ \texttt{in}\ eq\ \texttt{'*'}$$

in a context where (==) has type $Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$ (we have not written a simpler expression because we want to contrast the semantics of (==) with those of expressions (==) x and (==) $x\ y$); the translation of *eqStar* is given by:

$$\textbf{let}\ eq = \lambda v.\lambda x.\lambda y.\ (\texttt{==})\ v\ x\ y\ \textbf{in}\ \ eqdictEqChar\ \texttt{'*'}$$

We have that (==) *dictEqChar* (as well as *eq dictEqChar*) returns a primitive equality function for characters, say *primEqChar*. Expression (==) is itself a function that takes a dictionary of type t and returns the equality function from that dictionary, of type $t \rightarrow t \rightarrow Bool$. The translation of each occurrence of (==) passes a pertinent dictionary value to (==) so that the type obtained is the expected type for an equality function on values of type t. Both expressions (==) x and (==) $x\ y$ have also constrained types, but a dictionary is passed only in the case of (==). The semantics of an expression with a constrained type where the set of constraints is non-empty only considers this set of constraints if the expression is an overloaded variable; otherwise constraints are disregarded in the semantics. Furthermore, since each occurrence of an overloaded variable has a translation that is the application of pertinent dictionary values to that variable, translation of *types* with constraints are never input or output values of the translation function (see Fig. 11).

Type soundness follows directly from the fact that if $[\![\Gamma \vdash e : P \Rightarrow \tau]\!]\eta = \mathbf{e} : \tau$ holds or, if e is a variable, if $[\![\Gamma \vdash x :: P \Rightarrow \tau]\!]\eta = \mathbf{e} : \tau$ holds, then $\Gamma' \vdash \mathbf{e} : \tau$ is derivable, where Γ' is appropriately defined so as to remove overloading-related data from Γ. This can be done by creating dictionaries and selection functions as described above, and inserting corresponding type assumptions
in Γ'.

Type soundness is obtained as a result of disallowing all ambiguous expressions and all expressions involving unsatisfiability in the use of overloaded names. For example, letting $e_0 \equiv (\lambda x -> show(readx))$, we would not have a derivation of $\Gamma' \vdash e_0 : String \rightarrow String$ corresponding to $\Gamma \vdash e_0 : String \rightarrow String$ if $\Gamma' \vdash d_{Readt} : String \rightarrow t$ is not derivable, which would happen if t is a fresh type variable or t can be more than one simple type. In other words, $\Gamma' \vdash d_{Readt} : String \rightarrow t$ is derivable if and only if t is a unique simple type.

5 Conclusion

This paper discusses ambiguity in the context of languages that support context-dependent overloading, such as Haskell.

A type system is presented that does not follow the Hindley-Milner approach of providing context-free type instantiation, as usually done in type systems for such languages. As a consequence, ambiguous expressions can be considered to be not well-typed, in conformance with type inference algorithms.

The type system does not allow context-free type instantiation and allows only a single type to be derived for an expression, in a given typing context, making it look closer and easier to be converted into a type inference algorithm. There is no notion of principal type (and thus no notion of "principal translation" of a term), in a given typing context. Related notions of instance type and most specific instance type for each occurrence of an expression, dependent on program contexts, are instead defined and used in the paper.

A semantics is defined by induction on the type system rules, for which coherence is trivial. Type soundness is obtained as a result of disallowing all ambiguous expressions and all expressions involving unsatisfiability in the use of overloaded names.

A standard definition of ambiguity is followed in the support for context-dependent overloading, where satisfiability is tested —i.e., "the world is closed"—if only if overloading is resolved (or should have been resolved), that is, if and only if there exist unreachable variables in the constraints on types of expressions. Nowadays satisfiability is tested in Haskell, in the presence of multi-parameter type classes, only upon the presence of functional dependencies or an alternative mechanism that specifies conditions for closing the world, and that may happen when there exist or not unreachable type variables in constraints. The satisfiability trigger condition is then given automatically, by the existence of unreachable variables in constraints, and does not need to be specified by programmers, using an extra mechanism.

References

1. Mitchell J (1996) Foundations of Programming Languages. MIT Press, Cambridge
2. Pierce B (2002) Types and Programming Languages. MIT Press, Cambridge
3. Jones M (1994) Qualified Types: Theory and Practice. Ph.D. thesis, Distinguished Dissertations in Computer Science. Cambridge Univ. Press, Cambridge
4. Stuckey P, Sulzmann M (2005) A Theory of Overloading. ACM Trans Prog Lang Syst (TOPLAS) 27(6):1216–1269
5. Vytiniotis D, Jones S, Schrijvers T, Sulzmann M (2011) OutsideIn(X): modular type inference with local assumptions. J Funct Program 21(4–5):333–412

6. Faxén K (2003) Haskell and Principal Types. In: Proceedings 2003 ACM SIGPLAN Haskell, Workshop, pp 88–97
7. Faxén K (2002) A static semantics for Haskell. J Funct Program 12:295–357
8. Camarão C, Figueiredo L (1999) Type Inference for Overloading without Restrictions, Declarations or Annotations. In: Proceeding 4th Fuji International Symp. on Functional and Logic Programming (FLOPS'99), LNCS 1722, Springer-Verlag, New York, pp 37–52
9. Jones SP, et al (2012) GHC—the Glasgow Haskell compiler. http://www.haskell.org/ghc/
10. Hall C, Hammond K, Jones S, Wadler P (1996) Type classes in Haskell. ACM Trans Program Lang Syst 18(2):109–138
11. Chakravarty MG, Keller SP, Jones S (2005) Marlow: associated types with class. In: Proceeding ACM Symp. on Principles of Programming Languages (POPL'05), ACM Press, Florida, pp 1–13
12. Robinson J (1965) A machine-oriented logic based on the resolution principle. J ACM 12:32–41
13. Eder E (1985) Properties of substitutions and unification. J Sym Comput 1:31–46
14. Palamidessi C (1990) Algebraic properties of idempotent substitutions. Lect Notes Comput Sci 443:386–399
15. Camarão C, Ribeiro R, Figueiredo L, Vasconcellos C (2009) A Solution to Haskell's Multi-Parameter Type Class Dilemma. In: Proceeding 13th Brazilian Symp. on Programming Languages (SBLP'2009), pp 5–18. http://www.dcc.ufmg.br/camarao/CT/solution-to-mptc-dilemma.pdf
16. Chakravarty M, Keller G, Jones S (2005) Associated type synonyms. In: Proceeding 10th ACM SIGPLAN International Conference on Functional Programming, pp 241–253
17. Jones M (1995) Simplifying and improving qualified types. In: Proceeding ACM Conf. on Functional Programming and Computer Architecture (FPCA'95), Cambridge University Press, Cambridge, pp 160–169
18. Smith G (1991) Polymorphic type inference for languages with overloading and subtyping. Ph.D. thesis, Cornell University, NY
19. Jones SP, others (2011) GHC—The Glasgow Haskell Compiler 7.0.4 User's Manual. http://www.haskell.org/ghc/
20. Wadler P, Blott S (1989) How to make ad-hoc polymorphism less ad hoc. In: Proceeding 16th ACM Symp. on Principles of Programming Language (POPL'89), ACM Press, Florida, pp 60–76
21. Peterson J, Jones M (1993) Implementing type classes. In: Proceeding ACM Conference on Programming Language Design and Implementation, SIGPLAN Notices 28(6), Florida, pp 227–236
22. Jones M (1994) Dictionary-free Overloading by Partial Evaluation. In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation