Ambiguity and Constrained Polymorphism

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

Rodrigo Ribeiro

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

Abstract

This paper considers the problem of ambiguity in Haskell-like languages. Overloading resolution is characterized in the context of constrained polymorphism by the presence of unreachable variables in constraints on the type of the expression. A new definition of ambiguity is presented, where existence of more than one instance for the constraints on an expression type is considered only after overloading resolution. This introduces a clear distinction between ambiguity and overloading resolution, makes ambiguity more intuitive and independent from extra concepts, such as functional dependencies, and enables more programs to type-check as fewer ambiguities arise.

The paper presents a type system and a type inference algorithm that includes: a constraint-set satisfiability function, that determines whether a given set of constraints is entailed or not in a given context, focusing on issues related to decidability, a constraint-set improvement function, for filtering out constraints for which overloading has been resolved, and a context-reduction

*Corresponding author

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

function, for reducing constraint sets according to matching instances. A standard dictionary-style semantics for core Haskell is also presented.
Keywords: Ambiguity; Context-dependent overloading; Haskell

---

1. Introduction

This paper considers the problem of ambiguity in the context of constrained polymorphism.

We use *constrained polymorphism* to refer to the polymorphism originated by the combination of parametric polymorphism and context-dependent overloading.

Context-dependent overloading is characterized by the fact that overloading resolution in expressions (function calls) $e\,e'$ is based not only on the types of the function (e) and the argument (e'), but also on the context in which the expression ($e\,e'$) occurs. As result of this, constants can also be overloaded — for example, literals (like `1`, `2` etc.) can be used to represent fixed and arbitrary precision integers as well as fractional numbers (for instance, they can be used in expressions such as `1 + 2.0`) — and functions with types that differ only on the type of the result (for example, *read* functions can be overloaded, of types $String \rightarrow Bool$, $String \rightarrow Int$ etc., each taking a string and generating the denoted value in the corresponding type). In this way, context-dependent overloading allows overloading to have a less restrictive and more prominent role in the presence of parametric polymorphism, as explored mainly in the programming language Haskell.

Ambiguity is however a major concern in context-dependent overloading. The usual meaning of an *ambiguous expression* is, informally, an expression that has more than one meaning, or an expression that can be interpreted in two or more distinct ways.

A formalization of this, with respect to a language semantics definition by means of type derivations, defines that an expression e is ambiguous if there exist two or more type derivations that give the same type and may assign

distinct semantics values to e (in the following, $\Gamma \vdash e : \sigma$ specifies that type σ is derivable for expression e in typing context Γ, using the axioms and rules of the type system; $[\![\Gamma \vdash e : \sigma]\!]$ denotes the semantic value obtained by using such axioms and rules):

Definition 1 (Standard Ambiguity). An expression e is called *ambiguous* if there exist derivations Δ and Δ' of $[\![\Gamma \vdash e : \sigma]\!]$ and of $[\![\Gamma' \vdash e : \sigma]\!]$, respectively, such that $[\![\Gamma \vdash e : \sigma]\!] \neq [\![\Gamma' \vdash e : \sigma]\!]$, where Γ and Γ' give the same type to every x free in e.

This is equivalent to defining that an expression e is ambiguous if it prevents the definition of a coherent semantics to e [1, page 286], that is, a semantics defined by induction on the structure of expressions where the semantic value assigned to a well-typed expression is not independent of the type derivation.

Without an explicit reference to a distinct definition, ambiguous refers to the standard definition above.

Detection of ambiguity is usually done at compile-time, by the compiler type analysis phase — in Haskell, by the type inference algorithm. Unfortunately, however, detection of ambiguity can not be based on type system definitions, at least for usual definitions, that allow context-free type instantiations, that is, type instantiations that can be done independently of the context where an expression occurs. This causes a well-known incompleteness problem for usual definitions of Haskell type systems [2, 3, 4]. This problem is not the focus of this paper.

This paper concentrates instead on another issue related to ambiguity in Haskell, which has not received attention in the technical literature, namely the relation between ambiguity and overloading resolution in the context of constrained polymorphism, in particular the fact that the possibility of inserting new (instance) definitions disregards that an expression may be disambiguated by occurring in some context where there exists a single instance which can be used to instantiate type variables that do not occur in the simple type component of the constrained type.

Specifically, our contributions are:

- A precise characterization of overloading resolution and ambiguity.
- Discussion of Haskell's open-world definition of ambiguity and proposal of a new definition, called delayed-closure ambiguity, that is distinguished from overloading resolution: in the open-world approach, ambiguity is a syntactic property of a type, not distinguished from overloading resolution, whereas with delayed-closure this syntactic property (existence of unreachable variables in constraints) characterizes overloading resolution, and ambiguity is a property depending on the context where the relevant expression occurs, namely the existence of two or more instances that entail the constraint with unreachable variables. Ambiguity is tested only after overloading resolution.

In Section 2 we present Haskell's definition of ambiguity, called open-world ambiguity. In Section 3 we compare open-world ambiguity with the standard, semantical notion of ambiguity.

Substitutions, denoted by meta-variable ϕ, possibly primed or subscripted, are used throughout the paper. A substitution denotes a function from type variables to simple type expressions. $\phi\,\sigma$ and $\phi(\sigma)$ denote the capture-free operation of substituting $\phi(\alpha)$ for each free occurrence of type variable α in σ, and analogously for the application of substitutions to constraints, sets of types and sets of constraints.

Symbol $\circ$ denotes function composition, and $dom(\phi) = \{\alpha \mid \phi(\alpha) \neq \alpha\}$ and id denotes the identity substitution. The restriction $\phi|_V$ of ϕ to V denotes the substitution ϕ' such that $\phi'(\alpha) = \phi(\alpha)$ if $\alpha \in V$, otherwise α.

A substitution ϕ is more general than another ϕ', written $\phi \leq \phi'$, if there exists ϕ_1 such that $\phi = \phi_1 \circ \phi'$.

Section 4 presents an alternative definition of ambiguity, called *delayed-closure* ambiguity, that specifies essentially that:

1. Ambiguity should be checked when (and only when) overloading is resolved. We identify that overloading is resolved in a constraint on the

type of an expression by the presence of unreachable variables in this constraint (overloading resolution is defined formally in Section 2). A type variable that occurs in a constraint is called reachable if it occurs in the simple type or in a constraint where another reachable type variable occurs, otherwise unreachable.

This is unlike open-world ambiguity, where the existence of any type variable that does not occur in the simple type component of a constrained type implies, in the abscence of functional dependencies (see below), ambiguity . For example, type $Coll\ \ c\ \ e\ \Rightarrow\ c$ of an *empty* member of a class $Coll\ c\ e$, is considered ambiguous in Haskell, since type variable e does not occur in the simple type component of the constrained type $Coll\ c\ e \Rightarrow c$ (despite being reachable). In delayed-closure ambiguity, types with only reachable type variables are not checked for ambiguity, since overloading is still unresolved and may be resolved later, depending on a program context in which it occurs.

2. Constraints with unreachable type variables may be removed if there exists only a single satisfying substitution that can be used to instantiate the unreachable type variables.

   An important observation is that such constraints, removed by the existence of a single satisfying substitution, become ambiguous by the addition of further instances if a satisfying substitution exists for a removed constraint with respect to the instances that have been added.

   The specification of defaults, as proposed in subsection 4.2, allows programmers to avoid types to become ambiguous by the addition of further instances.

Section 5 contains a description of constraint set satisfiability, focusing on issues related to decidability. Section 6 presents a type system for a core-Haskell language that adopts delayed-closure ambiguity. Section 7 presents a type inference algorithm for core-Haskell and discusses soundness and completeness of the type inference algorithm with respect to the type system. Section 8 presents a a

standard dictionary-style semantics for core Haskell. Section 9 discusses related work and Section 10 summarizes our conclusions.

2. Open-world ambiguity

The support of overloading in Haskell is based on the definition of *type classes*. A type class declaration specifies names or symbols, called class members, and their corresponding types. Several definitions of these names can be given, each one in an *instance definition*. Each definition of a name x, in an instance definition, must have a type that is an instance of the type given to x in the type class declaration.

Consider, for example, a declaration of type class *Eq* that defines symbols `(==)` and `(/=)` and their types, for comparing if two values are equal or not, respectively:

```
class Eq a where
  (==) ::  a → a → Bool
  (/=) ::  a → a → Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

The class declaration of *Eq* specifies also so-called *default* definitions. A default definition of a name x is assumed to be given in an instance definition that does not specify itself a definition for x.

Instances of type class *Eq* defining equality and inequality of operations, denoted by `(==)` and `(/=)`, for values of types *Int* and *Bool*, can then be given as follows, assuming that *primEqInt* is a primitive function for testing equality of values of type *Int*:

```
instance Eq Int where
  (==) = primEqInt

instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

It is well-known that it is possible to explore infinitary constrained polymorphism in Haskell, for example by defining equality for an infinite number of types of lists, as follows:

```
instance Eq a => Eq [a] where
  []     == []     = True
  (a:x) == (b:y) = (a==b) && (x==y)
  _      == _      = False
```

As a consequence of this instance definition, every list formed by elements which can be compared for equality can itself be compared for equality.

Polymorphic functions may be defined by the use of polymorphic overloaded symbols; for example:

```
member _ []     = False
member a (b:x) = (a == b) || member a x
```

The type of *member* is $\forall a\,.\ Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$. Constraint $Eq\ a$ restricts *member* to be applied only for types that are instances of type class *Eq*.

A type class can be defined as a subclass of an existing type class. For example:

```
class Eq a ⇒ Ord a where
  (>),(>=),(<),(<=):: a → a → Bool
```

defines *Ord* as a subclass of *Eq*, which means that every type that is an instance of *Ord* must also be an instance of *Eq*. Consider the following example:

```
search _ []     = False
search a (b:x)
  | (a==b)      = True
  | (a<b)       = False
  | otherwise   = search a x
```

The type of *search* is $\forall a\,.Ord\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$.

The fact that *Eq* is a subclass of *Ord* enables the constraint on the type of *search* to be *Ord a*, instead of (*Ord a*, *Eq a*). Constraint *Eq a* need not be explicitly included, because it is implied by the constraint *Ord a*.

$\overline{x}$ 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 ($\{x_1, \ldots, x_n\}$).

In general, in a constrained type $\forall\, \overline{a}.\, C \Rightarrow \tau$, C is a set of constraints, that restricts the set of types to which $\forall\, \overline{a}.\, \tau$ may be instantiated, so that every instance $\tau[\overline{a} \mapsto \overline{\tau}]$ is satisfiable in the program theory, where $\overline{a} = a_1, \ldots, a_n, \overline{\tau} = \tau_1, \ldots, \tau_n$ and $\tau[\overline{a} \mapsto \overline{\tau}]$ denotes the simultaneous substitution of a_i by τ_i in τ, for $i = 1, \ldots, n$. Notation $\tau[\overline{a} \mapsto \overline{\tau}]$ is defined similarly for quantified types σ ($\sigma[\overline{a} \mapsto \overline{\tau}]$) and for constraints. Constraint-set satisfiability is discussed in Section 5.

Ambiguity in Haskell is considered as a syntactic property on types of expressions. The definition of this property has been changing over time, since Haskell 98, which supports only single parameter type classes (it has remained the same in Haskell 2010): for single parameter type classes, ambiguity of a constrained type $\forall\, \overline{a}.\, C \Rightarrow \tau$ is characterized simply by $fv(C) \not\subseteq fv(\tau)$ (i.e. by the

fact that there is a type variable that occurs in C but not in τ) [2, 5].

In the sequel we consider multi-parameter type classes (MPTCs), and Haskell as it is defined in GHC [6] with extensions related to MPTCs (when we refer to standard Haskell, we mean Haskell 98 or Haskell 2010). MPTCs are recognized as a natural extension to Haskell, that should be incorporated into the language. This has been recognized as early as in the original paper related to type classes [7]. This has not happened, however, mainly because of problems related to ambiguity, namely that the use of overloaded symbols were thought to introduce expressions with ambiguous types.

In order to introduce support for MPTCs, the definition of ambiguity in GHC was changed so that ambiguity could be avoided. Ambiguity of a constrained type $C \Rightarrow \tau$ was changed to a definition based on the property of reachability of type variables occurring in C, from the set of type variables occurring in the simple type τ, where reachability is defined as follows:

Definition 2. A variable $a \in \mathit{fv}(C)$ is called reachable from, or with respect to, a set of type variables V if $a \in V$ or if $a \in \pi$ for some $\pi \in C$ such that there exists $b \in \mathit{fv}(\pi)$ such that b is reachable. $a \in \mathit{fv}(C)$ is called unreachable if it is not reachable.

The set of reachable and unreachable type variables of constraint set C *from* V are denoted respectively by $\mathit{reachableVars}(C, V)$ and $\mathit{unreachableVars}(C, V)$.

The subset of constraints with reachable and of unreachable type variables of constraint set C *from* V are denoted respectively by C_V^r and C_V^u.

We also say that type variables W are reachable in constrained type $C \Rightarrow \tau$ if $W \subseteq \mathit{reachableVars}(C, \mathit{fv}(\tau))$ (and similarly for unreachable type variables and if W is a type variable instead of a set of type variables).

For example, for type $\sigma = \forall c, e.\, \mathit{Coll}\ c\ e \Rightarrow c$, variable e is reachable from $\{c\}$, the set of type variables of the simple type (c) of σ; for type $\sigma = \forall a.\, (\mathit{Show}\ a, \mathit{Read}\ a) \Rightarrow (\mathit{String} \rightarrow \mathit{String})$, variable a is unreachable from the empty set of type variables of the simple type ($\mathit{String} \rightarrow \mathit{String}$) of σ.

It is easy to see that, for all C, V we have that:

$$fv(C) = reachableVars(C, V) \cup unreachableVars(C, V) \quad \text{and}$$
$$reachableVars(C, V) \cap unreachableVars(C, V) = \emptyset.$$

For example, both type variables a and b are reachable in constrained type $(F\ a\ b, O\ a) \Rightarrow b$, since b occurs in the simple type part (b), and a occurs in the constraint $F\ a\ b$, which contains b.

We also use, in this paper, the following:

Definition 3. Overloading (of symbols that originate the constraints) in constraint set D occurring in an expression with a constrained type $C \Rightarrow \tau$ is resolved if all type variables in $D \subseteq C$ are unreachable from $fv(\tau)$.

For example, overloading in constraint set $\{F\ a\ b, O\ a\}$, as well as in both constraints in this constraint set, of type $(F\ a\ b, O\ a) \Rightarrow b$ is yet unresolved, and overloading is resolved for any constraint set that occurs in a constraint set on a type where the simple type has no type variables, as $\{Show\ a, Read\ a\}$ on $\forall a.\, (Show\ a, Read\ a) \Rightarrow String \rightarrow String$.

The distinction between reachable and unreachable type variables in constraints on types of an expression is relevant because unreachable type variables can never be instantiated by unification with some other type, due to occurrence of this expression in some context.

GHC defines an expression as ambiguous by ambiguity of its type $C \Rightarrow \tau$, which does not mean simply the existence of an unreachable variable in C, with respect to the set of type variables occurring in τ, but takes into account the use of functional dependencies [8, 9, 10]. Following Haskell's open-world assumption, according to which instances may be added to a well-typed program without causing a type error, ambiguity of a constrained type $C \Rightarrow \tau$ is characterized by the existence of a type variable in C that is not *uniquely determined* from the set of type variables in the simple type τ [11].

Informally, this unique determination specifies that, for each type variable α that is in C but not in τ, there must exist a functional dependency $\beta \mapsto \alpha$,

for some β in τ (or a similar unique determination specified via type families, instead of functional dependencies). In this paper we use $\beta \mapsto \alpha$, instead of $\beta \to \alpha$, used in Haskell, to indicate a functional dependency (to avoid confusion with the notation used to denote functions).

This unique determination has been formalized in [8, 10], upon which the formalization of open-world ambiguity below is based.

Consider that:

1. sequences of constraints and of types can be indexed directly by type class parameters (i.e. type variable names), taken into account that to each type class parameter there is a corresponding integer, which gives its position in the class declaration;
2. $X \mapsto Y$ denotes a functional dependency from the set of type variables X to the set of type variables Y, specifying that the values in Y are determined by those in X;
3. $Fd(A)$ denotes the set of functional dependencies of type class A.

Then, for any constraint set C, there is a set of *induced functional dependencies* of C, given by:

$$I_{\mathrm{Fd}}(C) = \{fv(\overline{t_X}) \mapsto fv(\overline{t_Y}) \mid A\,\overline{t} \in C, (X \mapsto Y) \in Fd(A)\}$$

The transitive closure of V with respect to $I_{\mathrm{Fd}}(C)$, denoted by $V^+_{I_{\mathrm{Fd}}(C)}$, defines set of type variables in C that are *uniquely determined* from V.

For example, given `class F a b | b ↦ a where ...` (that specifies functional dependency $b \mapsto a$), we have that $\{b\}^+_{I_{\mathrm{Fd}}(F)} = \{a, b\}$.

We have:

Definition 4 (Open-world ambiguity). A type $\forall\,\overline{a}.\,C \Rightarrow \tau$ is called *open-world ambiguous* (abbreviated as *ow-ambiguous*) if $(\overline{a} \cap fv(P)) \not\subseteq fv(\tau)^+_{I_{\mathrm{Fd}}(C)}$.

For example, constrained type $(F\,a\,b, O\,a) \Rightarrow b$ is ow-ambiguous. To prevent

this ambiguity, programmers can use a functional dependency ($b \mapsto a$) in the declaration of class F.

Figuring out which functional dependencies (or type functions) need to be specified for dealing with ambiguity errors can be avoided with delayed-closure ambiguity, as explained in Section 4.

We define now the set of constraints formed by class and instance declarations that occur in a program, called a program theory (a term borrowed from [12]), and constraint set provability (entailment), in a program theory.

Definition 5. 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

  $$\texttt{class}\ C\ \Rightarrow\ TC\ a_1\ \ldots\ a_n\ \texttt{where}\ \ldots$$

  the program theory contains the following formula if C is not empty:

  $$\forall\, \overline{a}.\, C \Rightarrow TC\, \overline{a}$$

  where $\overline{a} = a_1\ \ldots\ a_n$.

- For each instance declaration

  $$\texttt{instance}\ C\ \Rightarrow\ TC\ t_1\ \ldots\ t_n\ \texttt{where}\ \ldots$$

  the program theory contains the following formula:

  $$\forall\, \overline{a}.\, C \Rightarrow TC\, t_1\ \ldots\ t_n$$

  where $\overline{a} = \mathit{fv}(t_1) \cup \ldots \mathit{fv}(t_n) \cup \mathit{fv}(C)$; if C is empty, then the instance declaration is of the form

  $$\texttt{instance}\ TC\ t_1\ \ldots\ t_n\ \texttt{where}\ \ldots$$

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

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

Figure 1: Constraint Set Entailment

and the program theory contains the formula:

$$\forall\, \overline{a}.\, TC\, t_1\ \ldots\ t_n$$

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 1. Following [13, 14], entailment is obtained from closed constraints contained in a program theory P.

3. Ambiguity and constrained polymorphism

Both the open-world and the standard definitions consider as ambiguous an expression e that might be used in a program context where distinct instances exist for an overloaded symbol that occurs in e. The motivation for this is that a coherent semantics for e, obtained by using type derivations that derive the type obtained by considering the chosen instance types are selected for each overloaded symbol, does not exist (because distinct semantics values could be given by considering such distinct instances).

However:

1. if the expression is effectively used in a context where overloading is resolved and there is a constraint on the expression's type for which distinct instances exist, then, and only then, a type-error, characterizing ambiguity, can be detected;
2. the expression may be used only in contexts where overloading is resolved in such a way that there exists a single instance for the type of each overloaded symbol used in the expression.

This indicates a prematureness of ambiguity detection because of the possibility of an expression being used in a context where two distinct types exist for some used overloaded symbol. Such possibility is what both the open-world and the standard definitions of ambiguity consider, albeit in different ways, as shown in the remainder of this section.

The standard definition of ambiguity considers the existing instances but closes the world for expressions without considering whether overloading has been resolved or not. We consider an example below (Example 2). The open-world definition of ambiguity disregards the existence or not of instances in the relevant context, and considers ambiguity by the possibility of inserting any instance definitions.

Example 1. Consider *the* canonical ambiguous expression in Haskell (cf. e.g. [2, 4]): `(`*show* `.` *read*`)`, called e_0 for further reference (where `"."` denotes function composition).

This expression is considered ambiguous in Haskell, irrespective of the context in which it occurs. This is related to the fact that instances definitions are global, i.e. are always present in any scope. However, defaults could be specified (in this example, for *Show*, *Read*) in order to avoid ambiguity. In standard Haskell, defaults are restricted, in a rather ad-hoc way, for constraint sets that include a constraint on class *Num*. Subsection 4.2 describes the use of defaults for avoiding ambiguity of constraint sets, and considers an extension for the use of defaults under delayed-closure ambiguity (Section 4). Under delayed-closure ambiguity, the type of e_0 is $String \rightarrow String$ if the context has only one instance of *Show* and *Read*; otherwise there is a type error (unsatisfiabilty if there is no such instance, ambiguity if there are two or more).

The fact that the simple type in the type of e_0 cannot be changed by placing e_0 in another context characterizes that ambiguity (and unsatisfiability) of e_0 should be checked, that is, it should be verified whether there exists or not only one instance that satisfies the constraints on the type of the expression. If there is only one instance, the constraints are satisfied, and can then be removed from

the type of the expression (in the example, the type of `(`*show* `.` *read*`)` can be simplified from $\forall a.\,(Show\ a, Read\ a) \Rightarrow (String \rightarrow String)$ to $String \rightarrow String$.

Although both open-world and standard definitions of ambiguity both disregard whether overloading is or is not yet resolved and both anticipate the test of ambiguity, they disagree in key aspects: there are expressions that are unambiguous according to the standard definition but ow-ambiguous and vice-versa.

Examples of the first case occur both when overloading is and is not resolved. e_0 is an example of when overloading is resolved: it is always ow-ambiguous, and standard ambiguity depends on the existence of two or more instances of `(`*Show* *a*`,` *Read* *a*`)` (in this case, the standard definition agrees with delayed-closure, presented in the next section). The following is an example of an expression for which overloading is not yet resolved (and is therefore not ambiguous according to the delayed-closure approach), that is ow-ambiguous and can be ambiguous or not according to the standard interpretation.

Example 2. Consider expression `((+) 1)` — which in Haskell can be written as `(1+)` —, in a program with classes *Sum* and *NumLit*, given in a program with the following classes, where `1` is considered to have type $\forall a.\, NumLit\ a \Rightarrow a$:

```
class Sum a b c where
  (+):: a → b → c
class NumLit a where ...
```

We have that `(1+)` is considered ow-ambiguous, but the standard definition would consider it ambiguous only if there exist two or more instances of the type of `(1+)`, namely $\forall a, b, c.\,(NumLit\ a, Sum\ a\ b\ c) \Rightarrow b \rightarrow c$, in the program. For example, let P_0 be a program theory that contains instances *Sum Int Float Float*, *Sum Float Float Float*, *NumLit Int* and *NumLit Float*. Then `(1+)` is considered ambiguous, according to the standard definition, in P_0.

Both open-world and standard ambiguity disregard that overloading is not yet resolved, and that the test of ambiguity should not be done yet, since the

type of the polymorphic expression `(1+)` can still change depending on the context where it is used (in this case, both disagree with delayed-closure, described in the next section; remember: overloading is not yet resolved for an expression of type $\forall a, b, c.\, (NumLit\ a, Sum\ a\, b\, c) \Rightarrow b \to c$).

4. Delayed-closure ambiguity

In this section we present an approach for dealing with ambiguity in Haskell that uses the presence of unreachable variables in a constraint for characterizing overloading resolution (or, more precisely, for characterizing that overloading should have been resolved), instead of characterizing ambiguity.

Informally, instead of issuing an ambiguity error, the presence of an unreachable type variable a from the set of type variables in $fv(\tau)$, in a constrained type $\forall\, \overline{a}.\, C \Rightarrow \tau$, triggers a test of whether this variable can be instantiated (i.e. eliminated), because of the existence of a single instance that can be used to instantiate it. We use the following for this.

Definition 6. Consider constrained type $C \Rightarrow \tau$, program theory P, and that type variable a occurs in $\pi \in C$ and is unreachable with respect to $fv(\tau)$ and consider a substitution ϕ, with domain restricted to unreachable type variables in C, such that $P \vdash_e \phi(C)$. Then ϕ is called a *satisfying substitution for C in P. A unique satisfying substitution for C in P is called an improvement substitution of C in P and $\phi(C)$ the improved constraint.*

We can now define delayed-closure ambiguity, as follows.

Definition 7. Type $\forall\, \overline{a}.\, C \Rightarrow \tau$ is *delayed-closure ambiguous*, with respect to a program theory P, if $unreachableVars(C, fv(\tau)) \neq \emptyset$ and there exists at least two satisfying substitutions for C in P.

If there exists no satisfying substitution for C in P then C is called unsatisfiable in P.

We call *improvement* (cf. [15]) the process of substituting a constrained type $C \Rightarrow \tau$, in a given program theory P, by $\phi(C) \Rightarrow \tau$, where $\phi(C)$ is the improved constraint of C in P.

Example 3. Consider type `(`$F\ a\ Bool$`)` $\Rightarrow$ $Bool$, in a program with the following instances (forming a program theory P):

```
instance C a => F a Bool where ...
instance C Char where ...
```

Substitution $(a \mapsto Char)$ is the improvement substitution of `(`$F\ a\ Bool$`)` in P, and `(`$F\ Char\ Bool$`)` is the improved constraint of `(`$F\ a\ Bool$`)` in P.

Example 4. Consider a classical example of MPTCs with functional dependencies[1], namely matrix multiplication.

Consider the following types:

```
data Vector = Vector Int Int
data Matrix = Matrix Vector Vector
```

Consider also the following:

```
class Mult a b c where
  (*) ::  a -> b -> c
instance Mult Matrix Matrix Matrix where ...
instance Mult Matrix Vector Vector where ...
```

The type of x in the following example:

```
m1, m2, m3:: Matrix
x = (m1 * m2) * m3
```

[1] Taken from `www.haskell.org/haskellwiki/Functional_dependencies`.

is inferred to be $Mult\ Matrix\ Matrix\ a, Mult\ a\ Matrix\ b \Rightarrow b$.

Type variable a is reachable from b, and thus, with delayed-closure ambiguity, there is no checking for ambiguity (no ambiguity arises, relieving the programmer from having to figure out if and which functional dependencies could solve the problem).

If x is used in a context that type *Matrix* is inferred for it (or of course if x is declared to be of type *Matrix*), then the type:

$$Mult\ Matrix\ Matrix\ a, Mult\ a\ Matrix\ Matrix \Rightarrow Matrix$$

can be simplified to *Matrix*, if there is a single instance of ($Mult\ Matrix\ Matrix\ a$) in the current context; if another instance of ($Mult\ Matrix\ Matrix\ a$) exists in this context, then we have ambiguity (in this case, the compiler can report a helpful error message, informing that there are two or more instances of $Mult\ Matrix\ Matrix\ a$ in the context).

The possibility of adding further instances is possible until overloading resolution. This happens when the context cannot anymore change (instantiate) the type of the expression, because of the existence of unreachable type variables in the constraint. The programmer can then rely on the type inference algorithm to instantiate unreachable type variables (whenever there exists a single instance in the context for such instantiation), relieving the programmer from having to figure out if and which functional dependencies could solve his problem.

4.1. Discussion

Haskell's open-world has a notable characteristic that a well-typed program never becomes untypeable by the introduction of new instance declarations. Delayed-closure restricts this advantage to expressions for which overloading is not resolved; when overloading is resolved, the world is closed, i.e. existing definitions of overloaded names are considered, by checking ambiguity and unsatisfiability.

This seems a significant disadvantage, but let us consider further aspects. With delayed-closure ambiguity more programs become well-typed and ambi-

guity becomes easier to understand: under delayed-closure, ambiguity is not a syntactic property of a type, and it does not mean a possibility, of using an expression in a context where two or more instances for the type of the expression *might* exist. It means the actual fact that there exist two or more instances, when overloading is resolved. This agrees with the usual understanding of ambiguity in a natural language, that considers ambiguity for concrete sentences, that may be interpreted in distinct ways. In our view the most important aspect is that ambiguity is distinguished from overloading resolution. Ambiguity is tested only after overloading resolution. The notion of unsatisfiability becomes a related notion, that refers to the nonexistence of instances for entailment of a constraint set. Variables in a constraint are either all reachable or all unreachable. If they are unreachable, the constraint can be removed (in the case of single entailment) or there is a type-error (ambiguity in cases of two or more, unsatisfiability in cases of no entailment). Another slight counter weight in favour of delayed-closure ambiguity is the fact that it yields a more symmetric treatment: for expressions for which overloading is resolved, removal of an instance declaration may cause unsatisfiability and insertion may cause ambiguity.

The use of delayed-closure ambiguity in Haskell would benefit by another significant change: the ability to control exportation and importation of instances in modules. There are several proposals for doing this (see e.g. [16, 17, 18]), but this is left for future work.

We discuss next the specification of defaults for constraint sets in Haskell.

4.2. Defaults

Defaults can be specified in standard Haskell but only for constraint sets where all constraints consist of classes declared in the Haskell Prelude and one of them is class *Num*. Consequences of this are that predefined classes in general and class *Num* in particular have to be distinguished by Haskell compilers and, more significantly, an exceptional rule is created, without a strong technical reason for restricting defaults to specific classes. The motivation is to avoid

some frequent uses of type annotations.

Distinct proposals related to changing the way of handling defaults in GHC can be consulted at:

`http://ghc.haskell.org/trac/haskell-prime/wiki/Defaulting`

These include a proposal for removing the possibility of specifying defaults altogether. We basically follow the basic proposal (number 2) related to the possibility of specifying defaults for MPTCs.

A default clause should be in our view a top level declaration (like class and instance declarations) to be applied only within the module containing the declaration, and it should not be possible to either export or import defaults. The relevant issue here is to disallow a change in the behavior of a module because of a change in which modules are imported.

A default clause may specify a default for a constraint, which may be a type expression (not only a type) of any kind. For example, we can have:

```
default (Read a) Int
default (Monad m) []
```

Default application is only considered for constraint sets with unreachable variables, and the only result of applying defaults is the removal of constraints (since a constraint which contains an unreachable type variable can only contain unreachable type variables).

A constraint set C can be removed from $(A\,\tau_1\,\ldots\,\tau_n), D \Rightarrow \tau$ by application of a default if and only if the following conditions hold:

1. there is a default clause of the form `default (`$A\ \overline{\tau}$`)` $\overline{\rho}$ in the current module,
2. C is of the form $(A\,\overline{\epsilon}, C')$, for some $C' \subseteq D$ such that each constraint in C has unreachable type variables, $fv(C') \subseteq fv(A\,\overline{\epsilon})$, and there exists a substitution ϕ of C in the program theory such that $\phi(\overline{\tau}) = \overline{\rho}$.

For example, considering default clause `default (`*Read a*`)` *Int*, and that $C = \{Read\ a\}$, $D = \{Show\ a\}$, we have that the substitution ϕ must be such that $\phi(a) = Int$.

5. Satisfiability

This section contains a description of constraint set satisfiability, including a discussion of decidability, based on work already presented in [19].

Following [15], $\lfloor C \rfloor_P$ is used to denote the set of satisfiable instances of constraint set C with respect to program theory P:

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

Example 5. As an example, consider:

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

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

A proof that $P \vdash_e \{C\ [Bool]\ [Bool]\}$ holds can be given from the entailment rules given in Figure 1, since this is the conclusion of rule ($\mathtt{mp}_0$) with premises $P \vdash_e \{D\ Bool\ [Bool]\}$ and $P \vdash_e \{D\ Bool\ [Bool] \Rightarrow C\ [Bool]\ [Bool]\}$, and these two premises can be derived by using rule ($\mathtt{inst}_0$).

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

Constraint set satisfiability is in general an undecidable problem [20]. It is restricted in this work so that it becomes decidable, as described below.

The restriction is based on a measure of constraints, given by a so-called constraint-head-value function, based on a measure of the sizes of types in

this constraint head. Essentially, 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.

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:

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

Consider computation of satisfiability of a given constraint set 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 $\forall\, \overline{\alpha}.C_0 \Rightarrow \pi_0$ 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, in this argument position, of constraints that unify with π_0 decreases.

More precisely, constrain-head-value-function Φ associates a pair (I, Π) to each constraint $(\forall\, \overline{\alpha}.P_0 \Rightarrow \pi_0) \in P$, 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 $\forall\, \overline{\alpha}.\, P_0 \Rightarrow \pi_0 \in P$, 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 π in the program theory.

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 $\pi = C\,\tau_1 \cdots \tau_n$:

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

where $\Phi'(\pi_0) = ((v_0', v_1', \ldots, v_n'), \Pi \cup \{\pi\})$

$\Phi'(x) = \Phi(x)$ for $x \neq \pi_0$

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

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

Let $\mathit{sats}_1(\pi, P, \Delta)$ hold if

$$\Delta = \left\{ (\phi|_{fv(\pi)}, \phi C_0, \pi_0) \;\middle|\; \begin{array}{l} (\forall\, \overline{\alpha}.\, C_0 \Rightarrow \pi_0) \in P, \\ \mathit{mgu}(\pi = \pi_0, \phi) \text{ holds} \end{array} \right\}$$

where *mgu* is the most general (least) unifier relation[21]: $\mathit{mgu}(\mathcal{T}, \phi)$ is defined to hold between a set of pairs of simple types or constraints $\mathcal{T}$ 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 analogously for pairs of simple constraints $(\pi, \pi') \in \mathcal{T}$), 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 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 2.

The following examples illustrate the definition of constraint set satisfiability as defined in Figure 2. Let $\Phi(\pi).I$ and $\Phi(\pi).\Pi$ denote the first and second components of $\Phi(\pi)$, respectively.

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

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

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

Figure 2: Decidable Constraint Set Satisfiability

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

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

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:

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

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:

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

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

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

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

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

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 $\phi' = [a_3 \mapsto T^4\,\mathtt{I}, b_2 \mapsto \mathtt{I}]$, 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 $\pi_3 = A\,(T^6\,\mathtt{I})\mathtt{I}$, that unifies with $A\,(T\,a)\,\mathtt{I}$, 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 8. Consider the satisfiability of $\pi = A\,(T^2\,\mathtt{I})\,\mathtt{F}$ in program theory $P = \{A\,\mathtt{I}\,(T^2\,\mathtt{F}), \forall\, a, b.\, A\,a\,(T\,b) \Rightarrow A\,(T\,a)\,b\}$ and let $\pi_0 = A\,(T\,a)\,b$. Then:

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

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

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

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

Since satisfiability of type class constraints is in general undecidable [20], there exist satisfiable instances which are considered to be unsatisfiable according to the definition of Figure 2. Examples can be constructed by encoding instances of solvable Post Correspondence problems by means of constraint set satisfiability, using G. Smith's scheme [20].

To prove that satisfiability as defined in Figure 2 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.

5.1. Improvement

In this paper, improvement filters out constraints with unreachable type variables (remember that the presence of unreachable type variables in a constraint is an indication that overloading has been resolved) from a constraint C,

$$\frac{V = fv(\tau) \;\; C_V^u \vdash^{P,\Phi_0}_{\texttt{sats}} \{\phi\}}{C \Rightarrow \tau \vdash^{P}_{\texttt{impr}} C_V^r \Rightarrow \tau}$$

Figure 3: Constraint Set Improvement

on a constrained type $C \Rightarrow \tau$. Improvement tests satisfiability on $C^u_{fv(\tau)}$ (the subset of constraints of C with unreachable type variables) and removes $C^u_{fv(\tau)}$ if each constraint in this subset has a single satisfying substitution.

If the set $\mathbb{S}$ of satisfiable instances of $C^u_{fv(\tau)}$ has more than one element, or if it is empty, there is no improved constraint (improvement is a partial relation). Improvement is defined in Figure 3 (Φ_0 is as defined in section 5, page 22).

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 the relevant 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, 5, 3].

Context reduction uses *matches*, defined as follows:

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

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

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

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

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

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

Figure 4: Context Reduction

The third parameter of *matches* is either empty or a singleton set, since overlapping instances [22] are not considered.

Context reduction, defined in Figure 4, 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 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.

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

Figure 5: Context-free syntax of core Haskell expressions

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

6. Type system

In this section we present a type system for a core-Haskell language that adopts delayed-closure ambiguity.

We use a context-free syntax of core Haskell expressions, given in Figure 5, where meta-variable x represents a variable. Meta-variables x, y, z denote variables and e an expression, possibly primed or subscripted. We call the language core Haskell (not core ML) because expressions are considered to be typed in a program theory (as defined in Section 1), with information about overloaded symbols generated from class and instance declarations.

A context-free syntax of constrained types is presented in Figure 6, where meta-variable usage is also indicated. For simplicity and following common practice, kinds are not considered in type expressions (and thus type expressions which are not simple types are not distinguished from simple types). Also, type expression variables are called simply type variables.

We assume that a program theory is part of a typing context Γ, and can be denoted by P_Γ. The initial, global typing context under which program expressions are considered to be typed contain all assumptions $x : \sigma$, where x is a member of a type class A (declared as $\texttt{class}\ C \Rightarrow\ A\, \overline{\alpha}\ \texttt{where}\ \ldots x\ \texttt{::}\ \tau \ldots$) and $\sigma = \forall\, \overline{\alpha}.\, A\, \overline{a} \Rightarrow \tau$ is the type obtained including in $\overline{\alpha}$ type variables in $fv(\tau) \cup \overline{\alpha} \cup fv(C)$.

We use:

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

Class Name	A, B	
Type variable	a, b, α, β	
Type constructor	T	
Simple Constraint	π	$::= A\,\overline{\tau}$
Unquantified Constraint	ψ	$::= C \Rightarrow \pi$
Constraint	θ	$::= \forall\,\overline{\alpha}.\,\psi$
Set of Unquantified Constraints	C, D	
Constrained Type	δ	$::= C \Rightarrow \tau$
Simple Type	τ, ρ, ϵ	$::= \alpha \mid T \mid \tau\ \tau$
Type	σ	$::= \forall\,\overline{\alpha}.\,\delta$
Program Theory	P, Q	

Figure 6: Types, constraints and meta-variable usage

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

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

Figure 7: Partial order on Types, Constraints and Typing Contexts

A partial order on types, constraints and typing contexts is defined in Figure 7.

Note that type ordering disregards constraint set satisfiability. Satisfiability is only important when considering whether a constraint set C can be removed from a constrained type $C, D \Rightarrow \tau$ (C can be removed if and only if overloading for C has been resolved and there exists a single satisfying substitution for C; see Figure 8).

A type system for core Haskell is presented in Figure 9, using rules of the form $\Gamma \vdash e : \psi$, which means that e has type ψ in typing context Γ. The rules are similar to those for core ML [23, 24, 25, 26], with differences in rules (APP) and (LET). Rule (LET) performs constraint set simplification before type generalization.

In rule (APP), the constraints on the type of the result are those that occur

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

Figure 8: Constraint set simplification

in the function type plus not all constraints that occur in the type of the argument but only those that have variables reachable from (the set of variables that occur in) the simple type of the result (this has been already defined in [19]). This allows, for example, not including constraints on the type of the following expressions, where o is any expression, with a possibly non-empty set of constraints on its type: *flip const o* (where *const* has type $\forall a, b.\, a \to b \to a$ and *flip* has type $\forall a, b, c.\, (a \to b \to c) \to b \to a \to c$), which should denote an identity function, and *fst* (e, o), which should have the same denotation as e.

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

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

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

Relation $\gg_P$ is a simplification relation on constraints, defined as a composition of improvement and context reduction, defined respectively in subsections 5.1 and 5.2.

7. Type inference

In this section we present a type inference algorithm for core-Haskell, and discuss soundness and completeness of type inference with respect to the type system.

A type inference algorithm for core Haskell is presented in Figure 10, using rules of the form $\Gamma \vdash_i e : (\psi, \phi)$, which means that ψ is the least (principal) type of (derivable for) e in typing context $\phi\Gamma$, where $\phi\Gamma \leq \Gamma$ and, whenever $\Gamma' \leq \Gamma$ is such that $\Gamma' \vdash_i e : (\psi', \phi')$, we have that $\phi\Gamma \leq \Gamma'$ and $\psi' \leq \phi\psi$. Furthermore, we have that $\phi\,\Gamma \vdash_i e : (\psi, \phi')$ holds whenever $\Gamma \vdash_i e : (\psi, \phi)$ holds, where $\phi' \leq \phi$ (cf. theorem 1 below).

$$\frac{(x:\sigma)\in\Gamma \quad \sigma\leq\psi}{\Gamma\vdash x:\psi}\ (\textsc{var})$$

$$\frac{\Gamma, x:\tau\vdash e : C\Rightarrow\tau'}{\Gamma\vdash\lambda x.\, e : C\Rightarrow\tau\to\tau'}\ (\textsc{abs})$$

$$\frac{\begin{array}{c}\Gamma\vdash e : C\Rightarrow\tau'\to\tau \quad \Gamma\vdash e' : C'\Rightarrow\tau' \\ (C\oplus_{fv(\tau)} C')\gg_{P_\Gamma} D\end{array}}{\Gamma\vdash e\, e' : D\Rightarrow\tau}\ (\textsc{app})$$

$$\frac{\begin{array}{ll}\Gamma\vdash e\colon C\Rightarrow\tau & C\gg_{P_\Gamma} D \\ gen(D\Rightarrow\tau,\sigma,fv(\Gamma)) & \Gamma,\ x\colon\sigma\vdash e'\colon\psi\end{array}}{\Gamma\vdash \texttt{let}\ \ x=e\ \ \texttt{in}\ \ e' : \psi}\ (\textsc{let})$$

Figure 9: Type System

Example 9. Consider expression x and typing context $\Gamma = \{f : Int\to Int, x : \alpha\}$; we can derive $\Gamma\vdash_i f\ x : (Int,\phi)$, where $\phi = [\alpha\mapsto Int]$. From $\phi\,\Gamma = \{f : Int\to Int, x : Int\}$, we can derive $\phi\,\Gamma\vdash_i f\ x : (Int, id)$.

Theorem 1. *If $\Gamma\vdash_i e : (\psi,\phi)$ holds then $\phi\,\Gamma\vdash_i e : (\psi,\phi')$ holds, where $\phi'\leq\phi$.*

Furthermore, for all typing contexts Γ' with the same quantified type assumptions as Γ — i.e. for all Γ' such that $P'_\Gamma = P_\Gamma$ and for which $(x:\forall\alpha.\,\sigma)\in\Gamma'$ implies $(x:\forall\alpha.\,\sigma)\in\Gamma$ —, if $\Gamma'\vdash_i e : (\psi',\phi')$ is derivable, for some ψ',ϕ', we have that $\phi\Gamma\leq\Gamma'$, $\psi\leq\psi'$ and $\phi'\leq\phi$.

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

Theorem 2 (Soundness)**.** *If $\Gamma\vdash_i e : (\psi,\phi)$ holds then $\phi\Gamma\vdash e:\psi$ holds.*

Theorem 3 (Principal type inference)**.** *If $\Gamma\vdash_i e : (\psi,\phi)$ holds then, for all ψ' such that $\Gamma\vdash e:\psi'$ holds, we have that $\psi\leq\psi'$.*

A completeness theorem does not hold. For example, consider expression e_0

$$\frac{(x : \forall\, \overline{\alpha}.\, \psi) \in \Gamma \quad \overline{\beta} \text{ fresh}}{\Gamma \vdash_i x : (\psi[\overline{\alpha} \mapsto \overline{\beta}], id)}(\texttt{VAR}_i)$$

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

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

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

Figure 10: Type Inference

of Example 1; we have that there exists Γ such that $\Gamma \vdash e_0 : \mathit{String} \to \mathit{String}$ holds but there is no ψ, ϕ such that $\Gamma \vdash e_0 : (\psi, \phi)$ holds.

In our opinion, the greater simplicity obtained by allowing type instantiation to be done ("guessed") in a context-independent way, does not compensate the disadvantages of allowing ambiguous expressions to be well-typed and of having several translations for expressions, one of them a principal translation. We prefer a declarative specification of type inference, that allows a unique type to be derivable for each expression, where type instantiation is restricted to be done only in a context-dependent way, given by considering functions, used in the type inference algorithm, as relations. In other words, the type inference algorithm can be obtained from a declarative specification of type inference by transforming relations used into functions; see [19]. The fact that every element is an element of a unique type is a bonus that agrees with everyday spoken language. It is straightforward to define, a posteriori, the set of types that are instances of the type of an expression.

The fact that only a single type can be derived for each expression rules out the possibility of having distinct type derivations. Thus, an error message for an expression such as (*show* . *read*), in a context with more than one instance for *Show* and *Read*, should be that the expression can not be given (there is no type that would allow it to have) a well-defined semantics. Distinct meanings of (*show* . *read*) would be obtained from distinct instance types of *show*, *read*.

In the next section we give a semantics by induction on the derivation of the type of an expression by considering functions used in the type inference algorithm (mgu_I, *gen*, $\gg_P$) as relations.

8. Semantics

The semantics of core Haskell, given in Figure 11, follows a standard core Haskell semantics [7, 2, 5], based on the application of (so-called) *dictionaries* to names with constrained types. A dictionary is a tuple of denotations of definitions given in an instance declaration; in other words, denotations of *instance members*. 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. [2, 5, 3]).

A fundamental characteristic of core Haskell is that the semantics of an expression depends on its type. The semantics is defined below by induction on the rules of a type system where each variable occurrence has its type annotated (there is no guessing of types). The type annotated for a variable in rule ($\mathtt{VAR}_s$) is the greatest instance-type of the variable occurrence in the relevant typing and program contexts, as specified by Definition 8 below, where a program context $C[e]$ is an expression that has e as a subexpression. The type annotated for a lambda-bound variable is the type of the function parameter, and the type annotated for a let-bound variable is the type of its defining expression. Type annotations are indicated by a dot in bold face font (as in $x : \sigma$) in Figure 11.

Definition 8. $\phi'\psi$ is an *instance-type of* e *in* Γ (and an *instance-type of* e *in* Γ *and program context* $C[e]$) if both $\Gamma \vdash_i e : (\psi, \phi)$ and $\Gamma \vdash_i C[e] : (\psi', \phi')$ are

derivable, for some ϕ, ψ'.

Furthermore, ψ is the greatest (most specific) instance-type for e in Γ and program context $e' = C[e]$, modulo type variable renaming, denoted by $git(e, \Gamma, e')$, if ψ is an instance-type of e in Γ and program context $C[e]$ and there is no instance-type ψ' of e in Γ and program context $C'[e']$ such that ψ' is distinct from ψ and $\psi \leq \psi'$.

Example 10 below gives two distinct greatest instance-types of `(==)` in the same typing context and distinct program contexts (where `B` and `C` can be seen as abbreviations of *Bool* and *Char* respectively).

Example 10. Let $\{$`(==)` $: \forall a.\, Eq\ a \Rightarrow a \to a \to$ `B`, $True$: `B`, $'*'$: `C`$\} \subseteq \Gamma$, $P_\Gamma = \{Eq$ `B`, Eq `C`$\}$, $e =$ `((==)` $True$`,(==) '*')`. Then $\Gamma \vdash_i e : \big((\texttt{B} \to \texttt{B}, \texttt{C} \to \texttt{B}), \phi\big)$ is derivable, where $\phi = [a \mapsto \texttt{B}, b \mapsto \texttt{C}]$ and a, b are fresh type variables.

Instance-types of `(==)` in program contexts `(==)` $True$ and `(==) '*'` are respectively `B` $\to$ `B` $\to$ `B` and `C` $\to$ `C` $\to$ `B`.

Typing formulas have the form $\Gamma \vdash_a e : \psi$, except that types of variables are annotated, as mentioned. A formal description of how to compute the annotated type of variables is left for further work.

For each class declaration `class` $C \Rightarrow A\,\overline{\alpha}$ where $\overline{x} :: \overline{\tau}$, a selection function is generated for each overloaded name x_i in $\overline{x}$. Such name denotes, in the semantics, a function that merely selects the i-th component of a dictionary (tuple) parameter; if $n = 1$, selection corresponds to the identity function. For example, class Eq generates a pair of functions, with names (`(==)` and `(/=)`) that in the translation denote functions (*fst* and *snd*) for selecting respectively the first and second components of a pair.

$\overline{C}$ denotes a sequence of constraints of C in a standard (lexicographical) order.

Each instance declaration

$$\texttt{instance}\ C \Rightarrow \pi\ \texttt{where}\ \overline{x} = \overline{e}$$

generates either a dictionary or a dictionary constructor d_π, according to whether C is empty or not, respectively. A dictionary constructor takes as parameter one dictionary for each constraint in the sequence $\overline{C}$ and yields the dictionary of π. The instance declaration makes $\eta(\pi)$ equal to d_π and $\eta(x_i, \tau_i) = (d_\pi, C)$. If there is no instance declaration for π, we assume that $\eta(\pi)$ is undefined.

If x is (not an instance member but) a let-bound variable, then $\eta(x, \tau)$ is made equal (d_0, C), where a generalization of $C \Rightarrow \tau$ is the type of x; d_0 is an indication that no dictionary or dictionary constructor is associated to (x, τ); d_0 is used as indication that x is a let-bound variable of constrained type, not an instance member.

For example, for a typing context Γ associated to a program that defines instances of classes *Eq* for *Char* and lists, we have, where d_{EqChar} is a dictionary of class *Eq Char* and d_{EqL} is a dictionary constructor that constructs a dictionary of class *Eq* for lists of values of type a from a dictionary of values of (any) type a:

$$\begin{aligned} \eta(Eq\ Char) &= d_{EqChar} \\ \eta(Eq\,\texttt{[}a\texttt{]}) &= d_{EqL} \end{aligned}$$

where a is a fresh, arbitrary type variable, and :

$$\begin{aligned} \eta(\texttt{(==)}, Char \to Char \to Bool) &= (d_{EqChar}, \emptyset) \\ \eta(\texttt{(==)}, \texttt{[}a\texttt{]} \to \texttt{[}a\texttt{]} \to Bool) &= (d_{EqL}, \{Eq\ a\}) \end{aligned}$$

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

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

$\eta(x, \tau, \Gamma)$ gives the semantics of possibly overloaded name x with type $C \Rightarrow \tau$, for some C, in typing context Γ. The translation of an overloaded name x that is an instance member is a selection from a dictionary (possibly constructed by passing pertinent dictionaries to a dictionary constructor). Otherwise, the translation is not a selection but a function call that passes pertinent dictionaries, one for each constraint in the constraint set on the type of x, to an already defined function.

$$\frac{}{[\![\Gamma \vdash_a x \,:\, C \Rightarrow \tau]\!]_\eta = \eta(x,\tau,\Gamma) : \tau}(\texttt{VAR}_s)$$

$$\frac{[\![\Gamma, x \,:\, \tau' \vdash e : (C \Rightarrow \tau, \phi)]\!]_\eta = \mathbf{e} : \tau}{[\![\Gamma \vdash_a \lambda x.\, e : (C \Rightarrow \tau' \to \tau, S)]\!]_\eta = \lambda x.\, \mathbf{e} : \tau' \to \tau}(\texttt{ABS}_s)$$

$$\frac{\begin{array}{c}[\![\Gamma \vdash_a e : C \Rightarrow (\tau' \to \tau)]\!]_\eta = \mathbf{e} : \tau' \to \tau \\ [\![\Gamma \vdash_a e' : C' \Rightarrow \tau_2]\!]_\eta = \mathbf{e'} : \tau' \\ C|^*_{fv(\tau)} \gg_{P_\Gamma} D\end{array}}{[\![\Gamma \vdash_a e\, e' : D \Rightarrow \tau]\!]_\eta = \mathbf{e}\ \mathbf{e'} : \tau}(\texttt{APP}_s)$$

$$\frac{\begin{array}{c}[\![\Gamma \vdash_a e : C \Rightarrow \tau]\!]_{\eta_1} = \mathbf{e} : \tau_1 \\ C \gg_{P_\Gamma} D,\ \overline{v} = vSeq(D),\ gen\big(\sigma, D \Rightarrow \tau, fv(\Gamma)\big) \\ \eta_1 = \eta \dagger (D \mapsto \overline{v}),\ \eta' = \eta \dagger \Big((x,\tau) \mapsto \big(d_0, D\big)\Big) \\ [\![\Gamma,\, x{:}\sigma \vdash_a e' : D \Rightarrow \tau]\!]_{\eta'} = \mathbf{e'} : \tau\end{array}}{[\![\Gamma \vdash_a \texttt{let}\ \ x \,:\, \sigma = e\ \ \texttt{in}\ \ e' : \delta]\!]_\eta = \mathbf{let}\, x = \lambda\overline{v}.\, \mathbf{e}\ \mathbf{in}\ \mathbf{e'} : \tau}(\texttt{LET}_s)$$

Figure 11: Core Haskell Semantics

$\eta(C)$ denotes the sequence $\eta(\pi_1)\ldots\eta(\pi_n)$, where $\overline{C} = \pi_1\ldots\pi_n$; the dictionary for constraint π is given by from $\eta(\pi)$. We have:

$$\eta(x,\tau,\Gamma) = \begin{cases} x & \text{if } C_0 = \emptyset \\ x\,\overline{v} & \text{if } d = d_0 \\ x(d\,\overline{v}) & \text{otherwise} \end{cases}$$

where: $(\forall\,\overline{\alpha}.\, C_0 \Rightarrow \tau_0) = \Gamma(x)$

$\phi = mgu(\tau, \tau_0),\ \overline{v} = \eta(\phi D),\ (d, D) = \eta(x, \tau)$

Note that the constraint set C on x's type is disregarded in the semantics, which uses $\Gamma(x)$ to obtain the original constraint set used in the definition of x; simple type τ is used to instantiate the original constraint set. This is done because constraints removed from C due to overloading resolution must be considered in the semantics. We have:

Theorem 4. *For any derivations Δ, Δ' of typing formulas $\Gamma \vdash_a e : \phi$ and $\Gamma' \vdash_a e : \phi$, respectively, where Γ and Γ' give the same type to every x free in e,*

we have

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

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

The proof is straightforward: 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)
```

Several name bindings in a let-expression, the use of a variable pattern in a definition and a definition without a let-binding are considered to be syntactic sugar for, respectively, nested let-expressions, a definition of a lambda-abstraction and a definition of the form `let` *teqww* `=` $\lambda x.\, e$ `in` *teqww*.

The first occurrence of *teq* in line `(1)` above is translated to $teq(\, d_{TEqL}\, v_1\, v_2)$, where *teq*'s translation is the identity function, d_{TEqL} is a dictionary with a single function, say $\mathbf{teq}_L$, that receives the two dictionary arguments v_1 and v_2 passed to *teqww* and yields $\mathbf{teq}_L$, the translation of function *teq* for lists defined above. The translation is given with respect to environment η such that

$\eta(\{\, TEq\ a, Show\ a\}) = \overline{v}$ (where $\overline{v}$ is the sequence $v_1\, v_2$), and $\eta(teq, \tau) = d_{TEqL}$, where $\tau = \texttt{[[}a\texttt{]]} \rightarrow \texttt{[[}a\texttt{]]} \rightarrow (Bool, String)$.

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

$$teq(d_{TEqL}\, d_{TEqInt}\, d_{ShowInt})$$

The use of dictionaries and the ensuing dictionary construction and selection of member values at run-time can be avoided by passing values that correspond to overloaded names that are in fact used. For example, an equality function for lists can receive just an equality function for list elements, instead of a dictionary containing also an unused inequality function. Passing a dictionary to perform selection at run-time is unnecessary. Full laziness and common subexpression elimination are techniques used to avoid repeated construction of dictionaries at run-time [2, 3, 27, 28], but the optimization could be avoided a priori. This and related implementation issues are however outside of the scope of this paper and are left for further work.

Note that the 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 (intended for comparison of the semantics of `(==)` with those of expressions `(==)` x and `(==)` $x\ y$):

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

In a context where `(==)` has type $Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$, the translation of *eqStar* is given by:

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

We have that `(==)` v and *eq dictEqChar* denote a primitive equality function for characters, say *primEqChar*. The translation of each occurrence of `(==)` passes a pertinent dictionary 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.

9. Related Work

Blott [29] and Jones [2] have presented coherent semantics for ow-unambiguous expressions.

Sulzmann et al. [12] consider the encoding of multi-parameter type classes with functional dependencies via constraint handling rules [30]. In their work, as in many other related works (e.g. [31]), ambiguity means ow-ambiguity. Sulzmann et al.'s definition of ow-ambiguity is based on provability of constraints in a program theory, using constraint-handling rules (instead of being a definition that the set of type variables of a constraint set is not a subset of the set of induced functional dependencies of a simple type).

Functional dependencies, introduced in Haskell in order to allow the inference of more specific types and to avoid ambiguity errors, also allow computations at the type level, because of reductions forced by functional dependencies on the type inference algorithm. Type level programming based on functional dependencies has been explored for example in [32, 33] and has been used for instance to define heterogeneous collections and database access libraries for Haskell [34, 35]. The use of delayed-closure ambiguity eliminates the need of functional dependencies to avoid ambiguity but does not allow type level programming, which relies on the fact that type specialization occurs in types that involve reachable type variables.

The delayed-closure approach described in this paper can be seen as a vari-

ation of Agda's approach to overloading [36]. Agda uses a context-independent approach where any use of an overloaded name requires overloading to be resolved. There is no support for constrained types, that allow overloading resolution to be deferred. Overloading works though as in the delayed-closure approach, being based on verifying whether there exists or not a *a single definition of a value in scope at the call site* [36] that is of the type of a so-called *instance argument*. If no such unique definition exists, a type error is reported.

An instance argument is similar to an implicit argument in Agda, but, instead of just requiring a dummy value to be inserted for an implicit argument, it is required that there exists a unique definition of the type of the instance argument in the current scope.

The delayed-closure approach does not have (and could be used to avoid) Agda's restriction to non-recursive resolution for instance arguments, in cases where overloading resolution requires a recursive search for verification of uniqueness of satisfiability. This can be done without the introduction of constrained types. The addition of constrained types in Agda requires further investigation.

10. Conclusion

This paper discusses the problem of ambiguity in Haskell-like languages. A definition of ambiguity, called *delayed-closure*, is presented, where the existence of more than one instance (and more than one type derivation) for the same type of an expression is considered only when there exist unreachable variables in the constraints on the type of an expression. The presence of unreachable variables in constraints characterizes the nonexistence of a program context in which the expression could be placed that would allow instantiation of these variables and overloading resolution.

The paper describes an approach for using default declarations for avoiding ambiguity by the addition of new instance declarations, but leaves for further work a proposal for allowing the importation and exportation of type class instances.

Adopting delayed-closure ambiguity in Haskell would eliminate the need of using functional dependencies or type families for the purpose of dealing with ambiguity. It would also enable Haskell compilers to provide more helpful ambiguity-related error messages. There would be no influence on well-typed Haskell programs, but programs which currently cause ambiguity errors in Haskell could then become well-typed.

The paper presents a type system and a type inference algorithm that includes a constraint-set satisfiability function, that determines whether a given set of constraints is entailed or not in a given context, focusing on issues related to decidability, a constraint-set improvement function, for filtering out constraints for which overloading has been resolved, and a context-reduction function, for reducing constraint sets according to matching instances. A standard dictionary-style semantics for core Haskell is also presented.

As future work, we intend to investigate also the use of delayed-closure ambiguity in connection with type families.

[1] John Mitchell, Foundations of Programming Languages, MIT Press, 1996.

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

[3] K.-F. Faxén, A static semantics for Haskell, Journal of Functional Programming 12 (5) (2002) 295–357.

[4] D. Vytiniotis, S. P. Jones, T. Schrijvers, M. Sulzmann, OutsideIn(X): Modular Type Inference with Local Assumptions, Journal of Functional Programming 21 (4–5) (2011) 333–412.

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

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

[7] Philip Wadler and Stephen Blott, How to make *ad-hoc* polymorphism less *ad hoc*, in: Proc. of ACM POPL'89, 1989, pp. 60–76.

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

[9] Gregory Duck and Simon P. Jones and Peter Stuckey and Martin Sulzmann, Sound and decidable type inference for functional dependencies, in: Proc. of ESOP'04, 2004, pp. 49–63, Springer-Verlag LNCS 2986.

[10] M. Jones, I. Diatchki, Language and Program Design for Functional Dependencies, in: ACM SIGPLAN Haskell Workshop, 2008, pp. 87–98.

[11] P. Stuckey, M. Sulzmann, A Theory of Overloading, ACM TOPLAS 27 (6) (2005) 1216–1269.

[12] M. Sulzmann, G. Duck, S. P. Jones, P. Stuckey, Understanding functional dependencies via constraint handling rules, Journal of Functional Programming 17 (1) (2007) 83–129.

[13] M. Chakravarty, G. Keller, S. P. Jones, S. Marlow, Associated types with class, ACM SIGPLAN Notices 40 (1) (2005) 1–13.

[14] M. Chakravarty, G. Keller, S. P. Jones, Associated type synonyms, ACM SIGPLAN Notices 40 (9) (2005) 241–253.

[15] Mark Jones, Simplifying and Improving Qualified Types, in: Proc. ACM Conf. FPCA'95, 1995, pp. 160–169.

[16] W. Kahl and J. Scheffczyk, Named Instances for Haskell Type Classes, in: Proc. of the 2001 ACM SIGPLAN Haskell Workshop, 2001, pp. 71–100.

[17] Derek Dreyer and Robert Harper and Manuel Chakravarty and Gabriele Keller, Modular Type Classes, ACM SIGPLAN Notices 42 (1) (2007) 63–70.

[18] Marco Silva and Carlos Camarão, Controlling the Scope of Instances in Haskell, in: Proc. of SBLP'2011, 2011, pp. 29–30.

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

[20] G. Smith, Polymorphic type inference for languages with overloading and subtyping, Ph.D. thesis, Cornell Univ. (1991).

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

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

[23] Luís Damas and Robin Milner, Principal type schemes for functional programs, in: Proc. of ACM POPL'82, 1982, pp. 207–212.

[24] P. Kanellakis, H. Mairson, J. Mitchell, Unification and ML type reconstruction, in: Computational Logic: Essays in Honor of Alan Robinson, 1991, pp. 444–478.

[25] John Mitchell and Robert Harper, On the Type Structure of Standard ML, ACM TOPLAS 15 (2) (1993) 211–252.

[26] A. Kfoury, J. Tiuryn, P. Urzyczyn, An Analysis of ML Typability, Journal of the ACM 41 (2) (1994) 368–398.

[27] J. Peterson, M. Jones, Implementing type classes, Proc. of ACM PLDI'93 (1993) 227–236, ACM SIGPLAN Notices 28(6).

[28] M. Jones, Dictionary-free Overloading by Partial Evaluation, LISP and Symbolic Computation 8 (3) (1995) 229–248.

[29] Stephen Blott, An Approach to Overloading with Polymorphism, Ph.D. thesis, Dep. of Computing Science, Univ. of Glasgow (1992).

[30] Thom Früwirth, Theory and Practice of Constraint Handling Rules, The Journal of Logic and Algebraic Programming 37 (1998) 95–138.

[31] K. Chen, P. Hudak, M. Odersky, Parametric Type Classes, in: Proc. of the ACM Conf. on Lisp and Functional Programming, 1992, pp. 170–181.

[32] T. Hallgren, Fun with functional dependencies, in: Proc. of the Joint CS/CE Winter Meeting, 2001, pp. 135–145.

[33] Conor McBride, Faking it: Simulating dependent types in Haskell, Journal of Functional Programmming 12 (4/5) (2002) 375–392.

[34] Oleag Kiselyov and Ralf Lämmel and Keean Schupke, Strongly Typed Heterogeneous Collections, in: Proc. of the 2004 ACM SIGPLAN Haskell Workshop, 2004, pp. 96–107.

[35] A. Silva, J. Visser, Strong Types for Relational Databases, in: Proc. of the 2006 ACM SIGPLAN Haskell Workshop, Haskell '06, 2006, pp. 25–36.

[36] D. Devriese, F. Piessens, On the Bright Side of Type Classes: Instance Arguments in Agda, ACM SIGPLAN Notices 46 (9) (2011) 143–155.