A Solution to Haskell's Multi-Parameter Type Class Dilemma

Carlos Camarão[1], **Rodrigo Ribeiro**[1], **Lucília Figueiredo**[2], **Cristiano Vasconcellos**[3]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

[2]Departamento de Computação – Universidade Federal de Ouro Preto (UFOP)

[3]Universidade do Estado de Santa Catarina (UDESC)

{camarao,rribeiro,lucilia}@dcc.ufmg.br, damiani@joinville.udesc.br

Abstract. *The introduction of multi-parameter type classes in Haskell has been hindered because of problems associated to ambiguity, which occur due to the lack of type specialization during type inference. This paper proposes a minimalist, simple solution to this problem, which requires only a small change to the type inference algorithm and to what has been considered ambiguity in Haskell. It does not depend on the use of programmer specified functional dependencies between type class parameters nor any other extra mechanism, such as associated types. A type system and a type inference algorithm, sound and complete with respect to the type system, are presented.*

1. Introduction

The introduction of multi-parameter type classes (MPTCs) in Haskell has been hindered because of problems associated to ambiguity, which occur due to the lack of type specialization during type inference. This paper proposes a minimalist and simple solution to this problem, which does not require the use of functional dependencies between type class parameters[M. Jones 2000] nor any other extra mechanism, such as associated types [M. Chakravarty and others 2005b, M. Chakravarty and others 2005a]. No change to the language syntax is needed, just a small change to the type inference algorithm and to what has been considered ambiguity in Haskell.

Our proposal is based on the use of a so-called constraint set closure operation (already employed nowadays by Haskell compilers with support for MPTCs), in order to define an overloading resolution trigger condition, as part of type specialization (also called "type improvement") during type inference.

Section 2 describes the dilemma Haskell designers are facing nowadays, with respect to the introduction of MPTCs to the language, and includes an informal presentation of our proposal. Section 3 formalizes our proposal, by defining a criterion for well-formedness of types, including a syntatic characterization of type ambiguity. Sections 4 and 5 present respectively a type system and a type inference algorithm that is sound and complete with respect to the type system. Section 6 concludes.

2. Haskell's MPTC Dilemma

Type classes were introduced in the first version of Haskell, based on work by Wadler and Blott [P. Wadler and S. Blott 1989, S. Blott 1991]. The original motivation was to allow the definition and use of overloaded symbols, in particular equality and numeric operators. This was later extended, by allowing type class parameters to be type constructor variables, in addition to type variables[M. Jones 1993, M. Jones 1995a]. This extension allowed type classes such as, for example:

```
class Functor f where
  fmap::   (a → b) → f a → f b

class Monad m where
  return::   a → m a
   (>>=)::   m a → (a → m b) → m b
```

A further extension, still not incorporated in standard Haskell, allows definitions of type classes to have more than one parameter. MPTCs were recognized as a natural extension to Haskell's single parameter type classes, early in the original paper by Wadler and Blott[P. Wadler and S. Blott 1989] and subsequently by many others (cf. e.g. [M. Jones 1992, K. Chen and others 1992]). However, the use of overloaded symbols introduced in MPTCs was at first thought to introduce ambiguous types[1].

Usually, an expression e is considered semantically *ambiguous* if two distinct denotations can be obtained for it, using a semantics defined inductively on the derivation of a type for e[J. Mitchell 1996]; in other words, e is considered ambiguous if there exist two or more typing derivations that give the same type and distinct denotations for e.

EXAMPLE 1.

```
class F a b where
   f::   a → b
class O a where
   o::   a
h = f o
```

The type of h in 1 should be $(F\,a\,b,\ O\,a) \Rightarrow\ b$. The definition of h is rejected in Hugs[M. Jones and others 1998], giving rise to the error message "Unresolved top level overloading" (for binding h, with outstanding context $(F\ a\ b,\ O\ a)$). GHC[S. P. Jones and others 1998] accepts the definition, if run with compilation options

[1]Cf. e.g.[M. Jones 1995b]:

> "...On the surface, these definitions seem quite reasonable, but we soon run into difficulty if we try to use them. One of the first problems is that the type of the *empty* value is $\forall a.\,\forall c.\,(c \in \mathit{Collect}(a)) \Rightarrow c$, which is *ambiguous* in the sense that a type variable a appears on the left of the $\Rightarrow$ symbol, but is not mentioned on the right. As a result, there is no general way to determine the intended value of type a from the context in which *empty* is used. In general, it is not possible to use any term with an ambiguous principal type if we hope to provide a well-defined semantics for the language[M. Jones 1994]."

And [D. Duggan and J. Ophel 2002]:

> "The issue addressed by this article, that has hindered the usefulness of MPTCs, is the increased potential they introduce for ambiguous typing. "Ambiguity" refers to the situation where there is a free type variable in the overload constraints accumulated during type inference, with no occurrence of that type variable in the inferred type, and therefore no possibility (apparently) that that type variable can be resolved further."

that turn off the monomorphism restriction and allow multi-parameter type classes.[2]

The rule used in Haskell 98 to characterize well-formedness (in particular, unambiguity) of constrained types, with type class constraints and a single parameter, based mainly on the work of Mark Jones[M. Jones 1994], gives the following syntactic criterion for detecting ambiguity: a type $\forall\overline{\alpha}.\, \kappa \Rightarrow \tau$, where $\overline{\alpha} = tv(\kappa \Rightarrow \tau)$, is ambiguous if $tv(\kappa) \not\subseteq tv(\tau)$. That is, a type is ambiguous if there exists at least one type variable in the constraint set κ that does not occur in the simple (i.e. non-quantified and unconstrained) type τ. In Haskell, κ, called a "context", is a set of elements of the form $C\ \mu_1\ \ldots\ \mu_n$, where C is a class name and $\mu_1, \ldots, \mu_n$ are simple types or type constructors, one for each of the n parameters of class C. Naturally, $tv(\sigma)$ denotes the set of free type variables occurring in type σ, and similarly for constraints and sets thereof.

When type classes of more than one parameter are allowed, the above syntactic characterization of ambiguity is not appropriate. The reason for this is explained next.

GHC, since at least version 6.8, makes significant progress towards a characterization of well-formedness of constrained types in the context of MPTCs; GHC 6.10.3 User's Guide says (section 7.8.1.1):

> "GHC imposes the following restrictions on the constraints in a type signature. Consider the type: `forall tv1..tvn (c1, ...,cn) => type`. ...Each universally quantified type variable tv_i must be reachable from `type`. A type variable a is "reachable" if it appears in the same constraint as either a type variable free in the type, or another reachable type variable."

We propose that this *"reachability condition"* — formalized in subsection 3.3 — be used not to indicate well-formedness of constrained types (i.e. to indicate whether types with constraints are valid), but as a trigger condition for overloading resolution (which is dependent on a test of satisfiability in the current context). When, after triggered by this condition, overloading is found not to be resolved, we have ambiguity; otherwise, overloading is resolved — which means that there is a unique substitution that can be used to specialize the constraint set to one that does not contain "unreachable" type variables — and we do not have ambiguity (cf. subsection 3.6).

Nowadays, *functional dependencies* (FDs) are used often in Haskell extended with MPTCs to constrain parameters of type classes in order to avoid what is considered to be ambiguity. A FD lets the programmer state that one of the parameters of a MPTC must be determined from one or more of the other parameters (see e.g. [M. Jones 2000]). Consider, however, the following example (a modified version of an example from `www.haskell.org/haskellwiki/Functional_dependencies`), where a class *Mult* is defined to support the definition of multiplication functions over different types.

[2]
```
f    = show
g    = \x → show x
h :: (Show a) ⇒ a → String
h    = show
main = putStrLn (⊕ '1', ⊕ True)
```
Haskell's monomorphism restriction specifies that *f* cannot, whereas *g* and *h* can, be used in the place of $\oplus$ (the monomorphism restriction forbids *f* to have a polymorphic, constrained type).

EXAMPLE 2.

```
data Vector = Vector Int Int deriving (Eq, Show)
data Matrix = Matrix Vector Vector deriving (Eq, Show)

class Mult a b c where
  (*):: a → b → c
instance Mult Matrix Vector Matrix where
  (*) = ...
instance Mult Matrix Vector Vector where
  (*) = ...

-- code here declaring m1 of type Matrix,
-- and m2, m3 of type Vector
m = (m1 * m2) * m3
```

Hugs disallows m's definition, generating, as in Example 1, type error *Unresolved top-level overloading*. GHC gives the following type to m (as in Example 1, if compilation options that turn off the monomorphism restriction and allow MPTCs are used):

$$m :: \forall c_0, c_1. (\textit{Mult Matrix Vector } c_0, \textit{Mult } c_0 \textit{ Vector } c_1) \Rightarrow c_1$$

However, m cannot be used effectively: if we annotate type `Matrix` for m, a type error is reported. This occurs because the type inferred for m includes constraints *Mult Matrix Vector* c_0, *Mult* c_0 *Vector Matrix*, where c_0 appears only in the constraint set, and thus this type is considered ambiguous. Analogously, if m is annotated with type *Vector*, the type is also considered ambiguous.

In Example 2, there is no means of specializing type variable c_0 occurring in the type of m to *Matrix*. No FD may be used in order to achieve such specialization, because we do not have here a FD: a and b do not "determine" c: for a, b = *Matrix*,*Vector*, we can have c equal to either *Matrix* or *Vector*. This is an instance of a general problem that occurs when there exist non-FDs between type class parameters.

We refer to the above as *the MPTC ambiguity problem*.

The dilemma faced by Haskell designers is that MPTCs should certainly be introduced in Haskell but a solution to problems related to ambiguity, that arise due to lack of type specialization, are thought to require FDs or another similar mechanism, such as associated types[M. Chakravarty and others 2005b]. Yet, despite the added complexity introduced in the language, they do not completely solve the problem, since there are cases where there exist non-FDs between type class parameters (as shown in Example 2).

In our proposal, a constrained type $k \Rightarrow \tau$ is considered well-formed (in particular, non-ambiguous) if either all type variables that occur in κ are "reachable" from τ or, if not, there is a single substitution S that may be applied to κ so that it becomes instantiated to a constraint set having only reachable variables and whose constraints correspond to the instance declarations available in the current context. According to this rule, type $m :: \forall c_0. (\textit{Mult Matrix Vector } c_0, \textit{Mult } c_0 \textit{ Vector Matrix}) \Rightarrow \textit{Matrix}$ would not be considered ambiguous, in the context of class and instance declarations of Example 2, whereas type $m :: \forall c_0. (\textit{Mult Matrix Vector } c_0, \textit{Mult } c_0 \textit{ Vector Matrix}) \Rightarrow \textit{Vector}$ would be considered ambiguous in this context. Note also that FDs is a means of closing the world, and thus may lead to more satisfiability checking than in our proposal.

This establishes lack of reachability not as a syntactic ambiguity condition but as a satisfiabilty testing trigger condition. In other words, lack of reachability "closes the world", as does the use of a FD, in cases where a FD can be established between type class parameters. Closing the world means: perform a constraint set satisfiability test in order to verify whether there exists a substitution that can be used to instantiated type variable(s) occurring in constraints so as to resolve the overloading, with respect to definitions that exist in the current typing context; if there exists only one such substitution, perform the instantiation; if there exists none, report unsatisfiability, otherwise report ambiguity.

An important point, not addressed in this paper, is related to the need to introduce restrictions in the definition of type classes and types of overloaded symbols in order to guarantee decidability of type inference. For these, the reader is referred to e.g. [G. Duck and others 2004, M. Sulzmann and others 2007]. Our proposal does not introduce any extra complexity in this respect, when compared to FDs or an extra mechanism introduced in the language for coping with the MPTC ambiguity problem.

3. Proposal

To formally present our proposal, we introduce first some notational conventions and definitions (subsection 3.1). Well-formed programs and typing contexts are defined in subsection 3.2. Subsections 3.3 and 3.4 define, respectively, the notions of constraint set closure and constraint set satisfiability, which are used for the definition of well-formed constrained types in subsection3.5. Our proposal to solve the MPTC ambiguity problem is presented in subsection 3.6.

3.1. Preliminaries

We use a context-free syntax of type expressions with kinds and of possibly constrained types, as presented in Figure 1, where meta-variable usage is also indicated.

Type expressions are needed, not simply types, because type expressions with higher kinds are not types (for instance, c in type $c\ e$). We slightly abuse notation, in Figure 1, writing $\mu_1^{\imath\to\imath'}\ \mu_2^{\imath}$ to express that type expressions of the form $\mu_1\, \mu_2$ must be such that, letting $\imath$ be the kind of μ_2, the kind of μ_1 is of the form $\imath \to \imath'$ (for some kind $\imath'$).

Kind	$\imath$	$::= \star \mid \imath \to \imath'$
Simple type expression	μ	$::= \alpha^{\imath} \mid T^{\imath} \mid \mu_1^{\imath\to\imath'}\ \mu_2^{\imath}$
Simple type	τ	$\equiv \mu^{\star}$
Constraint	δ	$::= C\,\overline{\mu}$
Type	σ	$::= \tau \mid \kappa \Rightarrow \tau \mid \forall\alpha.\,\sigma$

Class name	C	Type constructor	T
Type expression variable	α, β	Constraint set	κ

Figure 1: Context-free syntax of types and their kinds, and meta-variable usage

For simplicity and following common practice we call α and β just type variables, instead of "type expression variables" (or "type or constructor variables"). To avoid clutter, we usually do not write kinds of type expressions. However, we are careful in distiguishing between the use of letters μ — which represent simple type expressions — and τ — which indicate simple types.

We use $\overline{x}$ as an abbreviation for any sequence of elements in the set $\{x_1, \dots, x_n\}$, for some $n \geq 0$; thus $\forall\,\overline{\alpha}.\,\sigma = \forall\alpha_1.\ \dots\ \forall\alpha_n.\,\sigma$. We let $(\forall\,\overline{\alpha}.\,\emptyset \Rightarrow \tau) = \forall\,\overline{\alpha}.\,\tau$.

A substitution S is a kind-preserving function from type variables to simple type expressions. The identity substitution is denoted by id and $dom(S)$ is defined by $dom(S) = \{\alpha \mid S\alpha \neq \alpha\}$. $S\sigma$ represents the capture-free operation of substituting $S\alpha$ for each free occurrence of type variable α in σ. For simplicity, $S\sigma$ is sometimes written as $[\alpha_j := \tau_j]^{j=1..n}\sigma$, where $dom(S) = \{\alpha_j\}^{j=1..n}$ and $S\alpha_j = \tau_j$, for $j = 1, \dots, n$.

3.2. Programs and Typing Contexts

The actual form of programs is not of much interest for us here. The important point is just that they contain global class and instance declarations that introduce class and instance constraints in a global typing context Γ of a given outermost expression.

A class declaration

$$\texttt{class}\quad \kappa \Rightarrow C\,\overline{\alpha}\quad \texttt{where}\quad \{\,x_1 :: \kappa_1 \Rightarrow \tau_1;\ \dots\ ; x_n :: \kappa_n \Rightarrow \tau_n\,\}$$

introduces in Γ a *class-constraint* $\forall\overline{\alpha}.\,\kappa \Rightarrow C\,\overline{\alpha}$, where C is a class name, $\overline{\alpha}$ is a sequence of class parameters and κ is called a class context. There are no restrictions on a class context, except that the class hierarchy must be acyclic. For each such class declaration, we let $\Gamma^{\texttt{cls}}(C) = \forall\overline{\alpha}.\kappa \Rightarrow C\,\overline{\alpha}$.

The above class declaration also introduces in Γ type assumptions $x_1 : \sigma_1, \dots ; x_n : \sigma_n$, where $\sigma_i = \forall\overline{\alpha_i}\,.\kappa \cup \kappa_i \Rightarrow \tau_i$ and $\overline{\alpha_i} = tv(\kappa \cup \kappa_i \Rightarrow \tau_i)$, for $i = 1, \dots, n$. In this case, we let $\Gamma^{\texttt{cls}}(x_i) = \sigma_i$ ($\Gamma^{\texttt{cls}}(x)$ is undefined if x is not an overloaded symbol).

Each instance declaration

$$\texttt{instance}\quad \kappa \Rightarrow C\,\overline{\mu}\quad \texttt{where}\,\{\,x_1 = e_1; \dots ; x_n = e_n\,\}$$

introduces in Γ an *instance-constraint* $\forall\overline{\alpha}.\,\kappa \Rightarrow C\,\overline{\mu}$, where $\overline{\alpha} = tv(\kappa \Rightarrow C\,\overline{\mu}) - tv(\Gamma)$, provided that it is well-formed in Γ. We let $\Gamma^{\texttt{ins}}(C)$ denote the set of instance-constraints introduced in Γ by instance declarations of type class C.

Well-formedness of an instance-constraint $\kappa \Rightarrow C\,\overline{\mu}$ in a typing context Γ, written $\Gamma \models^{\texttt{wfi}} \kappa \Rightarrow C\,\overline{\mu}$, is defined in Figure 3.

Informaly, an instance-constraint $\forall\,\overline{\alpha}.\ \kappa' \Rightarrow C\,\overline{\mu}$ is well-formed in Γ if the following conditions hold: 1) $tv(\kappa') \subseteq tv(C\,\overline{\mu})$; 2) the *instance head* $C\,\overline{\mu}$ matches with a

$$\boxed{\Gamma \models^{\texttt{cls}} \delta\,[S, \kappa]}$$

$$\frac{\Gamma^{\texttt{cls}}(C) = \kappa \Rightarrow C\overline{\alpha} \qquad S(C\overline{\alpha}) = C\overline{\mu}}{\Gamma \models^{\texttt{cls}} C\overline{\mu}\,[S, \kappa]}$$

Figure 2: Class-instance matching

$$\boxed{\Gamma \models^{\texttt{wfi}} \kappa \Rightarrow C\,\overline{\mu}}$$

$$\frac{\begin{array}{l} tv(\kappa) \subseteq tv(C\,\overline{\mu}) \\ \Gamma \models^{\texttt{cls}} C\,\overline{\mu}\,[S_C, \kappa_C] \\ S_C\,\kappa_C - \kappa_0 \subseteq \kappa \ \text{ where } \ \kappa_0 = \{\delta \in S_C\,\kappa_C \mid tv(\delta) = \emptyset\} \\ \text{for each } C'\,\overline{\mu}' \in \kappa_0, C'\,\overline{\mu}' \in \Gamma^{\texttt{ins}}(C') \end{array}}{\Gamma \models^{\texttt{wfi}} \kappa \Rightarrow C\,\overline{\mu}}$$

Figure 3: Well-formed instance constraint

corresponding class-constraint, that is $\Gamma \models^{\texttt{cls}} C\,\overline{\mu}\,[S_C, \kappa_C]$ is provable for some $[S_C, \kappa_C]$, according to the rules defined in Figure 2 (in this case, we call S_C the *instance matching substitution* for $C\,\overline{\mu}$); 3) the *instance context* κ is well-formed — that is, each constraint $C'\overline{\mu}' \in S_C\,\kappa_C$ must occur in κ, except if $C'\overline{\mu}'$ does not contain any type variables, in which case there must exist a corresponding instance declaration for class C'.

In order to clarify this definition, let us consider the following class and instance declarations:

```
class $A\,\alpha$ where ...
class $B\,\alpha\,\beta$ where ...
class $C\,\alpha$ where ...
class $\{A\,\alpha, B\,\alpha\,\beta\} \Rightarrow D\,\alpha\,\beta\,\gamma$ where ...
instance $A$ Int where ...
instance $\{B$ Int $\beta,\ C\,\gamma\} \Rightarrow D$ Int $\beta\,\gamma$ where ...
```

The instance declaration for class D above is well-formed in a typing context Γ including class and instance constraints introduced by the above declarations, since we have that: 1) $tv(\{B\ \texttt{Int}\,\beta,\ C\,\gamma\}) \subseteq tv(D\ \texttt{Int}\,\beta\,\gamma)$; 2) $\Gamma \models^{\texttt{cls}} D\,\text{Int}\,\beta\,\gamma\,[S_D, \kappa_D]$, where $S_D = [\alpha := \text{Int}]$ and $\kappa_D = \{A\ \alpha,\ B\ \alpha\ \beta\}$; 3) the context for this instance declaration includes constraint $B\,\text{Int}\,\beta$ (corresponding to $B\,\alpha\,\beta \in \kappa_D$), but does not include constraint $A\ \texttt{Int}$ (corresponding to $A\ \alpha \in \kappa_D$), which does not contain type variables, and has a corresponding instance declaration. Note that the context of an instance declaration may also include constraints that do not occur in its class declaration, as is the case for constraint $C\,\gamma$ in the above instance declaration for class D.

Further restrictions must be imposed in order to ensure that *context reduction* terminates (we refer the reader to [G. Duck and others 2004, M. Sulzmann and others 2007] and to section 7.6 of GHC user's manual[S. P. Jones and others 1998] for discussions on this topic).

Instance-constraints in $\Gamma^{\texttt{ins}}(C)$ must not unify with each other, i.e. two distinct instance-constraints $\kappa \Rightarrow (C\ \tau_1 \ldots \tau_n)$ and $\kappa' \Rightarrow (C\ \tau'_1 \ldots \tau'_n)$ must not be such that $S\tau_1 = S\tau'_1, \ldots, S\tau_n = S\tau'_n$, for some substitution S (again, assuming that $tv(\{\tau_1, \ldots, \tau_n\}) \cap tv(\{\tau'_1, \ldots, \tau'_n\}) = \emptyset$). This requirement — of *non-overlapping instances* — could be relaxed, as in Haskell, but this is outside of the scope of this paper.

The above instance declaration also introduces in Γ instance type assumptions $x_i : \sigma_i$, for $i = 1, \ldots, n$, where $\sigma_i = \forall\,\overline{\alpha}'_i.\,S_C(\kappa_i \cup \kappa \Rightarrow \tau_i)$, $\Gamma^{\texttt{cls}}(x_i) = \forall\overline{\alpha}.\kappa_i \Rightarrow \tau_i$,

$\overline{\alpha}' = tv(S_C(\kappa_i \cup \kappa \Rightarrow \tau_i)) - tv(\Gamma)$ and S_C is the instance matching substitution of constraint $C\,\overline{\mu}$ in Γ.

3.3. Constraint set closure

Section 3.5 defines well-formedness of a constrained type $\forall\overline{\alpha}.\,\kappa \Rightarrow \tau$ based on a (so-called) *constrant-set closure operation*, $\kappa|^*_V$, defined as follows (constraint set closure was previously defined in [C. Camarão and L. Figueiredo 1999, C. Camarão and others 2008]):

$$\kappa|_V = \{C\,\overline{\mu} \in \kappa \mid tv(\overline{\mu}) \cap V \neq \emptyset\}$$
$$\kappa|^*_V = \begin{cases} \kappa|_V & \text{if } tv(\kappa|_V) \subseteq V \\ \kappa|^*_{tv(\kappa|_V)} & \text{otherwise} \end{cases}$$

We have e.g. $tv\big(\{\textit{Collection}\ c\ e\}|^*_{\{e\}}\big) = \{c, e\}$ and $tv\big(\{F\ a\ b, G\ a\ c\}|^*_{\{c\}}\big) = \{a, b, c\}$.

3.4. Constraint Set Satisfiability

We write $\Gamma \models^{\mathtt{sat}} \kappa\,[S]$ to mean that the constraint set κ is satisfiable in typing context Γ by substitution S, according to the rules presented in Figure 4. If $\Gamma \models^{\mathtt{sat}} \kappa\,[S]$ is provable, for some substitution S, we call S a *solution* to the satisfiability problem (Γ, κ).

$$\boxed{\Gamma \models^{\mathtt{sat}} \kappa\,[S]}$$

$$\frac{}{\Gamma \models^{\mathtt{sat}} \emptyset\,[id]} \qquad \frac{\begin{array}{l}\kappa \Rightarrow C\,\overline{\mu}' \in \Gamma^{\mathtt{ins}}(C) \\ S(C\,\overline{\mu}) = S(C\,\overline{\mu}') \\ \Gamma \models^{\mathtt{sat}} S\kappa\,[S'] \\ S_0 = (S' \circ S)\,|_{tv(C\,\overline{\mu})}\end{array}}{\Gamma \models^{\mathtt{sat}} \{C\,\overline{\mu}\}\,[S_0]} \qquad \frac{\Gamma \models^{\mathtt{sat}} \kappa_1\,[S_1] \quad \Gamma \models^{\mathtt{sat}} S_1\kappa_2\,[S_2]}{\Gamma \models^{\mathtt{sat}} \kappa_1 \cup \kappa_2\,[S_2 \circ S_1]}$$

Figure 4: Constraint set satisfiability

3.5. Well-formedness, simplification and equality of types

DEFINITION 1 (Well-formed types). A constrained type $\kappa \Rightarrow \tau$ is well formed in a typing context Γ, written $\Gamma \models \kappa \Rightarrow \tau$, if $\Gamma \models^{\mathtt{wf}} \kappa \Rightarrow \tau\,[S]$ is provable for some S, according to the rule presented in Figure 5.

$$\boxed{\Gamma \models^{\mathtt{wf}} \kappa \Rightarrow \tau\,[S]}$$

$$\frac{\begin{array}{l}\text{there exists a } \textit{unique} \text{ substitution } S \text{ such that } \Gamma \models^{\mathtt{sat}} \kappa_0\,[S] \text{ and } tv(S\kappa_0) = \emptyset \\ \text{where } \kappa_0 = (\kappa - \kappa|^*_V) \cup \{\delta \in \kappa \mid tv(\delta) = \emptyset\} \\ \qquad V = tv(\tau) - tv(\Gamma)\end{array}}{\Gamma \models^{\mathtt{wf}} \kappa \Rightarrow \tau\,[S]}$$

Figure 5: Well formed constrained type

Informally, a constrained type $\kappa \Rightarrow \tau$ is well formed in a typing context Γ if the set of all constraints in κ that have no type variables or have only "unreacheable type variables" is uniquely satisfiable in Γ.

Note that, if $\Gamma \models^{\texttt{wf}} \kappa \Rightarrow \tau\,[S]$ is provable for some S, where typing context Γ have only well-formed class and instance constraints (according to the rules given in Section 3.2), then $S\tau = \tau$ and $S\kappa$ does not contain "unreachable type variables", i.e. $S\kappa = (S\kappa)|^*_V$ where $V = tv(\tau) - tv(\Gamma)$. In this case, type $\kappa \Rightarrow \tau$ can be specialized to $S\kappa \Rightarrow \tau$, that can be simplified to $\kappa' \Rightarrow \tau$, where $\kappa' = S\kappa - \{\delta \in S\kappa | tv(\delta) = \emptyset\}$ — this simplification is denoted by $\Gamma \models \kappa \Rightarrow \tau \gg \kappa' \Rightarrow \tau$. We also write $\Gamma \models \sigma \gg \sigma'$, where $\sigma = \forall\overline{\alpha}.\kappa \Rightarrow \tau$ and $\sigma' = \forall\overline{\alpha}'.\kappa' \Rightarrow \tau$, if $\Gamma \models \kappa \Rightarrow \tau \gg \kappa' \Rightarrow \tau$ and $\overline{\alpha}' = tv(\kappa' \Rightarrow \tau) \cap \overline{\alpha}$.[3]

Type simplification $\Gamma \models \sigma \gg \sigma'$ yields equal types in Γ, that is, equality between constrained types, written $\Gamma \models \sigma \equiv \sigma'$, is defined as the reflexive, symmetric and transitive closure of the type simplification relation $\Gamma \models \sigma \gg \sigma'$.

3.6. Solution to the MPTC ambiguity problem

According to our definition of well-formed constrained type above, the type inferred for $m :: Matrix$ in Example 2, namely

$$(\mathit{Mult}\ \mathit{Matrix}\ \mathit{Vector}\ c,\ \ \mathit{Mult}\ c\ \mathit{Vector}\ \mathit{Matrix}) \Rightarrow \mathit{Matrix}$$

is well-formed, and can be simplified to *Matrix*. This occurs because there is a unique solution S — namely $S = [c := \mathit{Matrix}]$ — to the satisfiability problem of κ in Γ, where:

$$\begin{array}{lcl} \kappa & = & \{\ \mathit{Mult}\ \mathit{Matrix}\ \mathit{Vector}\ c,\ \ \mathit{Mult}\ c\ \mathit{Vector}\ \mathit{Matrix}\ \} \\ \Gamma^{\texttt{cls}}\,(\mathit{Mult}) & = & \{\,(\star)::(\mathit{Mult}\ a\ b\ c)\ \Rightarrow\ a \rightarrow b \rightarrow c\ \} \\ \Gamma^{\texttt{ins}}\,((\star)) & = & \{\ \mathit{Matrix} \rightarrow \mathit{Vector} \rightarrow \mathit{Matrix},\ \ \mathit{Matrix} \rightarrow \mathit{Vector} \rightarrow \mathit{Vector}\} \end{array}$$

Note that the overall effect of using our definition of well-formed constrained type is to make well-typed some expressions that were previously considered ambiguous.

4. Type System

We use the context-free syntax of core-ML expressions given in Figure 6, called here core-Haskell to emphasize that we want to analyze typability and type inference for these expressions when they occur in an outermost typing context with information about overloaded symbols (cf. Section 3.2).

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

Figure 6: Context-free syntax of core-Haskell expressions

In type systems with support for parametric polymorphism, the type ordering is such that $\forall\alpha.\,\sigma \leq [\alpha := \tau]\sigma$, for *all* simple types τ. In the presence of constrained types, we must take the instantiation relation in the context of a typing context, because overloading is resolved according to constraints which are available in a typing context

[3]Constrained type specialization/simplificatiuon is also called *improvement* [M. Jones 1995b].

$$\boxed{\Gamma \vdash e : \kappa \Rightarrow \tau}$$

$$\frac{x : \sigma \in \Gamma \quad \Gamma \models \sigma \leq \kappa \Rightarrow \tau}{\Gamma \vdash x : \kappa \Rightarrow \tau} \qquad (\texttt{VAR})$$

$$\frac{\Gamma, x : \tau' \vdash e : \kappa \Rightarrow \tau}{\Gamma \vdash \lambda x.\, e :\ \kappa \Rightarrow \tau' \to \tau} \qquad (\texttt{ABS})$$

$$\frac{\Gamma \vdash e : \kappa \Rightarrow \tau' \to \tau \quad \Gamma \vdash e' : \kappa' \Rightarrow \tau'}{\Gamma \vdash e\, e' : \kappa \cup \kappa' \Rightarrow \tau} \qquad (\texttt{APP})$$

$$\frac{\Gamma \vdash e \colon \kappa \Rightarrow \tau \quad \Gamma \models \kappa \Rightarrow \tau \quad \Gamma,\, x\colon\sigma \vdash e' \colon \kappa' \Rightarrow \tau'}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \kappa' \Rightarrow \tau'} \qquad (\texttt{LET})$$

$$\text{where:}\quad \begin{aligned} \sigma &= \forall \overline{\alpha}.\kappa \Rightarrow \tau \\ \overline{\alpha} &= tv(\kappa \Rightarrow \tau) - tv(\Gamma) \end{aligned}$$

Figure 7: Type System

(cf. section 3.2). We define thus an ordering (i.e. a constraint-based instantiation) relation on types, with respect to a given typing context — written $\Gamma \models \sigma \leq \sigma'$ — similar to that in [K. Faxén 2003, C. Camarão and L. Figueiredo 1999], as follows.

DEFINITION 2 (Type Ordering). Let Γ be a typing context and $\forall \alpha.\, \sigma$ be a well formed type in Γ, that is $\Gamma \models \forall \alpha.\, \sigma$. Then $\Gamma \models \forall \alpha.\sigma \leq [\alpha := \tau]\sigma$, for any type τ such that $\Gamma \models [\alpha := \tau]\sigma$.

If $\Gamma \models \sigma \leq \sigma'$, then σ' is called an instance of σ in typing context Γ, and σ is said to be more general than σ' in typing context Γ.

A syntax directed type system for core-Haskell is presented in Figure 7. Type rules are as usual Hindley-Milner rules, except that constrained instatiation is used in rule (`VAR`), and well-formedness of a constrained type is required before its introduction in the typing context, in rule (`LET`). Also, in rule (`APP`), the type of an application $e\, e'$ includes both constraints that occur in the type of function e and in the type of argument e'.

The type system enjoys the important property that derivability of well-formed constrained types is closed under substitution (we assume always that all class-constraints and instance constraints in Γ are well-formed). (cf. [K. Faxén 2003]):

LEMMA 1 (Substitution). *Let* $\Gamma \vdash e : \kappa \Rightarrow \tau$. *Then* $\Gamma \models \kappa \Rightarrow \tau$ *and, for all* S *such that* $\Gamma \models S(\kappa \Rightarrow \tau)$*, we have that* $S\Gamma \vdash e : S(\kappa \Rightarrow \tau)$.

Proof. By induction on the structure of e. ∎

5. Type inference

The type inference algorithm is presented in Figure 5 as a syntax-directed proof system of judgements $\Gamma \vdash_I e : (\kappa \Rightarrow \tau, \Gamma')$.

Let $\Theta(\Gamma)$ denote the information contained in the outermost typing context, i.e. class and instance constraints introduced by class and instance declarations, and the

$$\boxed{\Gamma \vdash_{\text{I}} e : \kappa \Rightarrow \tau}$$

$$\frac{\Gamma(x) = \forall\,\overline{\alpha}.\,\kappa \Rightarrow \tau}{\Gamma \vdash_{\text{I}} x : \big(\kappa \Rightarrow \tau,\ \{x : \Gamma^{*}(x)\}\big)} \quad (\texttt{VAR}_{\text{I}})$$

$$\frac{\Gamma \vdash_{\text{I}} (e : \kappa \Rightarrow \tau, \Gamma')}{\Gamma \ominus x \vdash_{\text{I}} \lambda x.\,e : (\kappa \Rightarrow \tau' \to \tau,\ \Gamma' \ominus x)} \quad (\texttt{ABS}_{\text{I}})$$

$$\text{where: } \tau' = \begin{cases} \tau & \text{if } x : \tau \in \Gamma' \\ \alpha & \text{otherwise, } \alpha \text{ fresh} \end{cases}$$

$$\frac{\Gamma \vdash_{\text{I}} e : (\kappa \Rightarrow \tau, \Gamma_1) \qquad \Gamma \vdash_{\text{I}} e' : (\kappa' \Rightarrow \tau', \Gamma_2)}{\Gamma \vdash_{\text{I}} e\,e' : S'S(\kappa \cup \kappa') \Rightarrow S\alpha,\ S\Gamma_1 \cup S\Gamma_2)} \quad (\texttt{APP}_{\text{I}})$$

$$\begin{aligned} \text{where:}\quad & S = \mathit{unify}(\{\tau = \tau' \to \alpha\} \cup \mathit{st}(\Gamma_1, \Gamma_2)) \\ & S' = \mathit{wf}(S(\kappa \cup \kappa' \Rightarrow \alpha), \Gamma),\ \alpha \text{ fresh} \end{aligned}$$

$$\frac{\Gamma \vdash_{\text{I}} e : (\kappa \Rightarrow \tau, \Gamma_1) \qquad \Gamma, \{x : \sigma\} \vdash_{\text{I}} e' : (\kappa' \Rightarrow \tau', \Gamma_2)}{\Gamma \vdash_{\text{I}} \texttt{let}\ x = e\ \texttt{in}\ e' : (S'S\kappa' \Rightarrow S\tau',\ S\Gamma' \ominus x)} \quad (\texttt{LET}_{\text{I}})$$

$$\begin{aligned} \text{where:}\quad & S_0 = \mathit{wf}(\kappa \Rightarrow \tau, \Gamma_1) \\ & \sigma = \forall\overline{\alpha}.\,S_0\kappa \Rightarrow \tau \\ & \overline{\alpha} = tv(S_0\kappa \Rightarrow \tau) - tv(\Gamma_1) \\ & S = \mathit{unify}(\mathit{st}(\Gamma_1, \Gamma_2)) \\ & S' = \mathit{wf}(S(\kappa \cup \kappa' \Rightarrow \tau'), \Gamma) \end{aligned}$$

Figure 8: Algorithm for Inference of Principal Typings

principal type declared for each overloaded symbol, as specified in subsection 3.2, i.e. :

$$\Theta(\Gamma) = \bigcup_{C} \Gamma^{\texttt{cls}}(C) \cup \bigcup_{C} \Gamma^{\texttt{ins}}(C) \cup \bigcup_{x} \Gamma(x)$$

where C ranges over all class names and x over all overloaded symbols. The algorithm uses the notation $\Gamma(x)$ and $\Gamma^{*}(x)$, defined as follows:

$$\Gamma(x) = \begin{cases} \kappa \Rightarrow \tau & \text{if } x \text{ is an overloaded symbol and } \Gamma^{\texttt{cls}}(x) = \forall\,\overline{\alpha}.\,\kappa \Rightarrow \tau \\ & \text{or } x \text{ is a lambda or let-bound variable and } x : \forall\,\overline{\alpha}.\,\kappa \Rightarrow \tau \in \Gamma \\ & \quad \text{where type variables in } \kappa \Rightarrow \tau \text{ are fresh} \\ \alpha & \text{otherwise, where } \alpha \text{ is fresh} \end{cases}$$

$$\begin{aligned} &\Gamma^{*}(x) = \Gamma(x) \cup \Theta(\Gamma) \\ &\Gamma \ominus x = \Gamma - \{x : \sigma \mid x \text{ is a lambda or let-bound variable}, \sigma \in \Gamma(x)\} \end{aligned}$$

Also used are functions *lb*, *st*, *unify* and *wf*, were *unify* type is the usual type unification function; *lb*(Γ) denotes the subset of type assumptions for lambda-bound variables in Γ and *st*(Γ, Γ') yields the set of equality equations between (simple) types of each

```
wf $(\kappa \Rightarrow \tau, \Gamma)$ =
  let $\kappa_0 = \kappa - \kappa|^*_{tv(\tau)} \cup \{\delta \in \kappa \mid tv(\delta) = \emptyset\}$
   in if $\kappa_0 = \emptyset$ then id
     else case sat $(\kappa_0, \Gamma)$ of
            (Unsat, _)      → "error: unsatisfiability ..."
            (Single, $S$)     → if $tv(S\kappa_0) = \emptyset$ then $S$
                                else "error: ambiguity ..."
            __              → "error: ambiguity ..."
```

Figure 9: Well-formedeness (unambiguity) of constrained types

lambda-bound variable which occurs in both Γ and Γ' (which must be unified during type inference, because they are restricted to have monomorphic types):

$$st(\Gamma, \Gamma') = \{\tau = \tau' \mid x : \tau \in lb(\Gamma),\ x : \tau' \in lb(\Gamma')\}$$

Function *wf* tests well-formedness of a constrained type $\kappa \Rightarrow \tau$ in a typing context Γ, based on the definition of relation $\Gamma \models^{\text{wf}} \kappa \Rightarrow \tau$ given in Figure 5. The definition of *wf* is presented in Figure 9.

It uses function $sat(\kappa, \Gamma)$, defined in [C. Camarão and others 2004, C. Camarão and others 2008], which essentially implements $\Gamma \models^{\text{sat}} \kappa\ [S]$ defined in Figure 4, computing the solution for a satisfiability problem (κ, Γ) whenever one exists. More precisely, sat returns whether there exist zero (*Unsat*), one (*Single*) or more solutions to the satisfiability problem (κ, Γ) given as its argument and, in the case of *Single*, it returns also the solution S. In this last case, if overloading has been resolved, this substitution is used to specialize ("improve") type $\kappa \Rightarrow \tau$, so that the constraints in κ that triggered overloading resolution can be removed. If there is no solution to the satisfiability problem (κ, Γ), *wf* gives an error indicating that there exists no instance-constraint in Γ that corresponds to what is required by the type of the relevant expression; if there are two or more solutions to the satisfiability problem, then *wf* returns an error that indicates ambiguity.

The restrictions that must be imposed on class and instance declarations in order to have decidability of constraint set satisfiability — $\Gamma \models^{\text{sat}} \kappa\ [S]$ — are the same as imposed in Haskell (see e.g. [G. Duck and others 2004, M. Sulzmann and others 2007, P. Stuckey and M. Sulzmann 2005, S. P. Jones and others 1998]).

5.1. Principal Type and Typing

DEFINITION 3 (Principal Type). The principal type of an expression e in a typing context Γ is the least upperbound, in the partial order given by $\Gamma \models \sigma \leq \sigma'$ of Definition 2, of all types *that can be derived* for e in Γ.

Orderings on typing contexts and typings are straightforward extensions of the ordering on types and the notion of principal typing is a straightforward extension of that of principal type, that takes into account these new orderings. We have:

DEFINITION 4 (Ordering on Typing Contexts). Let $\Gamma = \Theta(\Gamma_1) = \Theta(\Gamma_2)$. We define $\Gamma \models \Gamma_1 \leq \Gamma_2$ if, for all lambda or let-bound variables x, we have that $\Gamma \models \Gamma_1(x) \leq \Gamma_2(x)$.

DEFINITION 5 (Ordering on Typings). Let (σ_1, Γ_1) and (σ_2, Γ_2) be typings — i.e. pairs where the first component is a type and the second component is a typing context — for which $\Theta(\Gamma_1) = \Theta(\Gamma_2)$. Let us call Γ the typing context $\Theta(\Gamma_i)$ ($i = 1$ or $i = 2$). Then $\Gamma \models (\sigma_1, \Gamma_1) \leq (\sigma_2, \Gamma_2)$ if $\Gamma \models \Gamma_1 \leq \Gamma_2$ and $\Gamma \models \sigma_1 \leq \sigma_2$.

DEFINITION 6 (Principal Typing). The principal typing of an expression e in an outermost (overloading) typing context Γ_0 is the least upperbound, in the partial order given by $\Gamma_0 \models (\sigma_1, \Gamma_1) \leq (\sigma_2, \Gamma_2)$ of Definition 5, of all typings (σ, Γ) such that $\Gamma_0 = \Theta(\Gamma)$, $\Gamma \vdash e : \kappa \Rightarrow \tau$ is provable and $\sigma = \forall \overline{\alpha}.\, \kappa \Rightarrow \tau$, where $\overline{\alpha} = tv(\kappa \Rightarrow \tau) - tv(\Gamma)$.

THEOREM 1 (Principal Typing). *For any expression* e *and well formed outermost typing context* Γ_0*, we have that* $\Gamma \vdash_I e : \kappa \Rightarrow \tau$ *is provable if and only if* (σ, Γ) *is the principal typing of* e *in* $\Theta(\Gamma)$*, where* $\sigma = \forall \overline{\alpha}.\, \kappa \Rightarrow \tau$*, where* $\overline{\alpha} = tv(\kappa \Rightarrow \tau) - tv(\Gamma)$.

A *principal type* theorem follows directly from Theorem 1 (of *principal typing*).

6. Conclusion

We have presented a simple, minimalist solution to Haskell's MPTC dilemma, which requires only a small change to the type inference algorithm and to what has been considered ambiguity in Haskell. It does not require the use of FDs between type class parameters nor any other extra mechanism, such as associated types.

More precisely, our proposal is based on using a simple reachability condition, already employed nowadays by Haskell compilers with support for MPTCs; but failure of this reachability condition is used to close the world, as opposed to being used directly as a syntactic criterion for type ambiguity. In comparison to the use of FDs between type class parameters or an extra mechanism to be employed in order to close the world by performing type specialization during type inference, our proposal does not require programmer's intervention and can be used when no FDs exist.

References

C. Camarão and L. Figueiredo (1999). Type Inference for Overloading without Restrictions, Declarations or Annotations. In *Proc. 4th Fuji International Symp. on Functional and Logic Programming (FLOPS'99)*, pages 37–52. Springer-Verlag, LNCS 1722.

C. Camarão and others (2004). Constraint-set Satisfiability for Overloading. In *Proc. of the 6th ACM SIGPLAN International Conf. on Principles and Practice of Declarative Programming (PPDP'04)*, pages 67–77.

C. Camarão and others (2008). Open and Closed Worlds for Overloading: a Definition and Support for Coexistence. *Journal of Universal Computer Science*, 13(6):854–873.

D. Duggan and J. Ophel (2002). Type Checking Multi-Parameter Type Classes. *Journal of Functional Programming*, 12(2):135–158.

G. Duck and others (2004). Sound and decidable type inference for functional dependencies. In *Proc. of the European Symposium on Programming (ESOP'04).* Springer-Verlag LNCS 2986.

J. Mitchell (1996). *Foundations of Programming Languages.* MIT Press.

K. Chen and others (1992). Parametric Type Classes. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 170–181.

K. Faxén (2003). Haskell and Principal Types. In *Proc. of the 2003 ACM SIGPLAN Haskell Workshop*, pages 88–97.

M. Chakravarty and others (2005a). Associated type synonyms. In *Proc. of the 10th ACM SIGPLAN International Conf. on Functional Programming (ICFP'05)*, pages 241–253.

M. Chakravarty and others (2005b). Associated types with class. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL'05)*, pages 1–13.

M. Jones (1992). A Theory of Qualified Types. In *Proc. of the European Symposium on Programming (ESOP'92)*, volume 582, pages 287–306. Springer-Verlag.

M. Jones (1993). A system of constructor classes: overloading and higher-order polymorphism. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, pages 52–64.

M. Jones (1994). *Qualified Types: Theory and Practice.* PhD thesis, Distinguished Dissertations in Computer Science. Cambridge University Press.

M. Jones (1995a). A system of constructor classes: overloading and higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–36.

M. Jones (1995b). Simplifying and Improving Qualified Types. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA'95)*, pages 160–169.

M. Jones (2000). Type Classes with Functional Dependencies. In *Proc. of the European Sympoisum on Programming (ESOP'2000).* Springer-Verlag LNCS 1782.

M. Jones and others (1998). Hugs98. http://www.haskell.org/hugs/.

M. Sulzmann and others (2007). Understanding Functional Dependencies via Constraint Handling Rules. *Journal of Functional Programming*, 17(1):83–129.

P. Stuckey and M. Sulzmann (2005). A Theory of Overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1216–1269.

P. Wadler and S. Blott (1989). How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 60–76. ACM Press.

S. Blott (1991). *Type Classes.* PhD thesis, Department of Computer Science, Glasgow University.

S. P. Jones and others (1998). GHC — The Glasgow Haskell Compiler 6.10 User's Manual. http://www.haskell.org/ghc/.