Type Inference for GADTs and Anti-unification

Adelaine Gelain[1], Cristiano Vasconcellos[1](✉),
Carlos Camarão[2], and Rodrigo Ribeiro[3]

[1] DCC, Universidade do Estado de Santa Catarina (UDESC), Joinville, Brazil
adelainegelain@gmail.com, cristiano.vasconcellos@udesc.br
[2] DCC, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil
camarao@dcc.ufmg.br
[3] DECSI, Universidade Federal de Ouro Preto (UFOP), João Monlevade, Brazil
rodrigo@decsi.ufop.br

Abstract. Nowadays the support of generalized algebraic data types (GADTs) in extensions of Haskell allows functions defined over GADTs to be written without the need for type annotations in some cases and requires type annotations in other cases. In this paper we present a type inference algorithm for GADTs that is based on a closed-world approach to overloading and uses anti-unification and constraint-set satisfiability to infer the relationship between the types of function arguments and result. Through some examples, we show how the proposed algorithm allows more functions defined over GADTs to be written without the need for type annotations.

1 Introduction

Generalized Algebraic Data Types (*GADTs*) constitute a powerful extension to algebraic data types of functional languages like Haskell and ML, and are nowadays widely used. A GADT is defined by giving an explicit type signature for each of its constructors. This allows functions to be defined by specifying equations that return expressions of distinct types, all instances of the GADT type. For example, the function *eval*, presented in e.g. [7,12], evaluates an expression and returns a value of a type that varies according to the argument type (due to space reasons the *Term* constructor is presented in a shortened form):

```
data Term a where
     Lit    :: Int → Term Int
     Inc    :: Term Int → Term Int
     IsZ    :: Term Int → Term Bool
     If     :: Term Bool → Term a → Term a → Term a
     Pair   :: Term a → Term b → Term (a,b)
```


A. Pardo and S.D. Swierstra (Eds.): SBLP 2015, LNCS 9325, pp. 16–30, 2015.
DOI: 10.1007/978-3-319-24012-1_2

```
eval :: Term a → a
eval (Lit i)      = i
eval (Inc t)      = 1 + eval t
eval (IsZ i)      = 0 == eval i
eval (If l e1 e2) = if eval l then eval e1 else eval e2
eval (Pair a b)   = (eval a, eval b)
```

The use of an algebraic data type would destroy the simplicity of the evaluator, by requiring a declaration of another algebraic data type with a distinct constructor (tag) for each possible distinct type of the result, with undesirable constructor tagging and untagging.

Type inference with GADTs is complex, mainly because of problems in identifying a principal type in many cases. Consider the following example, taken from [15]:

```
data T a where
  T1 :: Int → T Bool
  T2 :: T a

test (T1 n) _ = n > 0
test T2     r = r
```

In the first alternative of *test*, the result type inferred for the expression $n > 0$, *Bool*, is associated with the type of constructor $T1$, and $(T1\ n)$ can be determined to have type $T\ Bool$, with n of type *Int*. In the second alternative of *test*, there is no explicit association between type $T\ a$, constructed by the use of $T2$, and the return type (the type of r), and thus in this case the type of the result should be unified with that of the first alternative (*Bool*). A relation between the GADT type and the type of the result could exist, and be explicitly annotated: the following type signatures are both accepted for the function *test*, but none is an instance of the other:

```
test :: ∀a. T a → Bool → Bool
test :: ∀a. T a → a → a
```

Several approaches have been proposed to deal with type inference for GADTs, most of them imposing several restrictions. GADTs are supported in GHC 7.10.1 [20] as described in [6], where type checking is based on type signatures explicitly given by the programmer. Recent work [15,22] describes a type inference algorithm that can avoid type signatures in a restricted number of cases.

In this article we present another type inferencing algorithm that accepts the declaration of functions based on GADTs without the need for type signatures (Sect. 3). Examples where types can and cannot be inferred, and issues related to the existence or not of principal types, are also discussed in Sect. 3.

Our type inference algorithm uses anti-unification (defined in Sect. 2.1) to capture the relation between the types of the alternatives. Type variables that are not related to GADTs are unified as usual. Cases involving recursive calls can be polymorphic recursive, and are handled as if each alternative is an overloaded definition. In this case, a constraint is added to the type of the recursive call. Constraint-set satisfiability of these constraints is used to construct a substitution that is used for instantiating the type of the alternative, in a process similar to the handling of overloading in System CT [2]. A brief review of System CT is given in Sect. 2.

2 Preliminaries

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

Meta-variable usage is defined in the paper as follows: x, y denote term variables, C, D data constructors, α, β ($a, b, ...$ in examples) type variables, T a type constructor, e a term, τ, ρ simple types, κ a constraint set, $x : \tau$ a constraint, σ a type, Γ a typing context, that is, a set of pairs written as $x : \sigma$, and S a substitution.

The notation $\overline{a}^{\,n}$, or simply $\overline{a}$, denotes the sequence $a_1, \ldots, a_n$, where $n \geq 0$. When used in a context of a set, it denotes the corresponding set of elements in the sequence $\{a_1, \ldots, a_n\}$.

A substitution is a function from type variables to simple type expressions (cf. Sect. 3.2). The identity substitution denoted by id. $S\sigma$ represents the capture-free operation of substituting $S(\alpha)$ for each free occurrence of α in σ.

We overload the substitution application on constraints, constraint sets and sets of types. Definition of application on these elements is straightforward. The symbol $\circ$ denotes function composition and $dom(S) = \{\alpha \mid S(\alpha) \neq \alpha\}$.

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

2.1 Anti-unification

A type τ is a generalization — also called (first-order) *anti-unification* [3] — of simple types $\overline{\tau}^{\,n}$ if there exist substitutions $\overline{S}^{\,n}$ such that $S_i(\tau) = \tau_i$, for $i = 1, \ldots, n$.

We call a function that gives the least generalization of a finite set of simple types the *least common generalization* (*lcg*).

An algorithm for computing the *lcg* of a finite set of types in presented in Fig. 1. The concept of least common generalization was studied by Gordon Plotkin [10,11], that defined a function for constructing a generalization of two symbolic expressions.

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

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

$$lcg'(\{\tau_1, \tau_2\} \cup \mathbb{T}, S) = lcg''(\tau, \tau', S') \quad \texttt{where } \begin{array}{l}(\tau, S_0) = lcg''(\tau_1, \tau_2, S) \\ (\tau', S') = lcg'(\mathbb{T}, S_0)\end{array}$$

$$\begin{array}{l} lcg''(C\,\overline{\tau}^{\,n},\ D\,\overline{\rho}^{\,m}, S) = \\ \quad \texttt{if } S(\alpha) = (C\,\overline{\tau}^{\,n},\ D\,\overline{\rho}^{\,m}) \texttt{ for some } \alpha \texttt{ then } (\alpha, S) \\ \quad \texttt{else} \\ \qquad \texttt{if } n \neq m \texttt{ then } (\beta, S[\beta \mapsto (C\,\overline{\tau}^{\,n},\ D\,\overline{\rho}^{\,m})]) \\ \qquad\quad \texttt{where } \beta \texttt{ is a fresh type variable} \\ \qquad \texttt{else } (\psi\,\overline{\tau'}^{\,n}, S_n) \\ \qquad\quad \texttt{where } (\psi, S_0) = \begin{cases} (C, S) & \texttt{if } C = D \\ (\alpha, S\,[\alpha \mapsto (C, D)]) & \texttt{otherwise, } \alpha \texttt{ is fresh} \end{cases} \\ \qquad\qquad\quad (\tau_i', S_i) = lcg''(\tau_i, \rho_i, S_{i-1}), \texttt{ for } i = 1, \ldots, n \end{array}$$

Fig. 1. Least common generalization

2.2 System CT

System CT is an extension of the Damas-Milner type system for dealing with overloading [1,2,13,14]. Our initial view for the definition of system CT was to consider a simple extension where a name (or symbol) could have more than one type assumption in a typing context. This led to the adoption of a closed world approach for overloading [1,2,21]. However for efficiency reasons, we have changed our initial idea about the support of only a closed world approach to overloading, due to the need (in a closed world) of checking constraint-set satisfiability for each function application. Nowadays, our view, highly influenced by Haskell's open world approach, is that an open world is the preferred approach for supporting overloading. We leave discussion of an optional, instead of mandatory, use of type classes, as well as a related motivation for changing Haskell's ambiguity rule, to future work.

The principal type of overloaded symbols is defined in system CT by computing the anti-unification of the types of the available definitions of these symbols in the typing context, instead of requiring them to be explicitly annotated in a class declaration.

The least common generalization of the finite set of types of the definitions of an overloaded symbol in a given context is taken as the (principal) type of the overloaded symbol. In system CT a type is denoted by $\forall\bar{\alpha}.\,\kappa.\,\tau$, where κ is a possibly empty constraint set and τ is a simple type (i.e. an unconstrained and unquantified type). A constraint in system CT is a pair $x : \tau$, where x is an overloaded symbol and τ is a simple type. For example, a typing context, $\Gamma_{==}$, in

which the equality symbol is overloaded with types *Int* and *Char* contains the following type assumptions:

$$(\texttt{==}) : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$$
$$(\texttt{==}) : \mathit{Char} \to \mathit{Char} \to \mathit{Bool}$$

In this case, the principal type of (==) in $\Gamma_{\texttt{==}}$ is obtained by the least common generalization of the two types of (==) in this typing context, and is given by:

$$\forall a.\{(\texttt{==}) : a \to a \to \mathit{Bool}\}\,.\, a \to a \to \mathit{Bool}$$

Constraint $(\texttt{==}) : a \to a \to \mathit{Bool}$ on this type is similar to Haskell's constraint $\mathit{Eq}\ a$, where such type of (==) is annotated, the difference being essentially the abscence of a constraint on `(/=)` that is also available in type class *Eq*. The purpose of the constraint is the same as in Haskell: in this case, to allow instantiation of type variable a in $\Gamma_{\texttt{==}}$ only for types *Int* and *Char*. Type inference in system *CT* is a process similar to type inference in Haskell. In particular, the use of overloaded symbols in expressions for which overloading is not resolved causes a constraint to be included in the inferred type. For example, consider:

```
insert a []     = [a]
insert a (b:x)
  | a == b      = b: x
  | otherwise   = b: insert a x
```

The type of *insert* in this typing context is inferred to be:

$$\mathit{insert} :: \forall a.\,\{(\texttt{==}) : a \to a \to \mathit{Bool}\}.\, a \to \texttt{[}a\texttt{]} \to \texttt{[}a\texttt{]}$$

In general, in a constrained type $\forall\, \overline{a}^{\,n}.\kappa.\tau$, κ is a set of constraints that restricts the set of types to which $\forall\, \overline{a}^{\,n}.\kappa.\tau$ may be instantiated: every instance must be such that the resulting constraint set must be satisfiable in the relevant typing context. Constraint set satisfiability is in general an undecidable problem [16,17], but it can be made decidable without significantly affecting the set of typeable programs [14]. See also [2,4,5,18].

During type inference, the substitution returned by the function — called *sat* — that computes a substitution that verifies (proves) satisfiability of a given constraint set in a given typing context — or, equivalently, that verifies whether the constraint set is entailed by a set of constraints on the types of definitions of symbols in the typing context — can be used to "improve" the constrained type. See e.g. [5,13,14] for definitions of *sat* and for the problem of constraint-set satisfiability (CS-SAT).

In a closed world, the substitution returned by *sat* is needed to improve the type of recursive functions. For example, consider the inference of the principal type of overloaded equality for lists, in context $\Gamma_{(\texttt{==})}$:

$$\begin{array}{lll} \texttt{[]} & \texttt{==}\ \texttt{[]} & \texttt{=}\ \mathit{True} \\ (a : x) & \texttt{==}\ (b : y) & \texttt{=}\ a\ \texttt{==}\ b\ \texttt{\&\&}\ x\ \texttt{==}\ y \\ \_ & \texttt{==}\ \_ & \texttt{=}\ \mathit{False} \end{array}$$

The (principal) type of `(==)` is inferred by considering firstly the types of `(==)` used in the recursive definition, initially given by:

$$\{\texttt{(==)} : a \to b \to \mathit{Bool}, \texttt{(==)} : \texttt{[}a\texttt{]} \to \texttt{[}b\texttt{]} \to \mathit{Bool}\}.\ \texttt{[}a\texttt{]} \to \texttt{[}b\texttt{]} \to \mathit{Bool} \qquad (1)$$

The first constraint on the type above comes from a `==` b, and the second from x`==`y. Note that the type of `(==)` cannot be inferred from the *lcg* of the types of `(==)` in the typing context, because a new definition is being given, and this in general will modify the *lcg*. Function *sat* comes to the rescue, being able to compute a substitution that is used to improve the type of `(==)` to:

$$\forall a.\{(\texttt{==}) : a \to a \to \mathit{Bool}\}.\ \texttt{[}a\texttt{]} \to \texttt{[}a\texttt{]} \to \mathit{Bool}$$

despite the existence of an infinite set of substitutions that can be used to instantiate the type (1) above:

$$\{\{a \mapsto \mathit{Int}, b \mapsto \mathit{Int}\}, \{a \mapsto \mathit{Char}, b \mapsto \mathit{Char}\}, \{a \mapsto \texttt{[}\mathit{Int}\texttt{]}, b \mapsto \texttt{[}\mathit{Int}\texttt{]}\}, \{a \mapsto \texttt{[}\mathit{Char}\texttt{]}, \{b \mapsto \texttt{[}\mathit{Char}\texttt{]}\}, \{a \mapsto \texttt{[[}\mathit{Int}\texttt{]]}, b \mapsto \texttt{[[}\mathit{Int}\texttt{]]}\}, \{a \mapsto \texttt{[[}\mathit{Char}\texttt{]]}, b \mapsto \texttt{[[}\mathit{Char}\texttt{]]}\}, \ldots\}$$

The *sat* algorithm is defined in [2,14].

Type inference for polymorphic recursion is treated in a similar way.

3 Type Inference

Let's say that a GADT function is a function such that the type of a parameter or of the result is a GADT type. Type inference of a GADT function involves, in our approach, the generalization of the types used in the defining alternatives. The substitution returned by *sat* is used to improve the inferred type, using a process of type inference that has the following phases:

1. The type of each defining alternative is inferred, with a constraint (included in the constraint set) for each recursive use of the GADT function.
2. The type of each equation j is improved by the substitution given by $sat(\kappa_j, \Gamma)$, where κ_j is the constraint set on the type inferred for equation j and Γ contains the type assumptions of used overloaded symbols together with the set $\{x : \sigma_i \mid i = 1, \ldots, n\}$, where x is the name being defined and σ_i is the type of the i-th equation in the definition of x that is not recursively defined.
3. Compute the *lcg* of the simple type of the alternatives.
4. Compute the substitution that is the most general unifier of the types of alternatives of the GADT function which do not involve a GADT constructor and apply this substitution to obtain the type of each alternative.

We present next some examples that illustrate the type inference process.

3.1 Examples

Example 1. Type Inference of function *test*

The types inferred for each alternative of the function *test*, presented in Sect. 1, are:

$$\begin{array}{lll} test & :: & T\ Bool \rightarrow a \rightarrow Bool \\ test & :: & T\ b \quad\ \rightarrow c \rightarrow c \end{array}$$

In cases such as this, where recursive calls do not occur, no restriction is generated, making the call to *sat* unnecessary. The *lcg* is then computed, taking the set of types of the alternatives as a parameter. The type obtained by *lcg* allows observation of the dependency that exists between the types of each alternative with those of the generalized type: types for which there is no association with the type of a GADT are unified. The generalization of the types of the alternatives in the definition of the function *test* yields:

$$test\ ::\ T\ b' \rightarrow c' \rightarrow d'$$

Now, b' is associated with a GADT, and c', d' are not. Thus, phase (4) above specifies that a and c should be unified, as well as *Bool* and c, resulting in:

$$test\ ::\ \forall a\,.\,T\ a \rightarrow Bool\ \rightarrow Bool$$

As discussed in Sect. 1, type $\forall a.\ T\ a \rightarrow a \rightarrow a$ could also serve as the type of *test*, and in that case expression (*test* $T2$ `'a'`) would be type-correct.

In Haskell, type inference for the function *test*generates *implication constraints* [15,22], given by (where $\sim$ is a type equality constraint and $\supset$ denotes an implication constraint):

$$(a \sim T\ b) \wedge (b \sim Bool \supset c \sim Bool) \wedge (a \sim T\ d) \wedge (c \sim e)$$

Type equality constraint $(b \sim Bool)$ is generated from *T1* n, type equality constraint $(c \sim Bool)$ is generated from the first alternative in the definition of *test*, type equality constraint $(c \sim e)$ from the second alternative in the definition of *test*, where e is the type of r, which is in this case free to be unified ($\{e \mapsto c\}$). The meaning of an implication constraint can be understood by considering that, in this example, $(b \sim Bool \supset c \sim Bool)$ indicates that if type variable b is instantiated to *Bool* then so must c. These constraints have substitution $\{c \mapsto Bool\}$ as a solution. Application of this substitution on the type of *test* yields type $T\ b \rightarrow Bool\ \rightarrow Bool$, which is the same type inferred by our algorithm. However, type variable c is considered *untouchable* in the implication constraint, and then type inference fails. Type variables which occur

in implication constraints are considered untouchable within these constraints, and can only be substituted as a result of applying substitutions obtained as a result of solving other constraints. In GHC 7.6.x type inference proceeds as outlined, but from version 7.8.1 a more restricted set of GADT functions for non-annotated types was adopted.

Example 2. Type Inference of function *eval*

In the definition of *eval*, presented in Sect. 1, recursive calls involving polymorphic recursion occur in some alternatives, while the type of *eval* has not been inferred yet. To handle such cases, constraints are generated from the type required for each recursive call. These constraints are subsequently used in the type improvement process.

The alternative with pattern on constructor *If* has recursive calls for all arguments of the constructor (l, $e1$ and $e1$). l has type $Term\ Bool$, and $e1, e2$ have type $Term\ a$. Constraints $\{eval : Term\ a \to b, eval : Term\ Bool \to Bool\}$ are generated, and the type of the alternative is inferred also as $Term\ a \to b$. Type inference for the constructor *Pair* proceeds in a similar way. The types inferred for each alternative are as follows:

$$
\begin{array}{ll}
(Lit\ i) & eval\ ::\ Term\ Int \to Int \\
(Inc\ t) & eval\ ::\ Term\ Int \to Int \\
(IsZ\ i) & eval\ ::\ Term\ Bool \to Bool \\
(If\ l\ e1\ e2) & eval\ ::\ \{eval\ :\ Term\ Bool \to Bool, \\
 & \qquad\qquad eval\ :\ Term\ a \to b\}.\ Term\ a \to b \\
(Pair\ x\ y) & eval\ ::\ \{eval\ :\ Term\ c \to e, \\
 & \qquad\qquad eval\ :\ Term\ d \to f\}.\ Term\ (c,d) \to (e,f)
\end{array}
$$

After this, the types of the alternatives which contain constraints are subject to type improvement, which consists of the application of the substitution given by $sat(\kappa, \Gamma)$, where κ is the possibly empty constraint set in the type of each alternative and $\Gamma = \{eval : Term\ Int \to Int, eval : Term\ Bool \to Bool\}$. After type improvement the following types are inferred for each alternative:

$$
\begin{array}{ll}
(Lit\ i) & eval\ ::\ Term\ Int \to Int \\
(Inc\ t) & eval\ ::\ Term\ Int \to Int \\
(IsZ\ i) & eval\ ::\ Term\ Bool \to Bool \\
(If\ l\ e1\ e2) & eval\ ::\ Term\ a \to a \\
(Pair\ x\ y) & eval\ ::\ Term\ (c,d) \to (c,d)
\end{array}
$$

The type inferred for *eval* is the *lcg* of the types of the alternatives, given by:

$$eval\ ::\ \forall a.\ Term\ a \to a$$

In this case, all types are associated with the GADT, what characterizes them as types that should not be unified, and, as in this case all types are associated to the GADT, we have that, in this case, the type inferred is the principal type.

Example 3. In some cases anti-unification does not capture the relationship between types of alternatives. Consider for example the following function, presented in [18]:

```
data Erk a b where
     I :: Int→ Erk Int b
     B :: Bool→ Erk a Bool

f (I a) = a + 1
f (B b) = b && True
```

The generalization of the types of the alternatives in the definition of f is: $Erk\ a\ b \rightarrow c$. Since type variable c is not associated to a GADT, types Int and $Bool$ are unified, causing the definition of f to be rejected. However, for example with annotated type $Erk\ a\ a \rightarrow a$ this function can be given a proper type.

3.2 Term and Type Syntax

The context-free syntax of terms and types is presented in Fig. 2. For simplicity and following common practice, kinds are not considered in type expressions and type expressions which are not simple types are not explicitly distinguished from simple types. Type expression variables are called simply type variables. There is a distinguished type constructor that is written as an infix operator, $\tau \rightarrow \tau'$, as usual.

$$
\begin{array}{lll}
\text{Terms} & e & ::= x \mid C \mid \lambda x.\, e \mid e\, e' \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid \texttt{case}\ e\ \texttt{of}\ \overline{C\, \overline{x} \rightarrow e} \\
\text{Simple types} & \tau & ::= \alpha\, \overline{\tau} \mid T\, \overline{\tau} \\
\text{Type schemes} & \sigma & ::= \forall\, \overline{\alpha}.\, \kappa.\, \tau
\end{array}
$$

Fig. 2. Syntax of terms and types

We use the following operations over typing contexts:

$$
\begin{array}{ll}
\Gamma(x) & = \{\sigma \mid x : \sigma \in \Gamma\} \\
\Gamma, x : \sigma & = (\Gamma - \{x : \sigma \mid \ \sigma \in \Gamma(x)\}) \cup \{x : \sigma\}
\end{array}
$$

We let: (i) $tv(\sigma)$ denote the set of free type variables in σ, (ii) $gtv(\sigma)$ denote the set of free type variables that occur in the type of a GADT type constructor, (iii) $gtc(\tau)$ represent the set of GADT type constructors occurring in τ and (iv) $rtv(\tau)$ denote the set of free type variables that occur in the type of a recursive algebraic data type, such as lists and trees (the set $rtv(\tau)$ is used to avoid skolemization of type variables that occur in the type of the result of a generalized GADT function).

We use constraints to express a relationship between the return type of a function and its parameters, in case the type of parameters have a GADT constructor.

We also use the following notation to return the sets of constraints that contain types that mention GADT constructors:

$$\kappa^{\star}_{x} = \{x : \tau \in \kappa \mid gtc(\tau) \neq \emptyset\}$$

3.3 Algorithm Definition

For simplicity, we consider a language that is essentially core-ML extended with GADT functions — that is, we do not include inference of types of expressions with overloaded symbols. Readers interested in type inference for overloading are referred to [13].

The proposed algorithm is defined as a syntax-directed proof system, using formulas of the form $\Delta \mid \Gamma \vdash e : (\kappa.\tau, S)$, where Δ is an environment of names of recursive function definitions that contains constraints to be used in the process of type improvement for case branches involving GADTs, $\kappa.\tau$ is the type inferred for e and S is a substitution (used to instantiate type variables for obtaining type $\kappa.\tau$). Notation $\delta(x, \tau, \Delta)$ associates, with a symbol x and a type τ, constraint set $\{x : \tau\}$, if x is a recursively defined symbol, otherwise an empty constraint set. It is defined as:

$$\delta(x, \tau, \Delta) = \texttt{if}\ \ x \in \Delta\ \texttt{then}\ \{x : \tau\}\ \texttt{else}\ \emptyset$$

$$\frac{\Gamma(x) = \forall\, \overline{\alpha}.\, \tau' \quad \tau = [\overline{\alpha} \mapsto \overline{\beta}]\tau' \quad \overline{\beta}\ \text{fresh}}{\Delta \mid \Gamma \vdash x : (\delta(x, \tau, \Delta).\, \tau, id)}\ (\texttt{VAR})$$

$$\frac{\Delta \mid \Gamma, x : \alpha \vdash e : (\kappa.\, \tau, S) \quad \alpha\ \text{fresh}}{\Delta \mid \Gamma \vdash \lambda x.e : (\kappa.\, S\,\alpha \to \tau, S)}\ (\texttt{LAM})$$

$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash e : (\kappa.\, \tau_1, S_1) \qquad S' = \mathit{unify}(\{\tau_2 \to \alpha = \tau_1\}) \\ \Delta \mid S_1\,\Gamma \vdash e' : (\kappa'.\, \tau_2, S_2) \qquad S = S' \circ S_2 \circ S_1 \quad \alpha\ \text{fresh}\end{array}}{\Delta \mid \Gamma \vdash e\ e' : (S\,(\kappa \cup \kappa'.\, \alpha), S)}\ (\texttt{APP})$$

$$\frac{\begin{array}{c}\Delta, x \mid \Gamma \vdash e' : (\kappa_1.\, \tau_1, S_1) \qquad \overline{\alpha} = tv(\kappa_1.\, \tau_1) - tv(\Gamma) \\ \Delta \mid \Gamma, x : \forall\, \overline{\alpha}.\, \tau_1 \vdash e : (\kappa_2.\, \tau_2, S_2) \qquad S = S_2 \circ S_1\end{array}}{\Delta \mid \Gamma \vdash \texttt{let}\ \ x = e'\ \ \texttt{in}\ \ e : (S(\kappa_1 \cup \kappa_2.\, \tau_2), S)}\ (\texttt{LET})$$

$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash e : (\kappa'.\, \tau', S) \quad \Delta \mid S\,\Gamma \vdash_{\texttt{alts}} (\{\overline{C\,\overline{x} \to e_1}\}, \tau') : \overline{(\kappa_i.\, \tau_i' \to \tau_i, S_i)} \\ S\,\Gamma \Vdash_x \overline{\kappa_i.\, \tau_i' \to \tau_i} \rightsquigarrow \tau' \to \tau \\ x\ \text{name of GADT function with constructors}\ \overline{C}\end{array}}{\Delta \mid \Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{\overline{C\,\overline{x} \to e_1}\} : (\tau, S)}\ (\texttt{CASE})$$

Type inference rules are standard, with the exception of rules (VAR) and (CASE). The (VAR) rule generates a constraint for each symbol in Δ, used for

improvement of types of GADT functions. Each variable x that is not in Δ has a type with an empty constraint set (remember that, for simplicity reasons, overloading is not treated in this paper). The (CASE) rule is the main part of the algorithm. First, the type of case scrutinee e is inferred. Then, the type of each case alternative is inferred (in the textual order, but the order is not relevant). Finally, if the case expression involves GADT constructors, the type of the case expression is improved, by using a separate type improvement judgement (since case alternatives are not expressions). Distinct case alternatives for the same constructor must be unified, but in this paper we consider for simplicity that each case alternative has a distinct constructor.

$$\frac{\Delta \mid \Gamma \vdash_{\texttt{alt}} (C\,\overline{x} \to e, T\,\overline{\tau'}) : (\kappa.T\,\overline{\tau'} \to \tau, S)}{\Delta \mid \Gamma \vdash_{\texttt{alts}} (\{C\,\overline{x} \to e\}, T\,\overline{\tau'}) : \{(\kappa.T\,\overline{\tau'} \to \tau, S)\}} \ (\texttt{ALTSEnd})$$

$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash_{\texttt{alts}} (\overline{C\,\overline{x'} \to e_1}, T\,\overline{\tau'}) : \overline{(\kappa'.\tau_1, S_1)} \\ \Delta \mid \Gamma \vdash_{\texttt{alt}} (C\,\overline{x} \to e, T\,\overline{\tau'}) : (\kappa.T\,\overline{\tau'} \to \tau, S)\end{array}}{\Delta \mid \Gamma \vdash_{\texttt{alts}} ((\{C\,\overline{x} \to e\} \cup \overline{C\,\overline{x'} \to e_1}), T\,\overline{\tau'}) : \{(\kappa.T\,\overline{\tau'} \to \tau, S)\} \cup \overline{(\kappa'.\tau_1, S_1)}} \ (\texttt{ALTSRec})$$

$$\frac{\begin{array}{c}\Gamma(C) = \forall\,\overline{\alpha}.\,\overline{\tau_1} \to T\,\overline{\tau_2} \\ S' = \mathit{unify}(\{T\,\overline{\tau_2} = T\,\overline{\tau'}\}) \quad \Delta \mid S'(\Gamma, \overline{x : \tau'}) \vdash e : (\kappa.\,\tau, S)\end{array}}{\Delta \mid \Gamma \vdash_{\texttt{alt}} (C\,\overline{x} \to e, T\,\overline{\tau'}) : (\kappa.\,T\,\overline{\tau'} \to \tau, S)} \ (\texttt{ALT})$$

In order to infer the type of a case alternative, we need to unify its constructor range type with the type inferred for the case scrutinee, producing a substitution that is used to instantiate the types of the parameters, and add them to the typing context to infer the type of the right-hand side of the alternative.

The judgement $\Gamma \Vdash_x \overline{(\kappa_i.\,\tau_i' \to \tau_i)} \rightsquigarrow (\tau' \to \tau, S)$ denotes the type improvement necessary for the inference of types of functions defined by pattern matching on a GADT function named x. Given a typing context Γ and, for each alternative i, a set of constrained types $\kappa_i.\,\tau_i' \to \tau_i$, type improvement yields the improved type $\tau' \to \tau$. Note that only functions that have alternatives with polymorphic recursion generate constraints. $\mathit{sat}(\kappa, \Gamma)$ computes the improvement substitution S for a set of constraints κ, using type assumptions given by Γ. This judgement uses function *specialize* which computes a improvement substitution based on the types of the case alternatives and their generalization.

$$\begin{array}{ll}\mathit{specialize}(\tau, \emptyset) & = \mathit{id} \\ \mathit{specialize}(\tau, (\{\tau'\} \cup \mathbb{T})) & = \mathit{unify}(\{\tau = \tau'\}) \circ \mathit{specialize}(\mathbb{T})\end{array}$$

The type improvement judgement is defined as:

$$\frac{\begin{array}{c}\overline{S_i} = \mathit{sat}((\kappa_i)^\star_x, \{\overline{x : \tau_i' \to \tau_i}\} \cup \Gamma) \\ \tau_1 \to \tau_2 = \mathit{lcg}(\overline{S_i\,(\tau_i' \to \tau_i)}) \qquad \overline{K} \text{ are fresh Skolem constants} \\ \overline{\alpha} = \mathit{gtv}(\tau_1) - \mathit{rtv}\ (\tau_2) \qquad S = \mathit{specialize}([\overline{\alpha} \mapsto \overline{K}](\tau_1 \to \tau_2), \overline{S_i\,(\tau_i' \to \tau_i)})\end{array}}{\Gamma \Vdash_x \overline{\kappa_i.\,\tau_i' \to \tau_i} \rightsquigarrow (S(\tau_1 \to \tau_2), S)} \ (\texttt{IMPROVE})$$

This judgement works as follows. For each equation i of a GADT function x, let $(\kappa_i)^\star_x$ be the set of constraints that mention GADT type constructors (in the constraint set of the type of the i-th equation) and let S_i be the satisfiability substitution for this constraint set, in a typing context that contains type assumptions corresponding to all alternatives. Then, let $\tau_1 \to \tau_2$ be the *lcg* of all $\overline{S_i\,(\tau'_i \to \tau_i)}$). Now, we "skolemize" $\overline{\alpha}$ (i.e. treat them as non-unifiable), the set of type variables introduced by the generalization of types of parameters of a GADT. Type variables that occur in the return type of a function *and* also in the generalization of a parameter of a recursive algebraic type are not skolemized (for example, when the type `[`a`]` of the result is obtained from, say, the generalization of `[`*Int*`]` and `[`*Bool*`]`). The inferred types of case alternatives are then unified with non-skolemized type variables. The substitutions computed by satisfiability and unification are applied to the generalized type, which is then returned.

It is worth mentioning that the improvement judgement is conservative over non-GADT types, since $\kappa = \emptyset$ when alternative types do not involve GADTs, and no variable is skolemized, so all types must be unified.

3.4 GADT and Principal Type

In [22] Vytiniotis et al. argue that the principal type property offers fewer benefits than a guarantee of type safety (i.e. that well-typed programs will not cause an error at run-time). Consider for example the function *eval*, presented in Sect. 1, but now consider that it is declared with only the first alternative:

$$eval\ \texttt{(}Lit\ i\texttt{) = }i$$

Our algorithm infers type ($Term\ Int \to Int$) for *eval*, but in Haskell the following type annotations would be allowed for *eval*: $\forall a.\ Term\ a \to a$ and $\forall a.\ Term\ a \to Int$. None of these types is an instance of the other.

In our view, the type $Term\ Int \to Int$ is a good choice in this case, since it avoids expressions such as, for example, *eval* `(`*IsZ*`(`*Lit* `1))`, for which there exists no alternative in the definition of *eval*. With the algorithm given in [22] the following implication constraint is generated during type inference:

$$(a \sim Term\ \ b) \wedge (b \sim Int \supset c \sim Int)$$

Note that substitution $\{c \mapsto Int\}$ is a solution to this implication constraint and application of this substitution leads to the inference of type $Term\ a \to Int$. However, in this constraint variable c is considered untouchable, and then type inference fails in GHC. Again, in GHC version 7.6.x, the function *eval* defined with only this alternative would have inferred type $\forall a.\ Term\ a \to Int$; from version 7.8.1 type inference fails, due to type variable being considered untouchable.

On the other hand, by adding the alternative of constructor *IsZ*, where the type *Term Bool* is returned by the constructor, the type of *eval* becomes: $\forall a.$ *Term* $a \rightarrow a$, which is the same as the type inferred by our algorithm. It is important to point out that the type *Term Int* $\rightarrow$ *Int*, inferred by the alternative of constructor *Lit*, is an instance of this type, in contrast with the case of *Term* $a \rightarrow$ *Int*.

```
eval (Lit i) = i
eval (IsZ i) = 0 == eval i
```

Back to type safety, it would be desirable that in this case the type inference algorithm restricts instances of a to either *Int* or *Bool*; however, this seems to need a special way of constraining polymorphic types.

In many cases, such as that of Example 1, a relation between the types of alternatives is not expressed by the code in these alternatives. In these cases there is no guarantee that the inferred type is the principal type.

4 Related Work

Peyton Jones *et al.* present an extension of Haskell's type system for the support of GADTs [6,7]. The verification of the types of GADT functions is done using type annotations. These types, called rigid types, are propagated to inner scopes by means of some specific rules. Pottier and Régis-Gianas [12] define a two-pass type inference algorithm, separating traditional Hindley-Milner type inference from the propagation of explicit type annotations. This separation makes the mechanism of type propagation more efficient.

The type inference algorithm used in [15,22], called OutsideIn, extracts type constraints from expressions occurring in inner scopes and solves these constraints in the outermost scope, avoiding an *ad hoc* approach for the propagation of rigid types. Besides using a more natural mechanism for propagation of annotated types, this approach enables more helpful error messages and type inference in a restricted number of function declarations. In these cases a rather restrictive rule is adopted in the definition of untouchable variables, so that only the types of functions for which the existence of a principal type can be guaranteed are inferred. In [19] Sulzmann and Schrijvers introduce some ideas adopted in the OutsideIn algorithm.

Lin and Sheard present the Pointwise GADT type system [9], that uses a modified unification algorithm to support parametric instantiation and type indexing. In [8] Lin proposes algorithm $\mathcal{P}$, more restrictive than Pointwise, that does not require type annotations. The algorithm applies generalization only in patterns of alternatives and supports polymorphic recursion. Differently from our proposal, which handles polymorphic recursion similarly to overloading, algorithm $\mathcal{P}$ uses an approach similar to that used by an iteration limit to guarantee termination.

5 Conclusion

In this paper we have presented a type inference algorithm in the presence of GADTs. The ideas behind the algorithm are intuitive and easy to understand.

The presented algorithm handles alternative definitions of a defined symbol x as if they were overloaded definitions of x, in a closed world approach to overloading, with support for polymorphic recursion. The algorithm makes use of anti-unification to capture the relation between the types of distinct alternatives of a function that has a parameter or returns a GADT. Types which must not be unified are separated, before unifying the types of the alternatives. This enables type inference for functions that typically require type annotations in other implementations, such as that of GHC.

Further study in order to provide support for type annotations is necessary. When there is a relation from the types of arguments to the type of the result of a GADT function which is not made explicit in the code (e.g. Example 3), our type inference algorithm can reject expressions that could be considered type-correct.

References

1. Camarão, C., Figueiredo, L.: Type inference for overloading without restrictions, declarations or annotations. In: Middeldorp, A., Sato, T. (eds.) FLOPS 1999. LNCS, vol. 1722, pp. 37–52. Springer, Heidelberg (1999)
2. Camarão, C., Figueiredo, L., Vasconcellos, C.: Constraint-set Satisfiability for Overloading. In: Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, pp. 67–77. ACM (2004)
3. Chang, C.C., Keisler, H.J.: Model Theory: Dover Books on Mathematics, 3rd edn. North-Holland Press, New York (2012)
4. Demoen, B., de la Banda, M.G., Stuckey, P.J.: Type Constraint Solving for Parametric and Ad-hoc Polymorphism. In: Proceedings of the 22nd Australasian Computer Science Conference (1999)
5. Jones, M.: Simplifying and Improving Qualified Types. In: Proceedings of ACM Conference on Functional Programming and Computer Architecture, FPCA 1995, pp. 160–169 (1995)
6. Jones, S.P., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. SIGPLAN Not. **41**(9), 50–61 (2006)
7. Jones, S.P., Washburn, G., Weirich, S.: Wobbly types: type inference for generalised algebraic data types. Technical report MS-CIS-05-26, University of Pennsylvania, Microsoft Research (2004). http://research.microsoft.com/apps/pubs/default.aspx?id=65143
8. Lin, C.K.: Practical type inference for the GADT type system. Ph.D. thesis, Portland State University, Portland, OR, USA (2010)
9. Lin, C.K., Sheard, T.: Pointwise generalized algebraic data types. In: Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI 2010, pp. 51–62. ACM, New York (2010)
10. Plotkin, G.D.: A note on inductive generalisation. Mach. intell. **5**(1), 153–163 (1970)

11. Plotkin, G.D.: A further note on inductive generalisation. Mach. Intell. **6**, 101–124 (1971)
12. Pottier, F., Régis-Gianas, Y.: Stratified type inference for generalized algebraic data types. SIGPLAN Not. **41**(1), 232–244 (2006)
13. Ribeiro, R., Camarão, C.: Ambiguity and context-dependent overloading. J. Braz. Comput. Soc. **19**(3), 313–324 (2013)
14. Ribeiro, R., Camarão, C., Figueiredo, L.: Terminating constraint set satisfiability and simplification algorithms for context-dependent overloading. J. Braz. Comput. Soc. **19**(4), 423–432 (2013)
15. Schrijvers, T., Jones, S.P., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. SIGPLAN Not. **44**(9), 341–352 (2009)
16. Smith, G.: Polymorphic type inference for languages with overloading and subtyping. Ph.D. thesis, Cornell University (1991)
17. Smith, G.: Principal type schemes for functional programs with overloading and subtyping. Sci. Comput. Program. **23**(2–3), 197–226 (1994)
18. Stuckey, P., Sulzmann, M.: A Theory of overloading. In: Proceedings of the 7th ACM International Conference on Functional Programming, pp. 167–178 (2002)
19. Sulzmann, M., Schrijvers, T., Stuckey, P.J.: Type Inference for GADTs via Herbrand Constraint Abduction (2008)
20. Team, G., et al.: The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.10.1 (2015)
21. Vasconcellos, C.: Inferência de tipos com suporte para sobrecarga baseada no sistema CT. Ph.D. thesis, Universidade Federal de Minas Gerais, Minas Gerais, Brasil (2004)
22. Vytiniotis, D., Jones, S.P., Schrijvers, T., Sulzmann, M.: OutsideIn(X): modular type inference with local assumptions. J. Funct. Program. **21**(4–5), 333–412 (2011)