Type Inference for GADTs, OutsideIn and Anti-unification

Gabriela Moreira
Santa Catarina State University
Brazil
gabrielamoreira05@gmail.com

Cristiano Vasconcellos
Santa Catarina State University
Brazil
cristiano.vasconcellos@udecs.br

Rodrigo Ribeiro
Federal University of Ouro Preto
Brazil
rodrigo@decsi.ufop.br

Abstract

Support for generalized algebraic data types (GADT) in extensions of Haskell allows functions defined over GADTs to be written without the need for type annotations in some cases, but it requires type annotations in most of them. This paper presents a type inference algorithm for GADTs that extends OutsideIn algorithm using anti-unification to capture the relationship between the types of arguments and result of GADT functions. This approach allows inference in cases where the relationship between types of pattern matches is explicit in the code, allowing the type annotation in cases where the relationship is not explicit.

CCS Concepts • **Theory of computation** → Semantics and reasoning; *Functional constructs*; • **Software and its engineering** → *Functional languages*;

Keywords GADT, type inference, anti-unification

ACM Reference Format:
Gabriela Moreira, Cristiano Vasconcellos, and Rodrigo Ribeiro. 2018. Type Inference for GADTs, OutsideIn and Anti-unification. In *Proceedings of Brazilian Symposium (SBLP'2018).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Generalized Algebraic Data Types (GADTs) are a powerful extension to algebraic data types (ADTs) of functional languages like Haskell[1] and ML, and widely used nowadays.[2] Type inference for GADT is difficult and, in many cases, lacks principal type property.

[1]GADT is supported as an extension in GHC.
[2]Examples of applications of GADTs are described, for example, in [4, 8, 18, 19].

SBLP'2018, September 2018, São Carlos

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Haskell adopts an ingenious algorithm for type checking in presence of GADTs. This algorithm, called OutsideIn, generates constraints from outside a GADT pattern match to the inside, but not vice versa [17]. However, in order to guarantee the principal type property and decidability of constraints solver, in just a few cases, OutsideIn is able to infer the type of an expression. OutsideIn requires type annotation for simple functions, such as f:

```
data H a where
  H1 :: Bool → H Bool
  H2 :: Int  → H Int

f = λx. case x of
      H1 n → n
      H2 n → n
```

Gelain *et al.* [7] proposes an approach that uses anti-unification on the types of alternatives to capture the relationship between types associated with GADTs. However, there are cases where the relationship of arguments with GADT is not explicit in the code. For these cases it is necessary type annotation, which is not supported by the algorithm. This paper presents an algorithm that combines these two approaches. The main contribution involves adding type annotation support to the previous work by using a verification very similar to the one in OutsideIn algorithm. Additionally, we present a problem in the previous anti-unification algorithm and a solution including rigid type variables.

The base language for our system is defined in Section 2. The main ideas of the use of anti-unification on type inference algorithm are presented in Section 3, through some examples. That section also presents a simplified version of the algorithm proposed in [7]. The simplified version does not handle recursive calls, which usually involves polymorphic recursion in presence of GADTs. Section 4 presents a modification of OutsideIn by adding the approach described in Section 3. Section 5 discusses related work and Section 6 concludes.

2 Types, Terms and Basic Concepts

Before we discuss our approach, we briefly introduce our base language. We omit data type declarations for simplicity, and assume that type bindings for data constructors introduced in GADTs and ADTs declarations are given in an initial type environment.

Term variables	x, y		Type constructors	T	
Type variables	a, b, c α, β, γ		Data constructors	K	
Term	e	::=	$x \mid K \mid \lambda x.\, e \mid e\ e'$		
		$\mid$	$\texttt{let}\ g = e\ \texttt{in}\ e'$		
		$\mid$	$\texttt{let}\ g :: \sigma = e\ \texttt{in}\ e'$		
		$\mid$	$\texttt{case}\ x\ \texttt{of}\ \overline{p \to e}$		
Pattern	p	:=	$K\ \overline{x}$		
			$(p_1, \ldots, p_n)$		
Simple type	τ, υ	::=	$\nu \mid \tau \to \upsilon \mid T\ \overline{\tau}$		
Type scheme	σ	::=	$\tau \mid \forall\, \overline{\alpha}.\, \tau$		
Typing context	Γ	::=	$\emptyset \mid \Gamma \cup \{\nu : \sigma\}$		
Constraint	C, D	::=	$\tau \sim \upsilon \mid C \wedge C \mid \epsilon$		
Proper Constr.	F	::=	$C \mid [\overline{\alpha}] \forall\, \overline{b}.\, C \supset F$		
			$\mid F \wedge F$		
Substitution	S	::=	$\emptyset \mid S \cup [\alpha \mapsto \tau]$		

Figure 1. Syntax of terms and types

The context-free syntax of types and terms is defined in Figure 1. Programs are represented by terms in lambda calculus together with let bindings (possibly with a user-supplied type signature) and case expressions to perform pattern matching. For simplicity, we consider a simplified syntax for patterns, that does not include nested patterns, except for tuples to combine multiple pattern for match. For example, function *test* (taken from [17]):

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

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

can be translated to the core language as:

```
λx. case x of
      (T1 n, _ ) → n > 0
      (T2,   r ) → r
```

In the examples we used Haskell-like syntax. For simplicity and following common practice, kinds are not considered in type expressions. Type expressions which are not simple types are not explicitly distinguished from simple types. There is a distinguished function type constructor that is written as an infix operator, $\tau \to \tau'$, as usual.

Meta-variables can be used primed or subscripted. We use $\overline{a}$, to denote 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 S is a function from type variables to simple types. The identity substitution is denoted by id and $\circ$ denotes substitution composition. $S\sigma$ represents the capture-free operation of substituting $S(\alpha)$ for each free occurrence of α in σ. Substitution application is extended to sets of types and typing contexts as usual. The notation $S[\overline{\alpha \mapsto \tau}]$ denotes the substitution S' such that $S'(\beta) = \tau_i$ if $\beta = \alpha_i$, for $i = 1, \ldots, n$, otherwise $S(\beta)$. Also, $[\overline{\alpha \mapsto \tau}] = id[\overline{\alpha \mapsto \tau}]$.

We also use the following notation: i) $ftv(\sigma)$ for the set of free type variables in σ; ii) $gtv(\tau)$ for the set of free type variables that occur in τ as a parameter of a GADT type constructor.

2.1 Anti-Unification

A type τ is a generalization (also called first-order *anti-unification* [2]) for a set of simple types $\{\overline{\tau}\}$ if there exist substitutions $\overline{S}$ such that $S_i(\tau) = \tau_i$, for $i = 1, \ldots, n$. A generalization τ of $\{\overline{\tau}\}$ is a *least generalization* if, for any generalization τ' of $\{\overline{\tau}\}$, there exists a substitution S such that $S\tau' = \tau$.[3]

A function that gives a least generalization of a finite set of simple types will be called *lcg*. An algorithm for computing such function is given in Figure 2, where we use meta-variables X, Y, Z to denote either a type constructor or type variable, and S is a finite mapping from type variables to pairs of simple types.

$$lcg(S) = \tau \quad \texttt{where}\ (\tau, \phi) = lcg'(S, id),\ \texttt{for some}\ \phi$$

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

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

$$lcg_2(X\ \overline{\tau}^n, X'\ \overline{\rho}^m, \phi) =$$

if $\phi(a) = (X\ \overline{\tau}^n, X'\ \overline{\rho}^m)$ for some a **then** (a, ϕ)

else if $n = m$ **then** $(\psi\ \overline{\tau'}^n, \phi_n)$ where

$$(\psi, \phi_0) = \begin{cases} (X, \phi) & \text{if } X = X' \\ (a, \phi\,[a \mapsto (X, X')]) & \text{otherwise} \end{cases}$$

$$(\tau_i', \phi_i) = lcg_2(\tau_i, \rho_i, \phi_{i-1}), \text{for } i = 1, \ldots, n$$

else if $n = 0$ or $m = 0$

then $(a, \phi[a \mapsto (X\ \overline{\tau}^n, X'\ \overline{\rho}^m)])$ where a is a fresh type variable

else (τ', τ'', ϕ'') where

$(\tau', \phi') = lcg_2(X\ \overline{\tau}^{n-1}, X'\ \overline{\rho}^{m-1}, \phi)$

$(\tau'', \phi'') = lcg_2(\tau_m, \rho_m, \phi')$

Figure 2. Least Common Generalization

As an example of the use of *lcg*, consider the following types (of functions *map* on lists and trees, respectively):

[3] The concept of least common generalization was studied by Gordon Plotkin [14, 15], that defined a function for constructing a generalization of two symbolic expressions.

$$(a \to b) \to [a] \to [b]$$
$$(a \to b) \to \texttt{Tree}\ a \to \texttt{Tree}\ b$$

A call of *lcg* for a set with these types yields type $(a \to b) \to c\ a \to c\ b$, where c is a generalization of type constructors `[]` and *Tree* (for c to be used in cb, mapping $c \mapsto ([], Tree)$ is saved in parameter ϕ of lcg_2, to be reused).

3 Basic Idea

Let's call *GADT term* a `case` term that has a parameter or the result of a GADT type. In the approach proposed by Gelain *et al.* [7], type inference for a GADT term uses the generalization of the types inferred for all alternatives in its definition, in order to determine which types are associated with a GADT.

Let us consider a GADT term defined by:

$$\begin{array}{l}\lambda x.\ \texttt{case x of}\\ \quad (p_{11}, \ldots, p_{1n}) \to e_1\\ \quad \ldots\\ \quad (p_{m1}, \ldots, p_{mn}) \to e_m\end{array}$$

Type inference for the definition of terms, in a typing context Γ, consists of the following steps:

Step 1 Infer a type $(\tau_{i1}, \ldots, \tau_{in}) \to \tau'_i$ for each alternative $(p_{i1}, \ldots, p_{in}) \to e_i$, for $i = 1, \ldots, m$.

Step 2 Compute a least common generalization $(\rho_1, \ldots, \rho_n) \to \rho'$ of the types inferred for all the alternatives, that is, $(\rho_1, \ldots, \rho_n) \to \rho' = lcg(\{(\tau_{i1}, \ldots, \tau_{in}) \to \tau'_i\}^{i=1..m})$. This is done in order to capture the relationship between the types of each alternative and the GADT type, as illustrated by the examples below.

Step 3 Compute the most general unifier S that unifies the types of all the alternatives, where type variables in these types that occur as parameters of a GADT type are considered to be rigid, and the type obtained by skolemizing type variables in $(\rho_1, \ldots, \rho_n) \to \rho'$ that occur as parameters of a GADT type.

Step 4 Finally, the type inferred is $S((\rho_1, \ldots, \rho_n) \to \rho')$, where $(\rho_1, \ldots, \rho_n) \to \rho'$ is the type computed in Step 2 and S is the substitution computed in Step 3.

This type inference process is illustrated next through some examples.

Example 1. *Type Inference of function test*
Consider type inference for function *test* presented in Section 2. The types inferred for the alternatives in the definition of *test* are:

$$test\ ::\ T\ Bool \to a \to Bool$$
$$test\ ::\ T\ b \to c \to c$$

The *lcg* of these types yields type $T\ b' \to c' \to d'$. Note that type b' occurs as a parameter of a GADT type constructor, and thus, the types which correspond to b' in the type of each alternative may be distinct. On the other hand, the types corresponding to c' should be the same in each alternative, as well as for d'. This is checked by computing the most general unifier of the types of all the alternatives and type $T\ k \to c' \to d'$, where k is a skolem constant (that is, type variables that occur as a parameter of a GADT type constructor are not unified). The type inferred for function *test* is thus:

$$test\ ::\ \forall a.\ T\ a \to Bool \to Bool$$

$\forall a.\ T\ a \to a \to a$ could also serve as a type for *test*, and in that case expression (*test T2* `'a'`) would be type-correct. Note that none is an instance of the other and allowing a more general type signature for *test*, such as for example $\forall a, b.T\ a \to b \to b$, would not preserve type soundness since, for example, application *test* (*T1 3*) 3 == 5 would be type correct, and result in runtime type error.

In Haskell, type inference for function *test* generates *implication constraints* [17, 22], given by (where $\sim$ denotes type equality and $\supset$ denotes implication):

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

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 $T\ b \to Bool \to Bool$, 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 [7].

Example 2. In some cases anti-unification does not capture the relationship between the types of the alternatives and the GADT type, as illustrated by the following example, presented in [20]:

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

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

The types inferred for the first and second alternatives in the definition of g are, respectively, $(Erk\ Int\ b) \to Int$ and $(Erk\ a\ Bool) \to Bool$. The least common generalization of these types is: $(Erk\ a'\ b') \to c$. In our approach, since type

$$\boxed{\Gamma \vdash e : (\tau, S)}$$

$$\frac{\forall\, \overline{\alpha}.\, \tau' \in \Gamma \quad \tau = [\overline{\alpha} \mapsto \overline{\beta}]\tau' \quad \overline{\beta} \text{ fresh}}{\Gamma \vdash x : (\tau, id)}\ (\text{VAR}) \qquad \frac{\Gamma, x : \alpha \vdash e : (\tau, S) \quad \alpha \text{ fresh}}{\Gamma \vdash \lambda x.e : (S\,\alpha \to \tau, S)}\ (\text{ABS})$$

$$\frac{\begin{array}{ll}\Gamma \vdash e_1 : (\tau_1, S_1) & S' = unify(\{\tau_2 \to \alpha = \tau_1\}) \\ S_1\,\Gamma \vdash e_2 : (\tau_2, S_2) & S = S' \circ S_2 \circ S_1 \quad \alpha \text{ fresh}\end{array}}{\Gamma \vdash e_1\ e_2 : (S\,\alpha, S)}\ (\text{APP}) \qquad \frac{\begin{array}{ll}\Gamma \vdash e_1 : (\tau_1, S_1) & \overline{\alpha} = ftv(\tau_1) - ftv(S_1\,\Gamma) \\ S_1\,\Gamma, x : \forall\, \overline{\alpha}.\, \tau_1 \vdash e_2 : (\tau_2, S_2) & S = S_2 \circ S_1\end{array}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : (S\,\tau_2, S)}\ (\text{LET})$$

$$\frac{\begin{array}{c}\text{for } i = 1, \ldots, m \quad \Gamma \vdash_{\text{pat}} p_i \to e_i : \tau_i \to \tau_i' \\ \rho \to \rho' = lcg(\{\tau_i \to \tau_i'\}^{i=1..m}) \\ \overline{\alpha} = gtv(\rho) \quad \overline{k} \text{ are fresh Skolem constants} \\ \overline{\beta} = \bigcup^{i=1..m} gtv(\tau_i \to \tau_i') \quad \overline{\beta^r} \text{ are fresh rigid type variables} \\ S = unifyall([\overline{\alpha \mapsto k}](\rho \to \rho'), [\overline{\beta \mapsto \beta^r}]\{\tau_i \to \tau_i'\}^{i=1..m})\end{array}}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \overline{p \to e} : (S(\rho \to \rho'),\ S)}\ (\text{CASE})$$

$$\boxed{\Gamma \vdash_{\text{pat}} \overline{p \to e} : \tau \to \tau'}$$

$$\frac{\Gamma \vdash K : \forall \overline{a}.\tau_1 \to \ldots \to \tau_p \to T\ \overline{a} \quad \overline{\alpha} \text{ fresh} \quad S = [\overline{a \mapsto \alpha}] \quad \Gamma \cup S\{x_1 : \tau_1, \ldots, x_p : \tau_p\} \vdash e : (\tau_e, S_e)}{\Gamma \vdash_{\text{pat}} K\ x_1 \ldots x_p \to e : (T\ \overline{\alpha} \to \tau_e, S_e)}\ (\text{PAT})$$

$$\frac{\begin{array}{c}\text{for } i = 1, \ldots, n \quad \Gamma \vdash K_i : \forall \overline{a}_i.\tau_{i1} \to \ldots \to \tau_{ip_i} \quad \overline{\alpha}_i \text{ fresh} \quad S_i = [\overline{a \mapsto \alpha}]_i \\ S = S_1 \circ \ldots \circ S_n \quad \Gamma \cup \bigcup^{i=1..n}\{x_{i1} : \tau_{i1}, \ldots, x_{ip_i} : \tau_{ip_i}\} \vdash e : (\tau_e, S_e)\end{array}}{\Gamma \vdash (K_1 : x_{11} \ldots x_{1p_1}, \ldots, K_n : x_{n1} \ldots x_{np_n}) \to e : ((T_1\ \overline{\alpha}_1, \ldots, T_n\ \overline{\alpha}_n) \to \tau_e, S_e)}\ (\text{TPAT})$$

Figure 3. Type inference

variable c is not a parameter of a GADT type constructor, we try to unify types *Int* and *Bool*, which fails, causing the definition of g to be rejected. However, type $Erk\ a\ a \to a$ would be a valid type for g.

Example 3. When a type variable occurs associated with the GADT in some alternative, this variable is considered rigid. If it is necessary to unify this variable the function must be rejected. This occurs in cases where generalization is not able to determine this association in all alternatives, as in example:

```
data R a where
     R1 :: Int → R Int
     R2 :: a   → R a

testR (R1 x) y = if y then x else x + 1
testR (R2 x) y = if x == y then x else y
```

The types inferred for alternatives are, respectively, $R\ Int \to Bool \to Int$ and $R\ a \to a \to a$. The least common generalization of these types is: $R\ b \to c \to b$. The generalization is not able to capture the association of type variable c with the GADT, therefore, we try to unify c with *Bool* and a, but unification fails because a is substituted by a rigid type variable, and function *testR* is rejected.

This rigidity is necessary because, in the second alternative, the type of the second parameter must be the same that occurs in the GADT, and inference of a type that violates this relation can cause a runtime error. For example, $R\ a \to Bool \to a$ is not a valid type for *testR* since expression *testR* (*R2* 1) *True* would cause an runtime error.

3.1 Type inference for alternatives

The main ideas of our type inference algorithm are formalized by rule (CASE) presented in Figure 3. Type inference for expressions and patterns are standard.

Function *unifyall*, used in rule (CASE) for unifying the types of alternatives, is defined below. It uses a modified version of usual unification in order to take into account for Skolem constants (k) and rigid type variables (α^r):

$$
\begin{array}{lcl}
\mathit{unifyall}(\tau, \emptyset) & = & \mathit{id} \\
\mathit{unifyall}(\tau, \{\tau'\} \cup \{\overline{\tau}\}) & = & \mathit{unifyall}(S\tau, \{\overline{\tau}\}) \circ S \\
\quad \texttt{where } S = \mathit{unify}(\tau, \tau') & &
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{unify}(\alpha^r, \tau) & = & \texttt{if } \tau == \alpha^r \texttt{ then } \mathit{id} \texttt{ else } \mathit{fail} \\
\mathit{unify}(k, \tau) & = & \mathit{id} \\
\mathit{unify}(\alpha, \tau) & = & \texttt{if } \alpha \notin \mathit{ftv}(\tau) \texttt{ then } [\alpha \mapsto \tau] \\
 & & \quad \texttt{else } \mathit{fail} \\
\mathit{unify}(\tau_1, \tau_2) & = & \mathit{unify}(\tau_2, \tau_1) \texttt{ if } \tau_2 \text{ is a variable} \\
 & & \quad \text{or a Skolem constant} \\
\mathit{unify}(C\tau_1..\tau_n, C'\tau_1'..\tau_m') & = & \texttt{if } n == m \text{ and } C = C' \\
 & & \quad \texttt{then } S_n \circ \ldots \circ S_1 \\
\quad \texttt{where} & S_0 = \mathit{id} & \\
 & S_i = \mathit{unify}(S_{i-1}(\tau_i), S_{i-1}(\tau_i')) \text{ for } i = 1..n &
\end{array}
$$

Functions defined over GADTs often involves polymorphic recursion, Gelain *et al.* [7] proposes to handle polymorphic recursion in a way that is similar to type inference for overloading in system CT [1]. But, for simplicity, that problem was not addressed here.

4 Modified OutsideIn

Since our solution is mostly focused on the (CASE) rule, we are able to reuse a big portion of the OutsideIn algorithm [17]. The process[4] we propose is applied in addition to the OutsideIn constraint generator for case expressions, which would normally return a (τ, F) tuple. Since our algorithm returns a substitution S, the modified OutsideIn version returns a (τ, F, S) tuple.

Rules (VAR) and (CONS) had the *id* substitution added to their returns. Rules (APP) and (ABS) are changed only to propagate the substitution returned, composing when needed. Rule (LETA) is important, since our algorithm does nothing about type signatures. When an annotated `let` expression occurs, all substitutions returned are discarded, and the original OutsideIn constraint generation is done.

The most significant modification compared to Outside is in Rule (LET). When an unannotated `let` expression is found, the implication constraints generated are discarded, since they are only needed to check annotations. In this situation, the GADT inference done by our algorithm is finally applied to constraints and types. The remaining process is preserved.

In order to comport both methods, rule (CASE) does, independently, both the constraint generation for alternatives, from OutsideIn, and the procedure described on 3.1, which is abstracted by the so-called Anti-unification Solver (see Figure 6). On (CASE), the information returned by (PAT) is combined to form constraints (from OutsideIn) and groups of form $(\tau \rightarrow \alpha, F, S)$, that are going to be the anti-unification solver parameter.

[4] A Haskell implementation of this algorithm is available at github.com/GabrielaMoreira/GADTInference/

4.1 Anti-unification Solver

The purpose of the anti-unification solver is to infer a type for the alternatives of a case expression. This is done with the procedure described in Section 3. If one looks at the solver's formalization (Figure 6) and rule (CASE) from Figure 3, two differences can be noticed: on the first and the last step.

Such differences happen because the substitution returned by the anti-unification solver must contain bindings for the original type variables τ_i and α, from the modified OutsideIn (CASE). This is essential once said substitution will only be applied to anything when some unannotated `let` expression occurs.

Given $(\tau_i \rightarrow \alpha, F_i, S_i)$, the information about types obtained so far is gathered in both S_i and F_i. Therefore, the first step is to use that information to specialize $\tau_i \rightarrow \alpha$, obtaining the actual types inferred for each alternative. This is done by applying the substitution S_i and calling the constraint solver for simple constraints in F_i that are filtered by function *simple* (see Figure 5). Next, the *least common generalization* is found, GADT associated type variables are skolemized and *unifyall* is called for the resulted type and the list of specialized types obtained on first step.

The last step is necessary because the substitution returned by the first application of *unifyall* binds variables from specialized types. In order to get the bindings for the original types, another *unifyall* is called, unifying the actual type inferred $(S(\rho \rightarrow \rho'))$ with each one of them.

4.2 Solving stage

Rule (INFER) describes how the information gathered in constraints and substitutions is combined to actually infer a type. Given (τ, F, S), S is applied to F, since it can make unsolvable constrains solvable. Then, constraint solving is done for $S(F)$ exactly like on OustsideIn (Figure 7), and its result - a substitution - is composed with S finally applied to τ.

Example 4. Consider the expression from Section 1:

```
data H a where
  H1 :: Bool → H Bool
  H2 :: Int  → H Int

λx. case x of
      H1 n → n
      H2 n → n
```

Although this is somewhat simple, and does have a principal type, OutsideIn is not able to infer it. Once the expression is not annotated, the modified version will use the substitution returned and, for this example, infer the type. To understand how that substitution is found and applied, let's follow the algorithm from Figure 4.

The tuples returned, by the rule (PAT), for the first and second alternatives are: $(H\ \alpha_1 \rightarrow \alpha_1, [\alpha_1], \alpha_1 \sim \mathit{Bool}, \epsilon, \mathit{id})$

$$\boxed{\Gamma \vdash_{\mathtt{inf}} e : \tau}$$

$$\frac{\vdash e : (\tau, F, S) \qquad \vdash_s S(F) : S_s}{\vdash_{\mathtt{inf}} e : S_s \circ S\,\tau} \text{ (INFER)}$$

$$\boxed{\Gamma \vdash e : (\tau, F, S)}$$

$$\frac{\forall\, \overline{a}.\, \tau' \in \Gamma \quad \tau = [\overline{a} \mapsto \overline{\alpha}]\tau' \quad \overline{\alpha} \text{ fresh}}{\Gamma \vdash x : (\tau, \epsilon, id)} \text{ (VAR)} \qquad \frac{K : \forall \overline{a}.C \Rightarrow \tau \quad S = [\overline{a} \mapsto \overline{\alpha}]\tau \quad \alpha \text{ fresh}}{\Gamma \vdash K : (S\,\tau, S\,C, id)} \text{ (CONS)}$$

$$\frac{\Gamma \vdash e_1 : (\tau, F_1, s_1) \quad \Gamma \vdash e_2 : (\tau_2, F_2, S_2) \quad \alpha \text{ fresh} \quad F = F_1 \wedge F_2 \wedge (\tau_1 \rightarrow \tau_2 \sim \alpha)}{\Gamma \vdash e_1\ e_2 : (\alpha, F, S_1 \circ S_2)} \text{ (APP)}$$

$$\frac{\Gamma \vdash e_1 : (\tau, F_1, S_1) \quad \vdash_s simple(S_1\,F_1) : S_s \quad S = S_1 \circ S_s \quad \overline{b} = ftv(S\,\tau) - ftv(S\,\Gamma) \quad \overline{\beta} \text{ fresh} \quad S_k = [\overline{b} \mapsto \overline{\beta}] \quad \Gamma \cup \{g : \forall \overline{\beta}.S_k\,S\,\tau\} \vdash e_2 : (\tau', F_2, S_2)}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : (S\,\tau', F_2, S_2)} \text{ (LET)}$$

$$\frac{\Gamma, x : \alpha \vdash e : (\tau, F, S) \quad \alpha \text{ fresh}}{\Gamma \vdash \lambda x.e : (\alpha \rightarrow \tau, F, S)} \text{ (ABS)}$$

$$\frac{\Gamma \cup \{g : \forall \overline{a}.\tau\} \vdash e_1 : (\tau', F_1, S_1) \quad \Gamma \cup \{g : \forall \overline{a}.\tau\} \vdash e_2 : (\rho, F_2, S_2) \quad F = F_2 \wedge [ftv(\Gamma)](\forall \overline{a}.F_1) \wedge \tau \sim \tau')}{\Gamma \vdash \mathtt{let}\ \{g :: \forall \overline{a}.\tau = e_1\}\ \mathtt{in}\ e_2 : (\rho, F, S_2)} \text{ (LETA)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : (\tau_e, F_e, S_e) \quad \alpha \text{ fresh} \\ \Gamma \vdash_{\mathtt{pat}} p_i \rightarrow e_i : (\tau_i \rightarrow \rho_i, \overline{\alpha_i}, D_i, F_i, S_i) \quad F_i' = F_i \wedge \tau_e \sim \tau_i \wedge [\overline{\alpha}_i]D_i \supset \alpha \sim \rho_i \quad \text{for } i = 1, \ldots, n \\ F = F_e \wedge \bigwedge^{i=1..n} F_i' \quad \vdash_a [(S_i, \tau_i \rightarrow \alpha, F_i \wedge D_i \wedge \alpha \sim \rho_i)]^{i=1..n} : S_a \quad S = S_e \circ S_a \circ S_1 \circ \ldots \circ S_n\end{array}}{\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ [p_i \rightarrow e_i]^{i=1..n} : (\alpha, F, S)} \text{ (CASE)}$$

$$\boxed{\Gamma \vdash_{\mathtt{pat}} \overline{p \rightarrow e} : \tau \rightarrow \tau'}$$

$$\frac{K : \forall \alpha.\overline{b}.D \Rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_p \rightarrow T\ \overline{a} \quad \overline{b} \notin ftv(\Gamma, \tau_e) \quad \overline{\alpha} \text{ fresh} \quad S = [\overline{a \mapsto \alpha}] \quad \Gamma \cup S\{x_1 : \tau_1, \ldots, x_p : \tau_p\} \vdash e : (\tau_e, F_e, S_e)}{\Gamma \vdash_{\mathtt{pat}} K\ x_1 \ldots x_p \rightarrow e : (T\ \overline{\alpha} \rightarrow \tau_e, [\overline{\alpha} \cup ftv(\Gamma, \tau_e)], \forall \overline{b}.S\,D, F_e, S_e)} \text{ (PAT)}$$

$$\frac{\begin{array}{c}\text{for } i = 1, \ldots, n \quad \Gamma \vdash K_i : \forall \overline{a}_i.\overline{b}_i.D_i \Rightarrow \tau_{i1} \rightarrow \ldots \rightarrow \tau_{ip_i} \quad \overline{b} \notin ftv(\Gamma, \tau_e) \quad \overline{\alpha}_i \text{ fresh} \quad S_i = [\overline{a \mapsto \alpha}]_i \\ \overline{\alpha} = \bigcup^{1=1..n} \overline{\alpha}_i \quad D = \bigwedge^{1=1..n} D_i \quad S = S_1 \circ \ldots \circ S_n \quad \Gamma \cup \bigcup^{i=1..n} \{x_{i1} : \tau_{i1}, \ldots, x_{ip_i} : \tau_{ip_i}\} \vdash e : (\tau_e, F_e, S_e)\end{array}}{\Gamma \vdash (K_1 : x_{11} \ldots x_{1p_1}, \ldots, K_n : x_{n1} \ldots x_{np_n}) \rightarrow e : ((T_1\ \overline{\alpha}_1, \ldots, T_n\ \overline{\alpha}_n) \rightarrow \tau_e, [\overline{\alpha} \cup ftv(\Gamma, \tau_e)], \forall \overline{b}.S\,D, F_e, S_e)} \text{ (TPAT)}$$

Figure 4. Modified OutsideIn

$$\begin{array}{rcl} simple(C) & = & C \\ simple(F_1 \wedge F_2) & = & simple(F_1) \wedge simple(F_2) \\ simple([\overline{\alpha}](\forall b.F)) & = & [\overline{\alpha}](\forall b.simple(F)) \\ simple([\overline{\alpha}](\forall b.C \supset F)) & = & \epsilon \qquad C \not\equiv \epsilon \end{array}$$

Figure 5. Filter for simple constraints

and $(H\ \alpha_2 \rightarrow \alpha_2, [\alpha_2], \alpha_2 \sim Int, \epsilon, id)$. Since this is a `case` expression, the anti-unification solver will be invoked, solving constraints $\alpha_1 \sim Bool$ and $\alpha_2 \sim Int$, the types of alternatives are: $H\ Bool \rightarrow Bool$ and $H\ Int \rightarrow Int$. The *least common generalization* of the alternatives types will be found as: $H\ b \rightarrow b$.

Note that type variable b will be *skolemized*, once it is associated with a GADT. Unifying this with the alternatives types $H\ Bool \rightarrow Bool$ and $H\ Int \rightarrow Int$ will give us id, since b cannot be bound. Next, the *skolemization* is removed and id is applied, resulting still on $H\ b \rightarrow b$. This will be unified with $H\ \alpha_1 \rightarrow \alpha$ and $H\ \alpha_2 \rightarrow \alpha$, generating the substitution $S = [b \mapsto \alpha_1, \alpha_2 \mapsto \alpha_1, \alpha \mapsto \alpha_1]$.

$$\boxed{\vdash_a [(S_i, \tau_i \to \alpha, F_i)]^{i=1..n} : S_a}$$

$$\frac{\begin{array}{c}\text{for } i = 1, \ldots, n \quad \vdash_s simple(F_i) : S_i' \\ \rho \to \rho' = lcg(\{S_i' S_i\, \tau_i \to \alpha\}^{i=1..n}) \\ \overline{\gamma} = gtv(\tau) \quad \overline{k} \text{ are fresh Skolem constants} \\ \overline{\beta} = \bigcup^{i=1..n} gtv(S_i' S_i\, \tau_i \to \alpha) \\ \overline{\beta^r} \text{ are fresh rigid type variables} \\ S = unifyall([\overline{\gamma \mapsto K}](\rho \to \rho'), \\ \quad [\overline{\beta \mapsto \beta^r}]\{S_i' S_i\, \tau_i \to \alpha\}^{i=1..n}) \\ S_a = unifyall(S\,\rho \to \rho', \{\tau_i \to \alpha\}^{i=1..n})\end{array}}{\vdash_a [(S_i, \tau_i \to \alpha, F_i)]^{i=1..n} : S_a}$$

Figure 6. Anti-unification Solver

$$\boxed{\vdash_s F : S}$$

$$\frac{simple(F) = F_s \qquad \vdash_s^* F_s : S_s \qquad \vdash_s^* S_s F : S}{\vdash_s F : S \circ S_s} \text{ (S-SOLVE)}$$

$$\boxed{\vdash_s^* F : S}$$

$$\frac{}{\vdash_s^* \epsilon : id} \text{ (S-EMPTY)} \qquad \frac{\vdash_s^* F_1 : S_1 \quad \vdash_s^* S_1 F_2 : S_2}{\vdash_s^* F_1 \wedge F_2 : S_2 \circ S_1} \text{ (S-SPLIT)}$$

$$\frac{}{\vdash_s^* \tau \sim \tau : id} \text{ (S-REFL)} \qquad \frac{\vdash_s^* (\bigwedge_i \tau_i \sim \tau_i') : S}{\vdash_s^* T\overline{\tau} \sim T\overline{\tau}' : S} \text{ (S-CONS)}$$

$$\frac{v \notin \tau \quad S = [v \mapsto \tau]}{\vdash_s^* v \sim \tau : S} \text{ (S-UL)} \qquad \frac{v \notin \tau \quad S = [v \mapsto \tau]}{\vdash_s^* \tau \sim v : S} \text{ (S-UR)}$$

$$\frac{\vdash_s^* F : S \quad tv(S(\overline{\alpha})) \cap \overline{b} = \emptyset \quad \overline{b} \cap dom(S) = \emptyset}{\vdash_s^* [\overline{\alpha}](\forall b.F) : S} \text{ (S-SIMPL)}$$

$$\frac{\begin{array}{c}C \not\equiv \epsilon \quad \vdash_s^* C : S_s \\ \vdash_s S_s(F) : S \quad \overline{\alpha} \cap dom(S) = \emptyset\end{array}}{\vdash_s^* [\overline{\alpha}](\forall b.C \supset F) : S} \text{ (S-SIMPL)}$$

Figure 7. Constraint Solver

Since this is not a `let` expression, the solving will be done only at (INFER). However, the substitution returned by the anti-unification solver is enough to infer the type. Therefore, it's application on the constraints will make them all solvable, turning implication constraints into simple ones. Solving them will return the remaining bindings, which will be composed with S and finally applied to the fresh original type to return the final type inferred: $\forall \alpha.H\ \alpha \to \alpha$.

In some cases where our approach is not able to capture the relationship between the types and the GADT, the type could still be inferred by OutsideIn. In order to allow this, some condition should be added to verify if the anti-unification solver has failed, and if so, make it return the *id* substitution. This way, the original constraints will be passed to the constraint solver and type inference will be done entirely by OutsideIn. Such verification is not described in the algorithm formalization for clarity purposes.

5 Related Work

Haskell was one of the very first programming languages supporting GADTs. Woobly types [9, 10] describes one of the first implementations of GADTs in GHC, and most of type checking of GADT functions is done using type annotations. These types, called rigid types, are propagated to inner scopes by means of some specific rules. Such propagation strategies are also explored by Pottier and Régis-Gianas [16] which 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.

Currently, GHC uses the type inference algorithm described in [17, 22], called OutsideIn, which extracts type constraints from expressions occurring in inner scopes and solves these constraints in the outermost scope, avoiding an *ad hoc* approach for propagating 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 [21] Sulzmann and Schrijvers introduce some ideas adopted in the OutsideIn algorithm.

Lin and Sheard present the Pointwise GADT type system [12], that uses a modified unification algorithm to support parametric instantiation and type indexing. In [11] 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 using an approach similar to [13] and places an iteration limit to guarantee termination.

Chore uses choice types in order to describe the type of a function [3]. A set of choices with the same name represents a set of types of each alternative of an expression. For example, $f\langle H\ Int, H\ Bool\rangle \to f\langle Int, Bool\rangle$ can represent the type of f, presented in introduction. New sets of choices are added when new scrutinee type need to be represented, like functions with nested cases.

In Chore[3], choice types are propagated during all the inference process, the reconciled type is generated just to communicate with users using an algorithm (named reconcile)

on the MAIN rule. But, as choices do not have an order, an additional algorithm (named coherent) is needed to ensure that functions with types like $f\langle H\ Int, H\ Bool\rangle \rightarrow f\langle Bool, Int\rangle$ could not be accepted, the verification is made on the CASE rule.

Besides that, rule APP has to additionally verify whether the argument type matches with one or more alternatives of the function that is called, where there are three possible cases: 1) if it matches exactly, then the function is considered well typed, 2) if it matches in some cases, then a set of possible choice types is returned, 3) if there is no match, the function is considered ill typed. This third case makes possible to reject application of functions where the alternative exists for data type, but is not implemented. Chore can capture more errors compared to other approaches (including type inference used by GHC) due to rule APP, but it is not able to infer the type of GADT functions with polymorphic recursion.

Guarrigue et. al. [6] presents a type system and its inference algorithm for a language with GADTs with returns a principal type, when one exists. The main insight of [6] is to use a refined notion of type ambiguity to reduce the amount of type annotations needed. In order to avoid a aggressive type propagation mechanism, they introduce the notion of *ambivalent types*, which denote set of simple type expressions containing certain type equivalences within the type structure. According to the authors, their proposed inference algorithm follows the ideas developed by [5] for first-class polymorphism.

6 Conclusion

The OutsideIn algorithm is quite efficient in type checking of GADT expressions. However, the set of expressions that it is able to infer the type is restricted. It (and, consequently, GHC) is not able to infer a type for unannotated functions presented in Examples 1, 2 and 4. This paper adds an anti-unification-based type inference method to the OutsideIn algorithm. The algorithm presented is able to infer a type for Examples 1 and 4 preserving the capacity to verify annotated functions given by the constraints approach in OutsideIn.

Every GADT inference algorithm has difficulty to explain to the programmer which expressions can be accepted without type annotation. We believe that our approach is more intuitive. But it is necessary, like the OutsideIn, to determine in which cases it is possible to guarantee the principal type property. For this, we still must provide a declarative description that specifies the set of programs that are well typed according to our type inference.

We have focused exclusively on GADTs, GHC supports others advanced type system features, such as type classes and type families [22]. A future work is to adapt our approach to address these features.

References

[1] Carlos Camarão, Lucília Figueiredo, and Cristiano Vasconcellos. 2004. Constraint-set Satisfiability for Overloading. In *Proc. of the 6th ACM SIGPLAN International Conf. on Principles and Practice of Declarative Prog.* ACM, 67–77.

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

[3] Sheng Chen and Martin Erwig. 2016. Principal Type Inference for GADTs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* ACM.

[4] James Cheney and Ralf Hinze. 2003. *First-class phantom types.* Technical Report CUCIS TR-2003-1901. Cornell University.

[5] Jacques Garrigue and Didier Rémy. 1999. Semi-explicit First-class Polymorphism for ML. *Inf. Comput.* 155, 1-2 (Nov. 1999), 134–169. https://doi.org/10.1006/inco.1999.2830

[6] Jacques Garrigue and Didier Rémy. 2013. Ambivalent Types for Principal Type Inference with GADTs. In *Programming Languages and Systems*, Chung-chieh Shan (Ed.). Springer International Publishing, Cham, 257–272.

[7] Adelaine Gelain, Cristiano Vasconcellos, Carlos Camarão, and Rodrigo Ribeiro. 2015. Type Inference for GADTs and Anti-unification. In *Proceedings of the 19th Brazilian Symposium on Programming Languages - Volume 9325.* Springer-Verlag New York, Inc., New York, NY, USA, 16–30.

[8] Ralf Hinze. 2003. Fun with phantom types. In *The fun of programming*, Jeremey Gibbons and Oege de Moor (Eds.). Palgrave, 245–262.

[9] Simon P. Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-based Type Inference for GADTs. *SIGPLAN Not.* 41, 9 (2006), 50–61.

[10] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. 2004. *Wobbly Types: Type Inference for Generalised Algebraic Data Types.* Technical Report MS-CIS-05-26. University of Pennsylvania. http://research.microsoft.com/apps/pubs/default.aspx?id=65143 Microsoft Research.

[11] Chuan-Kai Lin. 2010. *Practical Type Inference for the GADT Type System.* Ph.D. Dissertation. Portland, OR, USA. Advisor(s) Tim Sheard.

[12] Chuan-Kai Lin and Tim Sheard. 2010. Pointwise Generalized Algebraic Data Types. In *Proc. of the 5th ACM SIGPLAN Workshop on Types in Lang. Design and Implementation (TLDI '10).* ACM, New York, NY, USA, 51–62.

[13] Alan Mycroft. 1984. Polymorphic Type Schemes and Recursive Definitions. In *Proc. of the 6th Colloquium on International Symp. on Prog.* Springer-Verlag, 217–228.

[14] Gordon D Plotkin. 1970. A note on inductive generalisation. *Machine intelligence* 5, 1 (1970), 153–163.

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

[16] François Pottier and Yann Régis-Gianas. 2006. Stratified Type Inference for Generalized Algebraic Data Types. *SIGPLAN Not.* 41, 1 (2006), 232–244.

[17] Tom Schrijvers, Simon P. Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and Decidable Type Inference for GADTs. *SIGPLAN Not.* 44, 9 (2009), 341–352.

[18] Tim Sheard. 2004. Languages of the Future. *SIGPLAN Notices* 39, 12 (2004), 119–132.

[19] Tim Sheard. 2005. Putting Curry-Howard to work. In *Proceedings of ACM Workshop on Haskell.* 74–85.

[20] Peter Stuckey and Martin Sulzmann. 2002. A Theory of Overloading. In *Proc. of the 7th ACM International Conf. on Func. Prog.* 167–178.

[21] Martin Sulzmann, Tom Schrijvers, and Peter J. Stuckey. 2008. Type Inference for GADTs via Herbrand Constraint Abduction. (2008).

[22] Dimitrios Vytiniotis, Simon P. Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4-5 (2011), 333–412.