Type inference for GADTs via Herbrand constraint abduction

Martin Sulzmann *Tom Schrijvers*
Peter J. Stuckey

Report CW 507, January 2008



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Type inference for GADTs via Herbrand constraint abduction

Martin Sulzmann *Tom Schrijvers*
Peter J. Stuckey

Report CW 507, January 2008

Department of Computer Science, K.U.Leuven

Abstract

Type inference for Hindley/Milner and variants is well understood as a constraint solving problem. Recent extensions to Hindley/Milner such as generalized algebraic data types (GADTs) force us to go beyond this approach to inference.

In this paper we show how to perform type inference for GADTs using Herbrand constraint abduction, a solving method to infer missing facts in terms of Herbrand constraints, i.e. conjunctions of type equations. But typeinference for GADTs is very hard, we are the first to give example programs with an infinite number of maximal types. We propose to rule out several kinds of "non-intuitive" solutions and show that we can construct in this way a complete and decidable type inference approach for GADTs and sketch how to support type error diagnosis. Our results point out new direction how to perform type inference for expressive type systems.

Keywords : type inference, generalized algebraic data types, abduction, Herbrand constraints
CR Subject Classification : D.3.2 Language Classifications (Applicative languages, Constraint and logic languages), D.3.4 Processors (Compilers).

Type Inference for GADTs via Herbrand Constraint Abduction

Martin Sulzmann
School of Computing
National University of Singapore
S16 Level 5, 3 Science Drive 2
Singapore 117543
sulzmann@comp.nus.edu.sg

Tom Schrijvers *
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
tom.schrijvers@cs.kuleuven.be

Peter J. Stuckey
NICTA Victoria Laboratory
Department of Comp. Sci and Soft. Eng.
The University of Melbourne
Vic. 3010, Australia
pjs@cs.mu.oz.au

Abstract

Type inference for Hindley/Milner and variants is well understood as a constraint solving problem. Recent extensions to Hindley/Milner such as generalized algebraic data types (GADTs) force us to go beyond this approach to inference.

In this paper we show how to perform type inference for GADTs using Herbrand constraint abduction, a solving method to infer missing facts in terms of Herbrand constraints, i.e. conjunctions of type equations. But type inference for GADTs is very hard, we are the first to give example programs with an infinite number of maximal types. We propose to rule out several kinds of "non-intuitive" solutions and show that we can construct in this way a complete and decidable type inference approach for GADTs and sketch how to support type error diagnosis. Our results point out new direction how to perform type inference for expressive type systems.

1. Introduction

Currently there is substantial interest in an extension of the Hindley/Milner (HM) type system known as *generalized algebraic data types* (GADTs) [2, 25].[1]. GADTs generalize the well-known algebraic data types [8] from Haskell and ML by supporting a form of type refinement connected to case expressions. This makes it possible for the programmer to impose more expressive forms of types on programs [1, 17], something which used to be the exclusive domain of specialized systems such as dependent [4] and index types [27, 26].

Type inference for GADTs is supposed to be difficult. For the first time, we show GADT programs that can be given an infinite set of maximal types. Hence, we establish that GADT type inference is incomplete and undecidable in general. We also show that principal types for GADTs (if they exist) are not necessarily "compositional". From the HM type system we are used to the fact that to infer the principal type of an expression, it is sufficient to infer the principal types of its sub-expressions. GADTs do not enjoy this property.

Previous works on GADT inference can be divided into two categories. Simonet and Pottier [19] develop a complete GADT inference system by enlarging the set of expressible types at the expense of readability of inferred types and incurring a non-elementary complexity of the inference algorithm. Work by Pottier/Régis-Gianas [16] and Peyton Jones et. al. [15] demand user input in form of type annotations to recover principal and decidable type inference via standard Hindley/Milner techniques.

In our approach, we attempt to find a balance between these two categories. Our method is less general than [19] without rejecting sensible programs. The advantage of our approach is that we infer familiar Hindley/Milner types and we are more general than [16, 15] (i.e. programs require fewer type annotations). Our method is completely predictable. That is, we can precisely characterize the set of programs for which our inference method succeeds and we do not rely on heuristical methods such as [16]. In addition, our method has also the potential to support type error diagnosis.

The key method behind our inference approach is *abduction*. The term abduction was coined by the philosopher Pierce [13] whose aim was to provide a theory about human reasoning, particularly concerned with scientific discovery. He identified abduction, the ability to reason from observed results to the basic missing facts from which they follow, as one of three fundamental principles. The other two principles are deduction and induction. In artificial intelligence, abduction has been embraced to overcome limitations of deductive reasoning. This has also led to extensive research in the area of abductive logic programming, e.g. see [3] for an overview.

To the best of our knowledge, we are the first to propose GADT type inference via *Herbrand constraint* abduction, i.e. the basic missing facts are represented via Herbrand constraints. In [10], Maher introduces Herbrand constraint abduction.[2] Applying his results to the GADT type inference setting is an entirely non-trivial task.

In summary, our contributions are:

- We provide concrete examples that explain why type inference for GADTs in general is such a hard problem (Section 2.1).
- We introduce a novel GADT inference method based on Herbrand constraint abduction (Section 2.2).
- To obtain a GADT type system specification which enjoys complete and decidable type inference, we restrict the set of typable programs by ruling out certain non-intuitive types (Section 5).
- We can verify that the intuitive GADT system (Section 5.3) enjoys complete and decidable type inference, is a conservative extension of Hindley/Milner, is stable (i.e. program remain intuitive if we annotate them with correct type annotations), and retains principal types under some sensible conditions.
- We explore applications of Herbrand constraint abduction to type error diagnosis (Section 2.4).

* Research Assistant of the fund for Scientifi c Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

[1] GADTs are also known as *guarded recursive data types* or *first-class phantom types*.

[2] We would like to point out that Maher [10] cites a previous draft of this paper for why he believes his work has connections to type inference.

2. The GADT Type Inference Problem

First, we give two example programs that precisely explain why type inference for GADTs is such a hard problem. Then, we highlight the key ideas behind our GADT type inference via Herbrand constraint abduction method. We will see that our inference method improves over previous works for the particular case of non-polymorphic recursive GADT programs. Typical GADT applications [17, 25] rely on polymorphic recursion for which inference is known to be undecidable unless we provide type annotations [6]. We show that Herbrand constraint abduction is also useful in case of type inference in the presence of (potentially incorrect) type annotations.

2.1 Loss of Principality and Undecidability

Here is a GADT program which can be given an infinite set of maximal types. We say a type is *maximal* if there is no other more general type. We will use Haskell style syntax [14] in all examples. Consider the following program:

```
data Erk x y z where
  K :: x -> y -> Erk x y Char
f (K x y) = x:y
```

When pattern matching over `K x y` we may make use of the fact that `z` has type $Char$. Among others, function `f` can be given the following set of types

$$\texttt{f} \quad :: \quad \forall t_z.Erk\ [t_z]^n\ [Char]^{n+1}\ t_z \rightarrow [Char]^{n+1}$$

for any $n \geq 0$. We write $[t]^n$ to denote a list of lists of ... lists (depth n) of t's. Note it also has the type $\forall t_x, t_z.Erk\ t_x\ [t_x]\ t_z \rightarrow [t_x]$. All these types are incomparable and there is no type which is more general than any of the above types. Hence, each of the above types is maximal. Hence, GADTs do not enjoy principal types (i.e. unique maximal type). We cannot even hope for a complete and decidable inference algorithm because the number of maximal types that can be given to `f` is infinite.

The GADT type system is less well-behaved than the HM system in another, more subtle way. From Hindley/Milner we are used that a typing derivation is principal iff all its sub-derivations are principal. This is the key to obtain a complete Hindley/Milner type inference algorithm. In case of GADTs, this property is lost. Consider the following variant of the above example:

```
data Erk x y z where
  K :: x -> y -> Erk x y Char
  L :: x -> y -> Erk x y Bool
g = let f (K x y) = x:y
        f (L x y) = y
    in f (L [True] [['a']])
```

Expression `(L [True] [['a']])` has type

$$Erk\ [Bool]\ [[Char]]\ Bool$$

Function `f` can be given the type

$$\forall t_z.Erk\ [t_z]\ [[Char]]\ t_z \rightarrow [[Char]]$$

which is maximal but not principal (see above). Hence, `f (L [True] [['a']])` has type $[[Char]]$. Hence, the outermost expression `g` has the principal type $[[Char]]$. The other maximal type $\forall t_x, t_z.Erk\ t_x\ [t_x]\ t_z \rightarrow [t_x]$ of `f` under which `g` type checks leads to the same type.

2.2 Type Inference via Herbrand Constraint Abduction

The question is how to perform type inference for such expressive type systems? The standard route towards type inference is to generate an appropriate set of constraints out of the program text which are then solved by a domain-specific solver. Each solution corresponds to a valid type. In case of the HM system, the constraint domain to represent the typing problem consists of *conjunctions of type equations*. This is also known as the Herbrand constraint domain. Solving of Herbrand constraints is achieved via unification [7].

GADTs demand a significant change to this process. To capture the typing problem precisely, we need a richer set of *implication constraints*. That is, formulas of the form $D \supset C$ where $\supset$ denotes Boolean implication and D and C are Herbrand constraints. Constraints D arise from pattern matchings over GADTs and constraints C are the Hindley/Milner constraints arising from the program text. In general, we may need conjunctions of $D \supset C$ and variables in the implication constraints may be universally or existentially quantified. How to generate such implication constraints out of the program text is entirely standard [19].

In case of the first example from the previous section

```
data Erk x y z where
  K :: x -> y -> Erk x y Char
f (K x y) = x:y
```

we generate the following implication constraint out of `f`'s program text

$$t_z = Char \supset t_y = [t_x]$$

where t_x, t_y and t_z denote the types of respectively `x`, `y` and `z`. The constraint $t_z = Char$ follows from the constraint on the data constructor `K` in the definition of type `Erk x y z`. When the argument of `f` matches this data constructor, we know that this *assumption* constraint holds. The *Hindley/Milner* constraint $t_y = [t_x]$ results from the program text `x:y` by applying a standard procedure such as algorithm W [12].

The approach in [19] takes the implication constraint itself as the solution of the typing problem. Then, function `f` can be given the type

$$\forall t_x, t_y, t_z.(t_z = Char \supset t_y = [t_x]) \Rightarrow Erk\ t_x\ t_y\ t_z \rightarrow t_y$$

This approach has two problems. Types become not understandable and we yet have to check that the implication constraint is satisfiable which has non-elementary complexity [19].

In our approach, we perform type inference problem by solving implication constraints in terms of the familiar Herbrand constraint domain. For example, the constraint $t_x = [t_z]^n \wedge t_y = [Char]^{n+1}$ is clearly a solution for any $n \geq 0$. That is,

$$(t_x = [t_z]^n \wedge t_y = [Char]^{n+1}) \supset (t_z = Char \supset t_y = [t_x])$$

which is equivalent to

$$(t_x = [t_z]^n \wedge t_y = [Char]^{n+1} \wedge t_z = Char) \supset t_y = [t_x])$$

after applying the law that $C_1 \supset (C_2 \supset C_3)$ iff $C_1 \wedge C_2 \supset C_3$. The last statement obviously holds. Each solution $t_x = [t_z]^n \wedge t_y = [Char]^{n+1}$ is maximal (i.e. there is no solution that is more general) and corresponds to one of the maximal types $\forall t_z.Erk\ [t_z]^n\ [Char]^{n+1}\ t_z \rightarrow [Char]^{n+1}$ which we have seen before. Similarly, the maximal solution $t_y = [t_x]$ corresponds to the maximal type $\forall t_x, t_z.Erk\ t_x\ [t_x]\ t_z \rightarrow [t_x]$.

Let's summarize. We perform type inference for GADTs by first generating implication constraints out of the program text. Our idea is to solve these implication constraints in terms of Herbrand constraints. That is, for each implication constraint F we infer some S, which can be expressed in terms of Herbrand constraints, such that $S \supset F$ is a true statement. In such a situation, we say that S is a solution of F. The process of solving implication constraints

via Herbrand constraints is known as Herbrand constraint abduction [10, 13]. Each solution corresponds to a valid type and vice versa. The problem with GADT type inference becomes clear now. We may encounter too many maximal solutions (see above example) and therefore too many maximal types.

2.3 The Key Idea: Intuitive Solutions

Our approach is to limit the set of solutions such that type inference becomes tractable. The key idea is to accept only those solutions which are not stronger than the combined assumption and Hindley/Milner constraints. We call such solutions *intuitive*. For example, $t_y = [t_x]$ is an intuitive solution of $t_z = Char \supset t_y = [t_x]$. We find that

$$t_z = Char \wedge t_y = [t_x] \supset t_y = [t_x]$$

On the other hand, the solutions $t_x = [t_z]^n \wedge t_y = [Char]^{n+1}$ are *non-intuitive* because

$$t_z = Char \wedge t_y = [t_x] \not\supset t_x = [z]^n \wedge t_y = [Char]^{n+1}$$

Our argument is that non-intuitive solutions require stronger assumptions than intended by the programmer. Hence, we reject them. Of course, the user may annotate her program with a non-intuitive type (as long as the type is correct).

Intuitive solutions must satisfy some further *sensible* conditions. We demand that the (1) the combination of the assumption and Hindley/Milner constraints must be satisfiable and (2) the combination of a solution and the assumption must be satisfiable as well. Otherwise, the "intuitive" GADT system accepts still too many types. For example, consider the program

```
data Foo a = (a=Int) => I a
f (I x) = x && True
```

The above program text gives rise to $a = Int \supset (a = Bool \wedge t = Bool)$. We have that

$$(a = [Int]^n \wedge t = [Bool]^n) \supset (a = Int \supset (a = Bool \wedge t = Bool))$$

$$(a = Int \wedge a = Bool \wedge t = Bool) \supset (a = [Int]^n \wedge t = [Bool]^n)$$

for $n \geq 1$. The first statement shows that $a = [Int]^n \wedge t = [Bool]^n$ is a solution and the second statement shows that solutions are intuitive. These solutions are maximal. Hence, `f` has the infinite set of maximal, intuitive types $Foo\ [Int]^n \rightarrow [Bool]^n$ for $n \geq 1$. Of course, these types are non-sensical because we can never construct a value of type $Foo\ [Int]^n$. Notice that $a = Int \wedge a = Bool \wedge t = Bool$ is not satisfiable. Therefore, the above solutions are non-sensical. As pointed out in [2], rejecting such non-sensical programs is not an onerous restriction.

The important consequence is that for sensible solutions the set of maximal, intuitive solutions is finite. Hence, type inference for intuitive GADT programs is complete and decidable.

For example, consider the program.

```
data Erk a b where
   I :: Int -> Erk Int b
   B :: Bool -> Erk a Bool
f (I a) = x + 1
f (B b) = x && True
```

The program text gives rise to

$$t_f = Erk\ a\ b \rightarrow t \wedge$$
$$(a = Int \supset t = Int) \wedge (b = Bool \supset t = Bool)$$

The two implication constraints have two maximal intuitive solutions each: $t = a$ and $t = Int$ for the former and $t = b$ and $t = Bool$ for the latter. This yields four candidates for the overall intuitive solution.

$$\begin{array}{lcl} S_1 & \equiv & t = a \wedge t = b \\ S_2 & \equiv & t = a \wedge t = Bool \\ S_3 & \equiv & t = Int \wedge t = b \\ S_4 & \equiv & t = Int \wedge t = Bool \end{array}$$

Clearly, S_1 is the only maximal candidate among all combinations. Hence the principal intuitive type of the function is `f` :: $\forall t.Erk\ t\ t \rightarrow t$.

Previous GADT inference approaches [16, 15] fail here (unless we provide a type annotation). In general, our method strictly improves over these works for non-polymorphic recursive GADT programs. For polymorphic recursive GADT programs, we demand type annotations to retain decidable inference [6]. Hence, the approaches in [16, 15] and our approaches perform equally well in case of (type-annotated) polymorphic recursive GADT programs. The advantage of our method is that we can support type error diagnosis via Herbrand constraint abduction in case the type annotation and the program text contradict each other. This is what we will discuss next.

2.4 Type-Error Diagnosis

We consider two realistic GADT programs and sketch how our Herbrand constraint abduction inference method can possibly support type error diagnosis. First, we introduce a GADT declaration of a `List` data type that records the length of the list.

```
-- singleton types
data Z   -- zero
data S n -- successor

data List a n where
  Nil :: List a Z
  Cons :: a -> List a m -> List a (S m)
```

The constructors `Z` and `S` mimic natural numbers on the level of types via the singleton types approach. The GADT `List` is parameterized in an element type `a` and a parameter `n` recording the length of the list.

Here is a program that uses the length information recorded in the GADT `List`.

```
replace :: (a->a->Bool)->a->a->List a l->List a l
replace eq x y xs =
  let f :: List a m -> List a m
      f Nil = Nil
      f (Cons z xs) = if (eq z x) then Cons y (f xs)
                      else Cons z (f xs)
  in f xs
```

Function `replace` replaces all occurrences of the second by the third argument in the input list. The first argument is the equality testing function among types `a`. The type of `replace` guarantees that the length of the list remains the same. For convenience, we have defined a local function `f` to avoid passing around `eq`, `x` and `y`. Function `f` uses a lexically scoped type annotation where the element type `a` refers to `replace`'s annotation.

The above program text is type correct. For this it is crucial that in the type annotation of `replace` and `f` variable `a` refers to the same (element) type. Let's assume that the programmer makes a mistake and wrongly provides the annotation `f::List b m -> List b m` instead of `f::List a m -> List a m`. The type of `f` is now too polymorphic. The program text `eq z x` constrains $a = b$ assuming that `x` has type a and `z` has type b. However, a and b are polymorphic variables. Hence, the program does not type check.

In our inference approach, we generate the following implication constraint. For simplicity, we only consider the second function clause of `f` and include on the right-hand side of $\supset$ only the constraints arising out of `eq z x`.

$$\forall a, l. \forall b, m, m'. (m = S\ m' \supset a = b)$$

The left-hand side constraint $m = S\ m'$ arises from the pattern match `Cons z xs`. Ignoring the universal quantifiers, $a = b$ is a maximal intuitive solution. Of course, this solution is not correct because we are not allowed to equate different polymorphic variables, hence, `f`'s annotation is too polymorphic. The important insight is that (the incorrect) solution provides important clues how to fix the type error. By combining our GADT inference method with our earlier work on type error reporting for Hindley/Milner [20, 21], we can report to the user that a likely fix is to replace `b` by `a` in the annotation of `f`, because this is enforced by the program text `eq z x`.

In the above example, we had to fix the type annotation. In our next example, the program text is the cause of the error. We refine the type of the well-known `append` function. Our goal is to express the property that appending two lists yields a list whose length is the sum of the two input lists. We first introduce a GADT `Sum` to represent (type) summation.

```
data Sum l m n where
  Base :: Sum Z n n
  Step :: Sum l m n -> Sum (S l) m (S n)
```

Effectively, a value of type `Sum l m n` represents a proof that `n` is the sum of `l` and `m`. The `append` function therefore takes an additional argument of this type to guarantee the above property.

```
append :: Sum l m n -> List a l -> List a m -> List a n
append Base Nil ys = Nil
append (Step p) (Cons x xs) ys = Cons x (append p xs ys)
```

Notice that the program text of the first clause contradicts the type annotation. We wrongly return `Nil` where it should be `ys`. Hence, the program does not type check.

Let's take a look at the implication constraint arising from the above program. As before, we focus only on the interesting program parts.

$$\forall l, m, n, a. (m = n \wedge l = Z \supset n = Z)$$

Constraints $m = n$ and $l = Z$ arise from the pattern match over `Base` and `Nil` and $n = Z$ arises from returning `Nil`. Ignoring the universal quantifiers, we find that $m = Z$ is a maximal, intuitive solution. This solution is obviously incorrect once we include the universal quantifiers.

In this example, we are not interested in fixing the type annotation. Replacing `m` by `Z` in `append`'s annotation allows to type check the first clause but leads to a type check failure for the second clause. The point here is that the (incorrect) solution $m = Z$ tells us that the program will type check if the second input list is empty. Hence, we could insert a run-time test which provides "evidence" for $m = Z$.

```
append :: Sum l m n -> List a l -> List a m -> List a n
append Base Nil ys = case ys of
                       Nil -> Nil
                       _   -> error "run-time error"
append (Step p) (Cons x xs) ys = Cons x (append p xs ys)
```

The program type checks now.

We conclude that Herbrand constraint abduction has the potential to propose ways how to fix a type error in case type annotation and program text contradict each other.

3. The GADT System

We define the formal underpinnings of the GADT type system. First, we spell out some basic assumptions which are used throughout the rest of the paper. Next, we describe the syntax of programs. Then, we define the GADT typing rules. In our formulation, we support constraints explicitly, instead of representing them via substitutions. We also support lexically scoped type annotations, e.g. see the `replace` function from Section 2.4 where the nested function `f` uses a lexically scoped type annotation. There are really no surprises here. Similar forms of lexically scoped annotations can be found elsewhere [16, 15]. Our constraint-based presentation of the typing rules is based on [22]. We omit a description of the semantic meaning and its type soundness proof which is a standard exercise by now [19, 25].

3.1 Preliminaries

We write $\bar{o}$ to denote a sequence of objects o_1,...,o_n. As it is common, we write Γ to denote an environment consisting of a sequence of type assignments $x_1 : \sigma_1, \ldots, x_n : \sigma_n$. Types σ will be defined later. We commonly treat Γ as a set. We write "$-$" to denote set subtraction. We write $\mathit{fv}(o)$ to denote the set of free variables in some object o with the exception that $\mathit{fv}(\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\})$ denotes $\mathit{fv}(\sigma_1, \ldots, \sigma_n)$. In case objects have binders, e.g. $\forall a$, we assume that $\mathit{fv}(\forall a.o) = \mathit{fv}([b/a]o) - \{b\}$ where b is a fresh variable and $[b/a]$ a renaming.

We generally assume that the reader is familiar with the concepts of substitutions, unifiers, most general unifiers (m.g.u.) etc [7] and first-order logic [18]. We write $[\overline{t/a}]$ to denote the simultaneous substitution of variables a_i by types t_i for $i = 1, .., n$. We use common notation for Boolean conjunction ($\wedge$), implication ($\supset$) and universal ($\forall$) and existential quantifiers ($\exists$). Sometimes, we abbreviate $\wedge$ by "," and use set notation for conjunction of formulae. We sometimes use $\exists_V . \mathit{Fml}$ as a short-hand for $\exists \mathit{fv}(\mathit{Fml}) - V. \mathit{Fml}$ where *Fml* is some first-order formula and V a set of variables, that is existential quantification of all variables in *Fml* apart from V. We write $\models$ to denote the model-theoretic entailment relation. When writing logical statements we often leave (outermost) quantifiers implicit. Let Fml_1 and Fml_2 be two formulae where Fml_1 is closed (contains no free variables). Then, $\mathit{Fml}_1 \models \mathit{Fml}_2$ is a short-hand for $\mathit{Fml}_1 \models \forall \mathit{fv}(\mathit{Fml}_2). \mathit{Fml}_2$ stating that in any (first-order) model for Fml_1 formula $\forall \mathit{fv}(\mathit{Fml}_2). \mathit{Fml}_2$ is satisfied. We say Fml_2 is *satisfiable* in Fml_1 iff $\mathit{Fml}_1 \models \exists \mathit{fv}(\mathit{Fml}_2). \mathit{Fml}_2$. In case Fml_1 is true we say Fml_2 is satisfiable for short.

3.2 Expressions, Types and Constraints

The language of expressions, types and (Herbrand) constraints is as follows.

Expressions	e	::=	$K \mid x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ g = e\ \mathsf{in}\ e \mid \mathsf{let}\ \begin{array}{l} g :: C \Rightarrow t \\ g = e \end{array}\ \mathsf{in}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I}$
Patterns	p	::=	$x \mid K\ x \ldots x$
Types	t	::=	$a \mid t \rightarrow t \mid T\ \bar{t}$
Constraints	C	::=	$t = t \mid C \wedge C$
Type Schemes	σ	::=	$t \mid \forall \bar{\alpha}. C \Rightarrow t$

Expressions form the minimal core of a functional language such as ML and Haskell. For simplicity, we omit (un-annotated) recursive function definitions. Type-annotated functions may be recursive. As mentioned, in our system annotations `f::C=>t` are lexically scoped. In our syntax, annotations do not have explicit quantifiers. Hence, variables appearing in those annotations cannot be renamed without changing the programs meaning, and once introduced, these variable names cannot be rebound. This is the assumption

$$\text{(Var-}\forall\text{E)}\ \frac{(x : \forall \bar{a}.D \Rightarrow t') \in \Gamma \quad \models C \supset \overline{[t/a]}D}{V, C, \Gamma \vdash x : \overline{[t/a]}t'} \qquad \text{(Abs)}\ \frac{V, C, \Gamma \cup \{x : t_1\} \vdash e : t_2}{V, C, \Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad \text{(App)}\ \frac{V, C, \Gamma \vdash e_1 : t_1 \to t_2 \quad V, C, \Gamma \vdash e_2 : t_1}{V, C, \Gamma \vdash e_1 e_2 : t_2}$$

$$\text{(Let)}\ \frac{V, C_1, \Gamma \vdash e_1 : t_1 \quad \bar{a} = \mathit{fv}(C_1, t_1) - \mathit{fv}(C_2, \Gamma, V) \quad V, C_2, \Gamma \cup \{g : \forall \bar{a}.C_1 \Rightarrow t_1\} \vdash e_2 : t_2}{V, C_2, \Gamma \vdash \mathsf{let}\ g = e_1\ \mathsf{in}\ e_2 : t_2}$$

$$\text{(LetA)}\ \frac{\bar{a} = \mathit{fv}(C_1, t_1) - V \quad V \cup \bar{a}, C_2 \wedge C_1, \Gamma \cup \{g : \forall \bar{a}.C_1 \Rightarrow t_1\} \vdash e_1 : t_1 \quad V, C_2, \Gamma \cup \{g : \forall \bar{a}.C_1 \Rightarrow t_1\} \vdash e_2 : t_2}{V, C_2, \Gamma \vdash \mathsf{let}\ \begin{array}{l} g :: C_1 \Rightarrow t_1 \\ g = e_1 \end{array}\ \mathsf{in}\ e_2 : t_2}$$

$$\text{(Eq)}\ \frac{V, C, \Gamma \vdash e : t_1 \quad \models C \supset t_1 = t_2}{V, C, \Gamma \vdash e : t_2} \qquad \text{(Case)}\ \frac{V, C, \Gamma \vdash e : t_1 \quad V, C, \Gamma \vdash p_i \to e_i : t_1 \to t_2 \quad \text{for } i \in I}{V, C, \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I} : t_2}$$

$$\text{(Pat)}\ \frac{p : t_1 \vdash \forall \bar{b}.(D \mid \Gamma_p) \quad \mathit{fv}(V, C, \Gamma, t_2) \cap \bar{b} = \emptyset \quad V, C \wedge D, \Gamma \cup \Gamma_p \vdash e : t_2}{V, C, \Gamma \vdash p \to e : t_1 \to t_2}$$

$$\text{(Pat-Var)}\ x : t \vdash (\mathit{True} \mid \{x : t\}) \qquad \text{(Pat-K)}\ \frac{K : \forall \bar{a}, \bar{b}.D \Rightarrow t_1 \to \ldots \to t_l \to T\ \bar{a} \quad \bar{b} \cap \bar{a} = \emptyset}{K\ x_1 \ldots x_l : T\ \bar{t} \vdash \forall \bar{b}.(D \mid \{x_1 : t_1, ..., x_l : t_l\})}$$

Figure 1. GADT Typing Rules

made in implementations such as GHC [5] and Chameleon [23]. We will explain the exact meaning of lexically scoped type annotations shortly, see the upcoming typing rule (LetA).

Some readers may wonder about the (Herbrand) constraint component `C` in a lexically scoped type annotation. In our presentation, we allow for explicit Herbrand constraints in type schemes. Hence, we can write annotations such as `f :: List a m -> List a m` equivalently as `f :: t = (List a m -> List a m) => t`. When presenting types to the user, we can of course simplify constraints (and type schemes) by building and applying the most general unifier. But there is really no need to apply this "simplification" in the formal presentation.

We assume that K refers to constructors of user-defined data types. As usual patterns are assumed to be linear, i.e., each variable occurs at most once. For simplicity, we ignore nested patterns such as $K_1\ (K_2\ x)$. They can be translated into a series of shallow patterns by transforming nested patterns in a certain order, say left-to-right as it is done in GHC [5]. In examples, we will use pattern matching notation for convenience.

In our type language we assume that $T\ \bar{t}$ refer to user-definable data types. We use common Haskell notation for writing function, pair, and list types etc. We generally use symbols S, C and D to refer to Herbrand constraints, i.e. conjunctions of type equations. Type schemes have an additional constraint component which allows us to restrict the set of type instances. We often refer to a type scheme as a type for short. Note that we consider $\forall \bar{a}.t$ as a shorthand for $\forall \bar{a}.\mathit{True} \Rightarrow t$ where True denotes the always satisfiable constraint, e.g. $\mathit{Int} = \mathit{Int}$.

We assume that data type declarations are preprocessed and the types of their constructors are recorded in some initial environment Γ_{init}. In the GHC GADT notation, constraints are implicit. We make these constraints explicit by turning the types of constructors into a canonical form where the output has the shape $T\ \bar{a}$. For example,

```
data List a n where
  Nil :: List a Z
  Cons :: a -> List a m -> List a (S m)
```

implies $\mathit{Nil} : \forall a, n.(n = Z) \Rightarrow \mathit{List}\ a\ n \in \Gamma_{init}$ and $\mathit{Cons} : \forall a, m.(n = S\ m) \Rightarrow a \to \mathit{List}\ a\ m \to \mathit{List}\ a\ n \in \Gamma_{init}$. The output type of both constructors has the same shape $\mathit{List}\ a\ n$. Effectively, the constraints $n = Z$ and $n = S\ m$ represent the GADT type refinement in constraint form. Notice that variable m does not appear in the output type. We refer to such variables as *abstract* variables. We are not allowed to make specific assumptions about them during pattern matching, see the upcoming typing rule (Pat).

In general, for each source GHC GADT declaration

```
data T a1 ... am  where
   K :: t1' -> ... -> tl' -> T t1 ... tm
```

we assume $K : \forall \bar{a}, \bar{b}.(a_1 = t_1 \wedge ... \wedge a_m = t_m) \Rightarrow t_1' \to \ldots \to t_l' \to T\ \bar{a} \in \Gamma_{init}$ where $\bar{a} \uplus \bar{b} = \mathit{fv}(t_1', \ldots, t_l', t_1, ..., t_m)$. The symbol $\uplus$ stands for disjoint union. Variables $\bar{b}$ are abstract. In case the constraint component is trivial, i.e. True, GADTs correspond to the form of algebraic data types [8] (a.k.a. (boxed) existential types) known from Haskell and ML.

3.3 Typing Rules

Well-typing of expressions is specified in terms of judgments $V, C, \Gamma \vdash e : t$ where V refers to the set of lexically scoped variables, C is a constraint, Γ refers to the set of lambda-bound variables, predefined and user-defined functions, e is an expression and t is a type. The set V plays a similar role for lexically scoped type variables as Γ for lambda-bound variables. We say a judgment is *valid* iff there is a derivation w.r.t. the rules found in Figure 1. Each valid judgment implies that the expression is well-typed.

Let us take a look at the typing rules in detail. Rule (Var-$\forall$E) combines the rules for variable introduction and type instantiation. We can build a type instance if we can verify that the instantiated constraint is logically contained by the given constraint. Formally, we write $\models C \supset \overline{[t/a]}D$ to denote that for any unifier ϕ of C we have that $\phi(\overline{[t/a]}D)$ is satisfiable.

Rules (Abs) and (App) are straightforward. The quantifier introduction rule is coupled with the (Let) rule. This rule is mostly standard with the exception that we additionally need to exclude type variables in scope from being bound. Further that the constraint C_2 may be satisfiable although C_1 is unsatisfiable, e.g. think of an un-

typable let-definition e_1 which is not used in e_2. Our formulation works fine under a non-strict semantics as is implemented in GHC.

Rule (LetA) deals with a lexically scoped annotation. Variables in $C_1 \Rightarrow t_1$ may refer to variables appearing in some annotation from the enclosing scope. These variables are recorded in V. Hence, when building an explicitly quantified type scheme we only quantify over variables which are not in C_2, Γ and V. Our assumption is that the scope of an annotation is the entire body of that declaration. Hence, when typing the let-body e_1 we extend V by the newly introduced scoped variables. Variables in the annotation do *not* scope over the let-body e_2 (although the value definition does). The inspiration for this form of annotation is taken from System F. Note that rule (LetA) allows for polymorphic recursive functions (for simplicity, we omit the rule to deal with monomorphic recursive functions). We extend the environment with $g : \forall \bar{a}.C_1 \Rightarrow t_1$ when typing the function body. We illustrate the workings of lexically scoped annotations with an example.

Recall the earlier example

```
replace :: (a->a->Bool)->a->a->List a l->List a l
replace eq x y xs =
  let f :: List a m -> List a m
      f Nil = Nil
      f (Cons z xs) = if (eq z x) then Cons y (f xs)
                      else Cons z (f xs)
  in f xs
```

When typing function `f` we find variables a and l in V. Hence, we only quantify over variable m. We type check `f`'s function body with the assumption $\texttt{f} : \forall m.List\ a\ m \rightarrow List\ a\ m$. This is crucial to verify that `f`'s program text is (type) correct (note the polymorphic recursive use of `f`). Furthermore, we can conclude that `f`'s annotation is indeed correct. Hence, the overall program type checks. It is interesting to point out there is no "closed" annotation for `f` such that the program type checks.

There is a bit of design space when it comes to lexically scoped type annotations. We refer the interested reader to [24].

In rule (Eq) we can change the type of expressions. The set C of constraints may not necessarily be the same in all parts of the program (see upcoming rule (Pat)). Therefore, this rule plays a crucial role. Recall (parts) of the earlier example

```
data Sum l m n where
  Base :: Sum Z n n
append :: Sum l m n -> List a l -> List a m -> List a n
append Base Nil ys = ys
```

The pattern match gives rise to $m = m$ (from `Base`) and $l = Z$ (from `Nil`). Variable `ys` has type m. Hence, we can change the type of `ys` to n. Thus, we can verify that `append`'s annotation is correct.

Rules (Case) and (Pat) deal with pattern matching. For convenience, we consider $p \rightarrow e$ as a (special purpose) expression only appearing in intermediate steps. In rule (Pat), we make use of an auxiliary judgment $p : t \vdash \forall \bar{b}.(D \mid \Gamma_p)$ to establish a relation among pattern p of type t and the binding Γ_p of variables in p. Constraint D arises from constructor uses in p. Abstract variables $\bar{b}$ are not allowed to escape which is captured by the side condition $\mathit{fv}(V, C, \Gamma, t_2) \cap \bar{b} = \emptyset$. We type the body of the pattern clause under the "local" type assumption D and environment Γ_p arising out of p.

The rules for the auxiliary judgment are as follows. In rule (Pat-K) we assume that variables $\bar{a}$ and $\bar{b}$ are fresh. Rule (Pat-Var) is standard.

4. Type Inference Framework

We introduce an inference system where we generate implication constraints out of the program text and solve them before building type schemes at let statements. Our main result is that the inference system is equivalent w.r.t. the GADT type system from the previous section. In essence, this result shows that GADT types are solutions of implication constraints and vice versa. In the upcoming Section 5, we make use of this result where we impose some sensible restrictions on solutions such that the number of maximal solutions is finite. Hence, type inference becomes tractable.

4.1 Implication Constraints

To express the typing problem we need the following richer domain of implication constraints. This is a significant difference compared to standard Hindley/Milner.

$$\begin{array}{lrcl} \text{Constraints} & C & ::= & t = t \mid U\ \bar{t} \mid C \wedge C \\ \text{ImpConstraints} & F & ::= & C \mid \forall \bar{b}.(C \supset \exists \bar{a}.F) \mid F \wedge F \end{array}$$

Assumption constraints C on the left-hand side of the implication symbol $\supset$ represent local assumptions arising pattern matchings and type annotations. Recall that type schemes may have a non-trivial constraint component. On the the right-hand side, we may find further implication constraints (due to nested case expressions and type annotations) and Hindley/Milner constraints, i.e. those constraints generated out of expressions following a standard procedure such as algorithm W [12]. Multiple pattern clauses gives rise to conjunctions of implication constraints. Our syntax obviously disallows multiple type annotations for a single definition. Universally quantified type variables refer to variables in type annotations and abstract variables. Existentially quantified type variables belong to Hindley/Milner constraints.

The GADT type system only admits Herbrand constraints. solutions of implication constraints to achieve type inference. We often use symbol S to refer to solutions, i.e. Herbrand constraints.

DEFINITION 1 (Solutions). *Let F be an implication constraint. We say constraint S is a* solution *of F iff $\models S \supset F$. In such a situation, we write $S = solve(F)$.*

4.2 Inference Algorithm

Following [19, 27], we generate implication constraints out of the program text. The actual algorithm is formulated in terms of a deduction system which uses judgments of the form $V, \Gamma, e \vdash_{\mathit{inf}} (F \mid t')$ where the set V of lexically scoped type variables, environment Γ and expression e are input values and implication constraint F and type t' are output values For patterns we introduce an auxiliary judgment of the form $p \vdash \forall \bar{b}.(D \mid \Gamma \mid t)$ to compute the set of abstract variables $\bar{b}$, constraint D, environment Γ and type t arising out of pattern p. The inference rules which are given in Figure 2. The statement "t fresh" is meant to introduce a fresh type variable t.

We briefly review the individual rules. Note that we write $\equiv$ to denote syntactic equality. In rules (Var) and (Pat-K) we assume that the bound variables $\bar{a}$ and $\bar{b}$ are fresh. In rule (Pat) we make use of the implication constraint $\forall \bar{b}.(D \supset \bar{\exists}_{\mathit{fv}(\Gamma, V, \bar{b}, t_e)}.F_e)$ which states that under the temporary assumptions D arising out of the pattern p we can satisfy the implication constraint F_e arising out of e. The $\forall$ quantifier ensures that no abstract variables in $\bar{b}$ escape. The notation $\bar{\exists}_{\mathit{fv}(\Gamma, V, \bar{b}, t_e)}.F_e$ is a short-hand for $\exists \bar{a}.F_e$ where $\bar{a} = \mathit{fv}(F_e) - \mathit{fv}(\Gamma, V, \bar{b}, t_e)$. That is, we existentially quantify over all variables which are strictly local in F_e. Rule ($\exists$Intro) performs a similar simplification step and is assumed to be applied aggressively, e.g. after each of inference step, to keep the set of "visible" variables in constraints small. In rule (LetA) we again make

$$\text{(Var)}\quad \frac{(x : \forall \bar{a}.D \Rightarrow t) \in \Gamma}{V, \Gamma, x \vdash_{inf} (D \mid t)} \qquad \text{(}\exists\text{Intro)}\quad \frac{V, \Gamma, e \vdash_{inf} (F \mid t) \quad \bar{a} = fv(F) - fv(V, \Gamma, t)}{V, \Gamma, e \vdash_{inf} (\exists \bar{a}.F \mid t)}$$

$$\text{(App)}\quad \frac{\begin{array}{c} V, \Gamma, e_1 \vdash_{inf} (F_1 \mid t_1) \\ V, \Gamma, e_2 \vdash_{inf} (F_2 \mid t_2) \\ F \equiv F_1 \wedge F_2 \wedge t_1 = t_2 \rightarrow t \quad t \text{ fresh} \end{array}}{V, \Gamma, e_1\ e_2 \vdash_{inf} (F \mid t)} \qquad \text{(Abs)}\quad \frac{a \text{ fresh} \quad V, \Gamma \cup \{x : a\}, e \vdash_{inf} (F \mid t)}{V, \Gamma, \lambda x.e \vdash_{inf} (F \mid a \rightarrow t)}$$

$$\text{(Let)}\quad \frac{\begin{array}{c} V, \Gamma, e_1 \vdash_{inf} (F_1 \mid t_1) \\ C_1 = solve(F_1) \quad \bar{a} = fv(C_1, t_1) - fv(\Gamma, V) \\ V, \Gamma \cup \{g : \forall \bar{a}.C_1 \Rightarrow t_1\}, e_2 \vdash_{inf} (F_2 \mid t_2) \end{array}}{\Gamma, \mathsf{let}\ g = e_1\ \mathsf{in}\ e_2 \vdash_{inf} (F_2 \mid t_2)}$$

$$\text{(LetA)}\quad \frac{\begin{array}{c} \bar{a} = fv(C_1, t_1) - V \\ V \cup \bar{a}, \Gamma \cup \{g : \forall \bar{a}.D_1 \Rightarrow t_1\}, e_1 \vdash_{inf} (F_1 \mid t_1') \\ V, \Gamma \cup \{g : \forall \bar{a}.D_1 \Rightarrow t_1\}, e_2 \vdash_{inf} (F_2 \mid t_2) \\ F \equiv F_2 \wedge \forall \bar{a}.(D_1 \supset \bar{\exists}_{fv(\Gamma, V, t_1)}.F_1 \wedge t_1 = t_1') \end{array}}{V, \Gamma, \mathsf{let}\ \begin{array}{l} g :: D_1 \Rightarrow t_1 \\ g = e_1 \end{array}\ \mathsf{in}\ e_2 \vdash_{inf} (F \mid t_2)}$$

$$\text{(Case)}\quad \frac{\begin{array}{c} V, \Gamma, p_i \rightarrow e_i \vdash_{inf} (F_i \mid t_i') \quad \text{for } i \in I \\ V, \Gamma, e \vdash_{inf} (F_e \mid t_e) \quad t_1, t_2 \text{ fresh} \\ F \equiv F_e \wedge t_1 = t_e \rightarrow t_2 \wedge \bigwedge_{i \in I} (F_i \wedge t_1 = t_i') \end{array}}{V, \Gamma, \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} \vdash_{inf} (F \mid t_2)} \qquad \text{(Pat)}\quad \frac{\begin{array}{c} p \vdash \forall \bar{b}.(D \mid \Gamma_p \mid t_1) \\ V, \Gamma \cup \Gamma_p, e \vdash_{inf} (F_e \mid t_e) \quad t \text{ fresh} \\ F \equiv \forall \bar{b}.(D \supset \bar{\exists}_{fv(\Gamma, V, \bar{b}, t_e)}.F_e) \wedge t = t_1 \rightarrow t_e \end{array}}{V, \Gamma, p \rightarrow e \vdash_{inf} (F \mid t)}$$

$$\text{(Pat-Var)}\quad \frac{t \text{ fresh}}{x \vdash (True \mid \{x : t\} \mid t)} \qquad \text{(Pat-K)}\quad \frac{K : \forall \bar{a}, \bar{b}.D \Rightarrow t_1 \rightarrow ... \rightarrow t_l \rightarrow T\ \bar{a} \quad \bar{b} \cap \bar{a} = \emptyset}{K\ x_1 ... x_l \vdash \forall \bar{b}.(D \mid \{x_1 : t_1, ..., x_l : t_l\} \mid T\ \bar{a})}$$

Figure 2. Translation to Implication Constraints

use of implication constraints to represent the subsumption problem among two type schemes. To make the annotation constraints available in sub-expressions of e_1, the standard method is to "push" down the constraint the abstract syntax tree. We could so by for example applying our own method described in [24]. However, we avoid such a treatment here to keep the presentation simple. In rule (Let) we need to solve implication constraints before we can build the type scheme.

4.3 Soundness and Completeness

We compare our inference algorithm against the GADT type system.

In our first result, we state that any inference derivation can be turned into a typing derivation if we have a solution.

THEOREM 1 (Soundness of $\vdash_{inf}$). *Let Γ be an environment, e an expression, F an implication constraints and t a type such that $\Gamma, e \vdash_{inf} (F \mid t)$. Let C be a solution of $\bar{\exists}_{fv(\Gamma,t)}.F$. Then $C, \Gamma \vdash e : t$.*

We also obtain completeness. For each typing derivation we find an inference derivation where the constraint in the final judgment of the typing derivation is a solution

THEOREM 2 (Completeness of $\vdash_{inf}$). *Let Γ be an environment, e an expression, C a constraint and t a type such that $C, \Gamma \vdash e : t$. Then, there exists F and t' such that $\Gamma, e \vdash_{inf} (F \mid t')$ and C is a solution of $\bar{\exists}_{fv(C,t)}.(F \wedge t = t')$.*

In summary, the above results show that the GADT type system is equivalent w.r.t. the inference system. We define

$$\begin{array}{lcl} \mathcal{GADT} & = & \{(\Gamma, e) \mid \exists C, t.\ C, \Gamma \vdash e : t\} \\ \mathcal{IS} & = & \{(\Gamma, e) \mid \exists F, t.\ \Gamma, e \vdash_{inf} (F \mid t) \text{ and } F \text{ has a solution}\} \end{array}$$

COROLLARY 1. *We have that $\mathcal{GADT} = \mathcal{IS}$.*

From Section 2.1, we know that the GADT type system does neither enjoy principal types, nor is the set of maximal types finite. Based on the above corollary this translates to the GADT inference system as follows: Implication constraints do neither enjoy principal solutions, nor is the set of maximal solutions finite. We yet have to formally define principality and maximality for solutions and types. This is what we consider next.

4.4 Maximality and Principality

First, we introduce the notion of maximally general solutions (mgs). Such solutions imply maximal types.

DEFINITION 2 (Maximally General Solutions). *Let F be an implication constraint and S a constraint. A* maximally general solution *(or* maximal *for short) S of F is a solution of F such that no more general solution S' exists:*

$$\forall S' :\ \textit{if } \models S \supset S' \textit{ and } S' \textit{ is a solution of } F \textit{ then } \models S' \supset S$$

In case, we interpret $\supset$ as $\leq$, the above says that whenever $S \leq S'$ then $S' \leq S$. This is exactly what we expect from a maximal solution.

We define a comparison relation among types (and constraints to be precise due to the constraint-based nature of the GADT type system) w.r.t. a type environment and expression.

DEFINITION 3. *We define $(C_1, \Gamma \vdash e : t_1) \geq (C_2, \Gamma \vdash e : t_2)$ iff $\models C_2 \wedge t_1 = t_2 \supset C_1$. In such a situation we say that (C_1, t_1) is* more general *than (C_2, t_2) w.r.t. environment Γ and expression e. We say (C_1, t_1) is* principal *(most general) w.r.t. Γ and e if (C_1, t_1) is more general than any other (C_2, t_2) w.r.t. Γ and e.*

The above generalizes a similar definition found for Hindley/Milner which states that $\Gamma \vdash_{HM} e : t_1 \geq \Gamma \vdash_{HM} e : t_2$ iff $\phi(t_1) = t_2$ for some substitution ϕ.

It is straightforward to verify that maximal solutions imply maximal types. The converse holds as well but we omit a formalization. W.l.o.g., we normalize judgments. Notice that $C, \Gamma \vdash e : t$ iff $C \wedge t = a, \Gamma \vdash e : a$ where a is fresh.

LEMMA 1 (Maximal Types). *Let $\Gamma, e \vdash_{inf} (F \mid t)$ such that S is a maximally general solution of F. Let $C, \Gamma \vdash e : t$ such that $(C, \Gamma \vdash e : t) \geq (S, \Gamma \vdash e : t)$. Then, $\models C \supset S$, or equivalently $(S, \Gamma \vdash e : t) \geq (C, \Gamma \vdash e : t)$.*

A solution is principal if it is the only maximal solution.

DEFINITION 4 (Principal Solutions). *Let F be an implication constraint and S a constraint. We say S is a* principal solution *iff (1) S is a solution, and (2) for any other solution S' we have that $\models S' \supset S$*

Principal solutions imply principal types.

LEMMA 2 (Principal Types). *Let $\Gamma, e \vdash_{inf} (F \mid t)$ such that S is a principal solution of F. Then (S, t) is the principal w.r.t Γ and e.*

5. The Intuitive GADT System

This section introduces the most important formal details and results of the intuitive GADT system. Due to space restrictions, we omit the definitions of the formal typing and inference rules and only focus on the definition of solutions which is central to our inference approach. In our system, valid types are represented by valid solutions. As mentioned earlier in Section 2, we must reject non-sensical, non-intuitive solutions to guarantee complete and decidable GADT type inference. In order to define intuitive solutions, we first introduce sensible solutions. The definitions below are inspired by definitions found in [10].

5.1 Solutions

To express the typing problem we need the following richer domain of implication constraints.

$$\begin{array}{lrcl} \text{Constraints} & C & ::= & t = t \mid U\ \bar{t} \mid C \wedge C \\ \text{ImpConstraints} & F & ::= & C \mid \forall \bar{b}.(C \supset \exists \bar{a}.F) \mid F \wedge F \end{array}$$

The GADT system only admits Herbrand constraints as solutions of implication constraints to achieve type inference. We often use symbol S to refer to solutions, i.e. Herbrand constraints.

DEFINITION 5 (Solutions). *Let F be an implication constraint. We say constraint S is a* solution *of F iff $\models S \supset F$.*

5.2 Sensible Solutions

We strengthen the definitions of allowed solutions. For this purpose, we assume constraints in a normalized form:

LEMMA 3 (Normalization). *Each implication constraint F can be equivalently represented by*

$$C_0 \wedge \mathcal{Q}.((D_1 \supset C_1) \wedge \ldots \wedge (D_n \supset C_n))$$

where $\mathcal{Q}$ is a mixed prefix of the form $\exists \overline{b_0}.\forall \overline{a_1}.\exists \overline{b_1} \ldots \forall \overline{a_n}.\exists b_n$.

The idea of these normalization steps is to achieve convenient canonical normal form of implication constraints for the upcoming definitions. In the normal form, only the variables in C_0 are free. Our goal is to find solutions (in terms of types) to these variables, but not just any solutions. Non-sensical types may threaten complete and decidable type inference.

Therefore, we refine the definition of solutions.

DEFINITION 6 (Sensible Solutions). *Let F be an implication constraint, $C_0 \wedge \mathcal{Q}.((D_1 \supset C_1) \wedge \ldots \wedge (D_n \supset C_n))$ its normalized form such that (1) $C_0 \wedge \mathcal{Q}.(D_i \wedge C_i)$ is satisfiable for $i = 1, \ldots, n$ (in case $n = 0$ we demand that C_0 is satisfiable). We say constraint S is a* sensible solution *of F iff*

(2) *$S \wedge C_0 \wedge \mathcal{Q}.D_i$ is satisfiable for $i = 1, \ldots, n$ (in case $n = 0$, we demand that $S \wedge C_0$ is satisfiable),*

(3) *$\models S \wedge C_0 \wedge \mathcal{Q}.D_i \supset \mathcal{Q}.(D_i \wedge C_i)$ for $i = 1, \ldots, n$, and*

(4) *$\models S \supset F$.*

Conditions (1) and (2) have been motivated in Section 2.3 but are now stated for the general case which includes C_0 and quantifiers $\mathcal{Q}$.

Condition (3) we have not seen so far. The motivation behind this condition is to ensure that a case branch body is consistent for every possible instantiated type that satisfies the case pattern. Consider the following example:

```
data T a b where
   K :: a -> T a a
f u = ((\w -> case w of K x -> x + (1::Int)) (K u),
       (\w -> case w of K x -> x && True) (K u))
```

yields (simplified)

$$\begin{array}{l} t_f = t_u \rightarrow (t_1, t_2) \wedge \\ \exists t_x.(t_x = t_u \supset (t_x = Int \wedge t_1 = Int)) \wedge \\ \exists t_x.(t_x = t_u \supset (t_x = Bool \wedge t_2 = Bool)) \end{array}$$

The solution $t_1 = Int \wedge t_2 = Bool$ satisfies condition (2). The first branch is satisfiable

$$\begin{array}{l} t_1 = Int \wedge t_2 = Bool \wedge t_f = t_u \rightarrow (t_1, t_2) \wedge \\ \exists t_x.(t_x = t_u \wedge t_x = Int \wedge t_1 = Int) \end{array}$$

by taking $t_x = Int$, $t_u = Int$ and $t_f = Int \rightarrow (Int, Bool)$. Similarly, we can verify the second branch by taking $t_x = Bool$, $t_u = Bool$ and $t_f = Bool \rightarrow (Int, Bool)$. We can also verify that condition (1) and (4) are satisfied. Though, $t_1 = Int \wedge t_2 = Bool$ is not a sensible solution because variable `u` is in the first branch constrained by `Int` and in the second branch constrained by `Bool`. Indeed, $t_1 = Int \wedge t_2 = Bool$ does not satisfy condition (3).

From now on, we assume that all solutions are sensible.

5.3 Intuitive Types yield Tractable Type Inference

Finally, we give the key definitions behind the intuitive GADT system. Notice that there is a mutual dependency between the two up-coming definitions. We require that each intuitive solution must be built in terms of fully *maximally general solutions* (fms) and each fully maximally general solution must be maximally general among all intuitive solutions.

DEFINITION 7 (Fully Maximally General Solutions). *Let F be an implication constraint, $C_0 \wedge \mathcal{Q}.((D_1 \supset C_1) \wedge \ldots \wedge (D_n \supset C_n))$ its normalized form and S a constraint. We say that S is a* fully maximally general solution *of F iff S is a sensible, maximally general solution among all intuitive solutions of F and*

- *if F has the normalized form $\mathcal{Q}.(D \supset C)$ then $\models S \wedge \mathcal{Q}.D \leftrightarrow \mathcal{Q}.(D \wedge C)$.*
- *if F has the normalized form $C_0 \wedge \mathcal{Q}.((D_1 \supset C_1) \wedge \ldots \wedge (D_n \supset C_n))$ where $n \geq 0$ then $S = C_0 \wedge S_1 \wedge \ldots \wedge S_n$ where S_i is a fully maximally general solution of $\mathcal{Q}.(D_i \supset C_i)$ for $i = 1, \ldots, n$.*

The condition in the first (base) case is motivated in Section 2.3. We restrict our attention to solutions which can be constructed

from the constraints arising out of the program text combined with the assumption constraints arising from pattern matchings or type annotations.

The second case deals with multiple branches. Recall the program from Section 2.3.

```
data Erk a b where
   I :: Int -> Erk Int b
   B :: Bool -> Erk a Bool
f (I a) = x + 1
f (B b) = x && True
```

From function `f`'s program text we obtain the (simplified) implication constraint $(a = Int \supset t = Int) \wedge (b = Bool \supset t = Bool)$. The first implication has the fms $t = a$ and the second implication has the fms $t = b$. The combination $t = a \wedge t = b$ is a solution and therefore according to Definition 7 intuitive. Hence, `f` :: $\forall a.Erk\ a\ a \rightarrow a$ is an intuitive type.

Notice though that $t = a \wedge t = b$ is not an fms for either of the two implications. Definition 7 does not require that the combination of the fms for each individual implication remains an fms. Such a requirement seems too restrictive. We seek to accept as many programs as possible, as long as we can guarantee that their solutions can be described in terms of fms from the (base) implication constraints $C_0 \wedge \mathcal{Q}.(D \supset C)$.

DEFINITION 8 (Intuitive Solutions). *We say that S is an intuitive solution of an implication constraint F iff S is a sensible solution and for some fully maximally general solution S' of F and substitution ϕ we have that $\phi(S') \equiv S$.*

Here comes the all important result.

THEOREM 3. *The set of maximal, intuitive solutions for an implication constraints is finite.*

Thus, we can conclude the following.

COROLLARY 2 (Complete and Decidable Intuitive Type Inference). *We achieve complete and decidable type inference for intuitive GADT programs.*

5.3.1 Properties

Besides finite maximal, intuitive solutions, the intuitive GADT system enjoys three further important properties.

The intuitive GADT system is a conservative extension of the Hindley/Milner system. We refer to a Hindley/Milner program as a program which only makes use of "conventional" algebraic data types, i.e. the constraint component of constructors is $True$ in our internal representation. The implication constraints arising from Hindley/Milner programs can be normalized to the form $\mathcal{Q}.C$ and then we immediately obtain the following result.

LEMMA 4 (Conservative HM Extension). *Implication constraints arising from Hindley/Milner programs either have an intuitive solution or no satisfiable solution exists at all.*

We can also guarantee that an intuitive GADT program remains intuitive if we augment the program with "correct" type annotations. By correct we mean that the types provided by the programmer do not contradict the program text (i.e. the sensible condition (1) in Definition 6 is satisfied). For convenience, we state the result for a simplified case.

LEMMA 5 (Stable Intuitive Solutions). *Let $\forall \bar{a}.D \supset C$ be an implication constraint with an intuitive solution. Let D' be a constraint and $\bar{b}$ some sequence of type variables such that $\forall \bar{a}, \bar{b}.D \wedge D' \wedge C$ is satisfiable. Then, $\forall \bar{a}, \bar{b}.(D \wedge D') \supset C$ has an intuitive solution.*

The approaches in [16, 15] do not enjoy this property. Adding a correct type annotation to a type correct program in [15] may lead to inference failure. Similar issues arise in [16]. We refer to [15] for a discussion.

The final important property of the intuitive system is that it retains principal types.

LEMMA 6 (Retaining Principal Types). *If the implication constraint F has an intuitive solution, then every principal solution is also intuitive.*

The above guarantees that if a program (and all its sub-expressions) has a principal type and the program is intuitive, then our intuitive GADT type inference strategy will find this type.

6. Intuitive GADT Inference Algorithm

In this section we present an inference algorithm for the intuitive GADT system. It is a proof of concept for now, essentially brute-force in nature. More efficient approaches are future work.

6.1 Algorithm Outline

Before we go into the details we first present a highlevel outline of the algorithm.

First the implicication constraints are extracted from the program text according to the rules of Figure 2. Next, we normalize them to the form

$$C_0 \wedge \mathcal{Q}.((C_1 \supset D_i) \wedge \ldots \wedge (C_n \supset D_n))$$

as described in Lemma 3.

Now we consider the implication constraints $\mathcal{Q}.(C_i \supset D_i)$ separately and compute all their respective fms. Section 6.2 provides the detailed procedure for this task.

We combine all the individual fms, taking one candidate from every implication constraint, in all possible combinations. From this set of overall candidates we obtain the actual set of overall fms by enforcing the sensible solution conditions. We only retain those candidates that satisfy conditions (1) and (2) of Definition 6. It is not necessary to consider conditions (3) and (4) as these are ensured by construction and by the other tests. Finally, we drop the remaining candidates that are implied by others. That is, we keep only those which are maximal.

An example of the application of this algorithm was already given in the example at the end of Section 2.3.

6.2 Individual FMS Computation

Let S_{ij} be the jth fms of $\mathcal{Q}.(C_i \supset D_i)$. then we have by definition that:

$$S_{ij} \wedge \mathcal{Q}.C_i \leftrightarrow \mathcal{Q}.C_i \wedge D_i \tag{6.1}$$

We apply two normalizations to the quantified formulas $\mathcal{Q}.C_i$ and $\mathcal{Q}.C_i \wedge D_i$:

1. We eliminate universal quantifiers.
2. We bring the Herbrand constraints in normal form.

Universal Quantifier Elimination For the first we introduce the elim function.

DEFINITION 9 (Quantifier Elimination).
The function elim removes all universal quantifiers from a conjunction of equations over variables $x_i, i = 1 \ldots, n$ in solved form, where any equation $x_i = x_j$ is such $i > j$. Let $\mathcal{Q}$ be a quantifier prefix over variables $x_i, k \leq i \leq n$ where the variables are quantified in order of subscript, e.g. $\forall x_k.\exists x_{k+1}.\exists x_{k+2}.\forall x_{k+3}.\ldots.\exists x_n$. The elim function definition is given in Figure 3.

elim($\mathcal{Q}.F$)
  if $\mathcal{Q}$ is empty **return** F
  if $\mathcal{Q} \equiv \mathcal{Q}_1.Qx_j.\mathcal{Q}_2$ where $Q \in \{\forall, \exists\}$ and $x_j \notin fv(F)$
    return elim($\mathcal{Q}_1.\mathcal{Q}_2.F$)
  let $F \equiv x_i = t \wedge F'$ where x_i is the variable with
    largest subscript on the left hand side.
  if $(\forall x_i) \in \mathcal{Q}$ **return** $false$
  if $(\exists x_i) \notin \mathcal{Q}$
    let $X \equiv fv(t) \cap \mathcal{Q}$
    if $(\forall x_k) \in \mathcal{Q}$ for some $x_k \in X$ **return** $false$
    let $\mathcal{Q}'$ be $\mathcal{Q}$ with quantifiers for variables in X removed
    return $\exists X.x_i = t \wedge$ elim($\mathcal{Q}'.F'$)
  else
    let $\mathcal{Q} \equiv \mathcal{Q}_1.\exists x_i.\mathcal{Q}_2$
    let $X \equiv fv(t) \cap \{x_{i+1} \ldots x_n\}$
    if $(\forall x_k) \in \mathcal{Q}_2$ for some $x_k \in X$ **return** $false$
    let $\mathcal{Q}_{2b}$ be $\mathcal{Q}_2$ with quantifiers for variables in X removed
    return elim($\mathcal{Q}_1.\exists X.\mathcal{Q}_{2b}.F'$)

Figure 3. The elim function

The elim function can be seen as a special case of the quantifier elimination procedure of [9] for arbitrary formulae involving term equations. elim guarantees an existentially quantified term equations as result, where as the procedure of [9] may include negations of existentially quantified term equations.

THEOREM 4. *Let F be a conjunction of equations over variables $x_i, i = 1 \ldots, n$ in solved form, where any equation $x_i = x_j$ is such $i > j$. Let $\mathcal{Q}$ be a quantifier prefix over variables $x_i, k \leq i \leq n$ where the variables are quantified in order of subscript, e.g. $\forall x_k.\exists x_{k+1}.\exists x_{k+2}.\forall x_{k+3}. \ldots \exists x_n$. Then elim$(\mathcal{Q}.F) \leftrightarrow \mathcal{Q}.F$ and elim$(\mathcal{Q}.F)$ is an existentially quantified conjunction of equations or $false$.*

PROOF. We show that all the transformations are logically correct.

$\mathcal{Q}_1.Qx_i.\mathcal{Q}_2.F \;\leftrightarrow\; \mathcal{Q}_1.\mathcal{Q}_\in.F$ where $Q \in \{\forall, \exists\}$ and $x_i \notin fv(F)$.

$\mathcal{Q}_1.\forall x_i.\mathcal{Q}_2.x_i = t \wedge F \quad \leftrightarrow \quad false$. Suppose t is a non-variable term, then clearly $\forall x_i.\mathcal{Q}_2.x_i = t$ is equivalent to false, since some values of x_i cannot equal this term regardless of values of other variables. If $t = x_j$ then $j < i$ and hence we have $\mathcal{Q}_1 \equiv \mathcal{Q}_{1a}.Qx_j.\mathcal{Q}_{1b}$ where $Q \in \{\forall, \exists\}$. Clearly in either case the formula is equivalent to false, since for fixed x_j value it cannot be the case that all values of $x_i = x_j$.

We denote by $t[x_j]$ a term t that contains the variable x_j.

$\mathcal{Q}_1.\exists x_i.\mathcal{Q}_2.\forall x_j.\mathcal{Q}_3.x_i = t[x_j] \wedge F \quad \leftrightarrow \quad false$ Consider $\exists x_i.\forall x_j.x_i = t[x_j]$ then since any value of x_i fixes the value of x_j or causes the equation to be unsatisfiable, this is equivalent to $false$.

We denote by $t[X]$ a term t that contains (among others) the set of variables X.

$\mathcal{Q}_1.\exists x_i.\mathcal{Q}_2.x_i = t[X] \wedge F \;\leftrightarrow\; \mathcal{Q}_1.\exists X.\mathcal{Q}_2 F$. where the variable $X \subset fv(t)$ with subscript greater than i appear existentially in $\mathcal{Q}_2$, and $\mathcal{Q}_{2b}$ is $\mathcal{Q}_2$, with these variables removed.

$$
\begin{array}{ll}
 & \mathcal{Q}_1.\exists x_i.\mathcal{Q}_{2a}\forall x_j.\exists Y.x_i = t[X] \wedge F \\
\leftrightarrow & \mathcal{Q}_1.\exists x_i.\mathcal{Q}_{2a}\forall x_j.\exists X'.x_i = t[X] \wedge \exists X''.F \\
 & \text{where } X' = X \cap Y,\ X'' = X - X' \\
\leftrightarrow & \mathcal{Q}_1.\exists x_i.\mathcal{Q}_{2a}\neg\exists x_j.\neg\exists X'.x_i = t[X] \wedge \exists X''.F \\
\leftrightarrow & \mathcal{Q}_1.\exists x_i.\mathcal{Q}_{2a}\neg\exists x_j.(\neg\exists X'.x_i = t[X]) \\
 & \quad \vee(\exists X'.x_i = t[X] \wedge \neg\exists X''.F) \\
 & \text{by } \neg(\exists Y.x = t[Y] \wedge B) \\
 & \quad \leftrightarrow \; (\neg\exists Y.x = t[Y]) \vee \exists Y.x = t[Y] \wedge \neg B) \\
\leftrightarrow & \mathcal{Q}_1.\exists x_i.\mathcal{Q}_{2a}\neg((\neg\exists X'.x_i = t[X]) \\
 & \quad \vee(\exists X'.x_i = t[X] \wedge \exists x_j.\neg\exists X''.F)) \\
 & \text{since } x_j \text{ does not appear in } x_i = t[X] \\
\leftrightarrow & \mathcal{Q}_1.\exists x_i.\mathcal{Q}_{2a}\exists X'.x_i = t[X] \\
 & \quad \wedge(\neg\exists X'.x_i = t[X] \vee \forall x_j.\exists X''.F) \\
\leftrightarrow & \mathcal{Q}_1.\exists x_i.\mathcal{Q}_{2a}\exists X'.x_i = t[X] \wedge \forall x_j.\exists X''.F
\end{array}
$$

We can repeatedly apply this transformation to move the existential quantifiers for variables in X outside universal quantifiers. Either we reach $\exists x_i$ where we can remove the equation because $\mathcal{Q}.\exists x_i.x_i = t \wedge F' \;\leftrightarrow\; \mathcal{Q}.F'$ since x_i does not occur in F since the equations are in solved form. Or we escape all universal quantifiers and we have $\exists X.x_i = t \wedge \mathcal{Q} - \{\exists X\}.F'$. □

Let us illustrate the use of the elim function on a small example. Consider the following program fragment:

```
data T a = forall b c. a = [b] => K a c

f (K x y) = length x
```

which gives rise to the following implication constraint:

$$\tau_f = T\,\tau_1 \rightarrow \tau_2 \wedge \exists\sigma_1\forall\sigma_2\exists\sigma_3.(\tau_1 = [\sigma_1] \supset \tau_1 = [\sigma_3] \wedge \tau_2 = Int)$$

where `length` has signature $\forall$ `t. [t] -> Int`. Let us apply the elimination function to $\mathcal{Q}.(C \wedge D)$:

$$
\begin{array}{ll}
\text{elim}(\exists\sigma_1\forall\sigma_2\exists\sigma_3.(\tau_1 = [\sigma_1] \wedge \tau_1 = [\sigma_3] \wedge \tau_2 = Int)) & \\
\quad = & \text{elim}(\exists\sigma_1\exists\sigma_3.(\tau_1 = [\sigma_1] \wedge \tau_1 = [\sigma_3] \wedge \tau_2 = Int)) \\
\quad = & \exists\sigma_1.\tau_1 = [\sigma_1] \wedge \text{elim}(\exists\sigma_3.(\tau_1 = [\sigma_3] \wedge \tau_2 = Int)) \\
\quad = & \exists\sigma_1.\tau_1 = [\sigma_1] \wedge \exists\sigma_3.\tau_1 = [\sigma_3] \wedge \text{elim}(\tau_2 = Int) \\
\quad = & \exists\sigma_1.\tau_1 = [\sigma_1] \wedge \exists\sigma_3.\tau_1 = [\sigma_3] \wedge \tau_2 = Int
\end{array}
$$

Herbrand Solved Form In the second normalization, we bring the existentially quantified constraint in Herbrand *solved* form:

$$\exists\bar{y} \bigwedge_i x_i = t_i$$

where the variables x_i are distinct, disjoint from $\bar{y}$ and do not appear in any t_i. A unification algorithm to compute the solved form was sketched by Herbrand and further refined in [11].

Turning the running example into Herbrand solved form we get:

$$\exists\sigma_1.\tau_1 = [\sigma_1] \wedge \tau_2 = Int$$

i.e. $\exists\sigma_3.\tau_1 = [\sigma_3]$ is redundant.

Fully Maximally General Solutions After the two normalization steps, formula 6.1 becomes:

$$S_{ij} \wedge \exists\bar{y} \bigwedge_i x_i = t_i \leftrightarrow \exists\bar{u} \bigwedge_j v_j = s_j \qquad (6.2)$$

This equation is in a suitable form for being solved by Maher's algorithm for finding *fully maximally general answers* (fma) of Herbrand implication constraints. The definition of fma [10] exactly coincides with our definition of fms for the individual branch. In the following we formulate Maher's algorithm.

First we start of with some preliminaries. The *term positions* p of a (Hebrand term) t are ϵ and, if t's toplevel function symbol has arity

n, then, for every term position q of t's ith argument ($1 \leq i \leq n$), $i.q$ are also term positions. We write $pos(t)$ to denote the set of all term positions of t. A term position p of t can be used to retrieve a subterm of t with the function pos:

$$\begin{array}{rcl} pos(\epsilon, t) & = & t \\ pos(i.p, f(t_1, \ldots, t_n), s) & = & pos(p, t_i) \end{array}$$

In terms of this function we define:

$$samepos(t) = \{P \subseteq pos(t) \mid \forall p, q \in P : pos(p, t) \equiv pos(q, t)\}$$

We define the intersection $P \sqcap Q$ of two sets of term positions $P, Q \subset pos(t)$ as $\{p \in P \mid \exists q \in Q, r : q \equiv p.r\} \cup \{q \in Q \mid \exists p \in P, r : p \equiv q.r\}$.

Now we can define the notion of substitution as

$$\begin{array}{rcl} subst(\epsilon, t, s) & = & s \\ subst(i.p, f(t_1, \ldots, t_n), s) & = & f(t_1, \ldots, subst(p, t_i, s), \ldots, t_n) \end{array}$$

and extend it over a set of mutually disjoint term positions $\{p_1, \ldots, p_n\}$, i.e. $\forall i, j : i \neq j \Rightarrow \{p_i\} \sqcup \{p_j\} = \emptyset$, as follows:

$$subst(\{p_1, \ldots, p_n\}, t, s) = subst(p_n, \ldots, subst(p_2, subst(p_1, t, s), s), \ldots, s)$$

The following algorithm generates a superset of all fms.

Algorithm 1 Implication constraint solver

$B \leftarrow \bigwedge_i x_i = t_i$
$E \leftarrow \bigwedge_j v_j = s_j$
$A \leftarrow E$
$P \leftarrow samepos(A)$
while $\exists S \in P, V : subst(S, A, V) \wedge B \leftrightarrow E$ **do**
  choose $S \in P$
  $V \leftarrow$ new variable
  $A \leftarrow subst(S, A, V)$
  if $A \wedge B \not\supset E$ **then**
    backtrack
  end if
  $P \leftarrow \{T \in P \mid \forall p, q \in T : pos(p, A) \equiv pos(q, A)\} \setminus \{S\}$
end while
return A

To obtain the fms, we reject all A that are not maximal, i.e. that are implied by some other A.

Consider again the last example of Section 2.3. One of the two implication constraints is $a = Int \supset t = Int$. After normalization (which is trivial), we get $B \leftarrow a = Int$ and $E, A \leftarrow a = Int \wedge t = Int$. The set P initially contains the empty set, all singleton sets and one binary set:

$$P \leftarrow \{\emptyset, \{1.1.\epsilon\}, \{1.2.\epsilon\}, \{2.1.\epsilon\}, \{2.2.\epsilon\}, \{1.2.\epsilon, 2.2.\epsilon\}\}$$

The empty set $\emptyset$ has no effect at all; so we immediately leave it out of consideration.

1. Choosing $\{1.1.\epsilon\}$ we get: $A \leftarrow v_1 = Int \wedge t = Int$ and $P \leftarrow \{\{1.2.\epsilon\}, \{2.1.\epsilon\}, \{2.2.\epsilon\}, \{1.2.\epsilon, 2.2.\epsilon\}\}$.
   (a) Choosing $\{1.2.\epsilon\}$ we get: $A \leftarrow v_1 = v_2 \wedge t = Int$ and $P \leftarrow \{\{2.1.\epsilon\}, \{2.2.\epsilon\}\}$. Now there is no more position that we can substitute, without violating the implication. Hence our candidate solution is $\exists v_1, v_2.(v_1 = v_2 \wedge t = Int)$, or normalized $t = Int$.
   (b) Choosing $\{2.1.\epsilon\}$, the implication is violated.
   (c) Choosing $\{2.2.\epsilon\}$, the implication is violated.
   (d) Choosing $\{1.2.\epsilon, 2.2.\epsilon\}$, the implication is violated.
2. Choosing $\{1.2.\epsilon\}$ we get a similar single candidate solution $t = Int$.
3. Choosing $\{2.1.\epsilon\}$, the implication is violated.
4. Choosing $\{2.2.\epsilon\}$, the implication is violated.
5. Choosing $\{1.2.\epsilon, 2.2.\epsilon\}$ we get: $A \leftarrow a = v_1 \wedge t = v_1$ and $P \leftarrow \{\{1.1.\epsilon\}, \{2.1.\epsilon\}\}$. Now there is no more position that we can substitute, without violating the implication. Hence our candidate solution is $\exists v_1.(a = v_1 \wedge t = v_1)$, which can be written $a = t$.

Hence, we have two candidate fms $t = Int$ and $a = t$. Neither of these is more general than the other, hence both are actual fms.

7. Conclusion and Future Work

We have shown the incompleteness and undecidability of the general GADT inference problem. We have given a new perspective on type inference for GADTs by combining two well-established concepts: First generating implication constraints out of the program text and then solving these implication constraints via constraint abduction. Type inference becomes tractable if we limit the set of allowable solutions, i.e. solutions must be intuitive. We have explored applications of our inference method to support type error diagnosis.

There is lots of future work ahead. We are currently working on a precise comparison to [19, 16, 15]. We also plan to improve our brute-force implication solver algorithm. Results in [10] imply that the set of maximal, intuitive solutions may be exponential (for a single implication constraint branch). But combining the results of multiple branches will eliminate many solutions. Hence, we expect that a more clever algorithm will behave well in practice. In another direction, we intend to combine our inference method with Henglein's extended occurs check algorithm to alleviate the need to annotate polymorphic recursive programs.

References

[1] C. Chen and H. Xi. Combining programming with theorem proving. In *Proc. of ICFP'05*, pages 66–77. ACM Press, 2005.

[2] J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[3] M. Denecker and A. C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond*, volume 2407 of *LNCS*, pages 402–436. Springer-Verlag, 2002.

[4] P. Dybjer. Inductive sets and families in martin-löf's type theory and their set-theoretic semantics. pages 280–306, 1991.

[5] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[6] Fritz Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems*, 15(1):253–289, April 1993.

[7] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.

[8] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, 1994.

[9] M. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd Logic in Computer Science Conference*, pages 348–357, 1988.

[10] M. Maher. Herbrand constraint abduction. In *Proc. of LICS'05*, pages 397–406. IEEE Computer Society, 2005.

[11] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[12] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[13] C.S. Peirce. *Collected Papers of Charles Saunders Peirce*. Belknap Press of Harvard University Press, 1965.

[14] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[15] S. Peyton Jones, D.Vytiniotis, G. Washburn, and S. Weirich. Simple unifi cation-based type inference for GADTs. http://research.microsoft.com/ simonpj/papers/gadt/, November 2005.

[16] F. Pottier and Y. Régis-Gianas. Stratifi ed type inference for generalized algebraic data types. In *Proc. of POPL'06*, pages 232–244. ACM Press, 2006.

[17] T. Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM Press, 2004.

[18] J.R. Shoenfi eld. *Mathematical Logic*. Addison-Wesley, 1967.

[19] V. Simonet and F. Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, January 2005.

[20] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Proc. of Haskell'04*, pages 80–91. ACM Press, 2004.

[21] P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Proc. of Haskell'03*, pages 72–83. ACM Press, 2003.

[22] M. Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

[23] M. Sulzmann and J. Wazny. Chameleon. http://www.comp.nus.edu.sg/ ˜sulzmann/chameleon.

[24] M. Sulzmann and J. Wazny. Lexically scoped type annotations. http://www.comp.nus.edu.sg/˜sulzmann, July 2005.

[25] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. of POPL'03*, pages 224–235. ACM Press, 2003.

[26] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[27] C. Zenger. *Indizierte Typen*. PhD thesis, Universität Karlsruhe, 1999.