Principal Type Inference for GHC-Style Multi-Parameter Type Classes

Martin Sulzmann[1], Tom Schrijvers[2⋆], and Peter J. Stuckey[3]

[1] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg
[2] Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
tom.schrijvers@cs.kuleuven.be
[3] NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
pjs@cs.mu.oz.au

Abstract. We observe that the combination of multi-parameter type classes with existential types and type annotations leads to a loss of principal types and undecidability of type inference. This may be a surprising fact for users of these popular features. We conduct a concise investigation of the problem and are able to give a type inference procedure which, if successful, computes principal types under the conditions imposed by the Glasgow Haskell Compiler (GHC). Our results provide new insights on how to perform type inference for advanced type extensions.

1 Introduction

Type systems are important building tools in the design of programming languages. They are typically specified in terms of a set of typing rules which are formulated in natural deduction style. The standard approach towards establishing type soundness is to show that any well-typed program cannot go wrong at run-time. Hence, one of the first tasks of a compiler is to verify whether a program is well-typed or not.

The trouble is that typing rules are often not syntax-directed. Also, we often have a choice of which types to assign to variables unless we demand that the programmer supplies the compiler with this information. However, using the programming language may then become impractical. What we need is a type inference algorithm which automatically checks whether a program is well-typed and as a side-effect assigns types to program text.

For programming languages based on the Hindley/Milner system [19] we can typically verify that type inference is *complete* and the inferred type is *principal* [1]. Completeness guarantees that if the program is well-typed type inference

⋆ Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

will infer a type for the program whereas principality guarantees that any type possibly given to the program can be derived from the inferred type.

Here, we ask the question whether this happy situation continues in the case of multi-parameter type classes (MPTCs) [13], a popular extension of the Hindley/Milner system and available as part of Haskell [21] implementations such as GHC [5] and HUGS [9]. GHC and HUGS also support (boxed) existential types [17] and type annotations [20].[1] It is the combination of all these features that make MPTCs so popular among programmers.

In this paper, we make the following contributions:

- We answer the above question negatively. We show that the combination of MPTCs with type annotations and existential types does not enjoy principal types and type inference is undecidable in general (Section 2).
- However, under the GHC [5] multi-parameter type class conditions, we can give a procedure where every inferred type is principal among all types (Section 4).

We omit proofs for brevity, sketches can be found in [26].

To the best of our knowledge, we are the first to point out precisely the problem behind type inference for MPTCs. Previous work [3] only reports the loss of principal types but does not provide many clues about how to tackle the inference problem.

We have written this introduction as if Haskell (GHC and HUGS) is the only language (systems) that supports MPTCs. Type classes are also supported in a number of other languages such as Mercury [7, 10], HAL [2] and Clean [22]. However, as far as we know there is no formal description of multi-parameter type classes and the combination with existential types and type annotations. From now on, we will use MPTCs to refer to the system that combines all these features. For example, Läufer [16] only considered the combination of single-parameter type classes and existential types. The only formal description available is our own previous work [27] where we introduce the more general system of extended algebraic data types (EADTs). Notice that in [27] we discuss type checking but not type inference.

In the next section, we give a cursory introduction to MPTCs as supported in GHC based on a simple example. We refer to [13] for further examples and background material on MPTCs.

2 Multi-Parameter Type Classes

Example. We use MPTCs for the implementation of a stack ADT.

```
class StackImpl s a where
    pushImpl        :: a->s->s
    popAndtopImpl   :: s->Maybe (s,a)
instance StackImpl [a] a where
   pushImpl             = (:)
   popAndtopImpl []     = Nothing
   popAndtopImpl (x:xs) = Just (xs, x)
```

[1] For the purposes of this paper, we will use the term "type inference" to refer to type inference and checking in the presence of type annotations.

In contrast to a *single*-parameter type class, a *multi*-parameter type class such as `StackImpl` describes a relation among its type parameters `s` (the stack) and `a` (the type of elements stored in a stack). The methods `pushImpl` and `popAndtopImpl` provide a minimal interface to a stack. We also provide a concrete implementation using lists.

With the help of an existential type, the stack implementation can be encapsulated:

```
data Stack a = forall s. StackImpl s a => Stck s
```

Each stack is parameterized in terms of the element type `a` whereas the actual stack `s` is left abstract as indicated by the `forall` keyword. We generally refer to variables such as `s` as *abstract* variables. When scrutinizing a stack we are not allowed to make any specific assumptions about `s`. The type class constraint (a.k.a. context) `StackImpl s a` supplies each stack with its essential operations. We use a combination of multi-parameter type classes and existential types.

It is then straightforward to implement the common set of stack operations.

```
push :: a -> Stack a -> Stack a
push x (Stck s) = Stck (pushImpl x s)
pop :: Stack a -> Stack a
pop (Stck s) = case (popAndtopImpl s) of
                  Just (s',x::a) -> Stck s'   -- (1)

top :: Stack a -> a
top (Stck s) = case (popAndtopImpl s) of
                 Just (_,x) -> x
empty :: Stack a -> Bool
empty (Stck s) = case (popAndtopImpl s) of
                    Just (s'::s,x::a) -> False -- (2)
                    Nothing           -> True
```

In case of function `push`, the pattern `Stck s` brings into scope the type class `StackImpl s a`. Thus, we can access specific methods such as `pushImpl x s` to push new elements onto the stack. Functions `pop` and `empty` require lexically scoped type annotations at locations (1) and (2). For example, in case of function `pop` the call `popAndtopImpl s` yields a value of type `Stack b` for some `b` and demands the presence of a type class `StackImpl s b`. Though, the pattern match `Stck s` only makes available the type class `StackImpl s a`. Via the lexically scoped annotation `x::a`, notice that `a` refers to `pop`'s annotation, we convince the type inferencer that `a=b`. Then, the program is accepted.[2]

The informed reader will notice that instead of lexically scoped type annotations we could use functional dependencies [12] to enforce that `a=b`. In our opinion, for many practical examples the reverse argument applies as well. Furthermore, lexically scoped type annotations are a more light-weight extension than functional dependencies. Hence, we will ignore functional dependencies for the purpose of this paper.

What we discover next is that MPTC type inference is not tractable in general.

[2] GHC requires the somewhat redundant pattern annotation `pop (Stck s::Stack a)`, which we omit here for simplicity.

Loss of Principal Types and Undecidability of Type Inference. Consider the following (contrived) program.

```
class Foo a b where foo :: a->b->Int
instance Foo Int b              -- (F1)
instance Foo a b => Foo [a] b   -- (F2)
data Bar a = forall b. K a b
f (K x y) = foo x y
```

The surprising observation is that function `f` can be given the following infinite set of types

$$\texttt{f} :: \mathit{Bar}\ [\mathit{Int}]^n \to \mathit{Int}$$

for any $n \geq 0$, where $[\mathit{Int}]^n$ is a list of lists ... of lists (n times) of integers. We postpone a discussion on why the above types arise to the next section.

The devastating conclusion we draw is that principal types are lost in general. We even cannot hope for complete and decidable type inference because the set of maximal types given to a program may be infinite. We say a type is *maximal* if there is no other more general type. The above types are all clearly maximal.

Function `f` makes use of multi-parameter type classes and "pure" existential types. That is, the type class context of the existential data type definition is empty. This shows that type inference is already a problem for "simple" examples. We do not have to resort to "fancy" examples where we constrain the parameters of constructors by a multi-parameter type class.

The "simple" combination of multi-parameter type classes and type annotations poses the same problems. The following function where we assume the above instances

```
g y = let h :: c->Int
          h x = foo y x
      in h y
```

has a similar infinite set of types $\texttt{g} :: [\mathit{Int}]^n \to \mathit{Int}$ for any $n \geq 0$.

It should be intuitively clear that to establish completeness and decidability of type inference in the MPTC type system we would need to demand an excessive amount of type annotations, something which would seriously impair the practical usefulness of MPTCs.

Therefore, we seek for a compromise and give up on having both completeness and decidability. As is usual in the Hindley/Milner type system, we sacrifice completeness for the sake of decidability. For example, some well-typed programs with polymorphic recursion are rejected because it makes type inference undecidable [8]. Instead, we demand that if type inference succeeds, the inferred type must be principal.

An incomplete type inference has already been implemented in GHC; for example it does not produce a type for either `f` or `g`. The incompleteness of the GHC implementation is captured in a number of conditions on programs. Programs that do not satisfy these conditions are rejected. Unfortunately, there exists neither a formalization of GHC's inference, nor a proof that its conditions guarantee principal types. We will show that the GHC conditions are indeed sufficient, and we present a formal type inference that computes principal types under these conditions.

3 MPTC Inference Overview

We investigate in more detail why MPTC inference is so hard. Then, we motivate our MPTC inference procedure. We postpone a description of the GHC MPTC conditions to the next section.

3.1 Preliminaries

We introduce some basic assumptions and notation which we will use throughout the paper.

We often write $\bar{o}$ as a short-hand for a sequence of objects $o_1, ..., o_n$ (e.g. types etc). We write $fv(o)$ to denote the free variables in some object o. We write "$-$" to denote set subtraction.

We assume that t refers to types consisting of type variables a, function types $t_1 \rightarrow t_2$ and user-definable types $T\ \bar{t}$. We assume primitive constraints of the form $t_1 = t_2$ (type equations) and $TC\ \bar{t}$ (type class constraints).

We generally assume that the reader is familiar with the concepts of substitutions, unifiers, most general unifiers (m.g.u.) etc [15] and first-order logic [23]. 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 ($\exists$). Often, we abbreviate $\wedge$ by "," and use set notation for conjunctions of formulae. We sometimes use $\bar{\exists}_V.Fml$ as a short-hand for $\exists fv(Fml) - V.Fml$ 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. E.g., let Fml_1 and Fml_2 be two formulae where Fml_1 is closed (contains no free variables). Then, $Fml_1 \models Fml_2$ is a short-hand for $Fml_1 \models \forall fv(Fml_2).Fml_2$ stating that in any (first-order) model for Fml_1 formula $\forall fv(Fml_2).Fml_2$ is satisfied.

3.2 Type inference via implication constraints

The examples we have seen so far suggest that we need to perform type inference under "local assumptions." That is, the assumption constraints resulting from type annotations and pattern matches over existential types must satisfy the constraints resulting from the program body. In the case of multiple pattern clauses, the individual assumptions for each pattern clause do not interact with the other clauses. Hence, their effect is localized. This is a significant departure from standard Hindley/Milner inference where we are only concerned with solving sets of primitive constraints such as type equations and type classes.

Our MPTC type inference method makes use of the richer form of implication constraints.

Type Classes	$tc ::= TC\ \bar{t}$
Context	$D ::= tc \mid D \wedge D$
Constraints	$C ::= t = t \mid TC\ \bar{t} \mid C \wedge C$
Implication Constraints	$F ::= C \mid \forall \bar{b}.(D \supset \exists \bar{a}.F) \mid F \wedge F$

Constraints on the left-hand side of the implication symbol $\supset$ represent local assumptions arising from constraints in type annotations and data type definitions. For MPTC programs we can guarantee that only type classes appear on the left-hand side. Constraints on the right-hand side arise from the actual function body

by generating constraints out of expressions following a standard procedure such as algorithm W [19]. Universally quantified type variables refer to variables in type annotations and abstract variables. Recall that abstract variables are introduced by the `forall` keyword in data type definitions. Existentially quantified type variables belong to right-hand side constraints.

The example from before

```
pop :: Stack a -> Stack a
pop (Stck s) = case (popAndtopImpl s) of
                  Just (s',x::a) -> Stck s'   -- (1)
```

gives rise to the implication constraint

$$\forall a.\forall s.(\mathit{StackImpl}\ s\ a \supset \exists t_x.(\mathit{StackImpl}\ s\ t_x \wedge t_x = a))$$

For example, constraint $\mathit{StackImpl}\ s\ t_x$ arises from the program text `popAndtopImpl s` and constraint $t_x = a$ arises from `x::a`. On the other hand, the constraint $\mathit{StackImpl}\ s\ a$ on the left-hand side of $\supset$ arises from the pattern match `Stck s`. The above implication constraint is clearly a universally true statement. Hence, we can argue that the type of `pop` is correct.

If we replace the lexically scoped annotation `x::a` by `x` we find the following variation of the above implication constraint.

$$\forall a.\forall s.(\mathit{StackImpl}\ s\ a \supset \exists t_x.\mathit{StackImpl}\ s\ t_x)$$

This is also a universally true statement. But verifying this statement is more difficult. Checking is not enough here, we need to find a solution for t_x. The problem is that the solving procedure which we outline below will not necessarily find the answer $t_x = a$. The GHC type inferencer will fail as well.

The crucial observation is that without the annotation `x::a`, the program is in fact "ambiguous", hence, illegal. The type t_x of program variable `x` does not appear in the type of function `pop`. Therefore, several solutions for t_x may exist but this choice is not reflected in the type. Haskell type classes follow the open-world assumption. That is, at some later stage we can add further instances such as `instance StackImpl s Int` and thus we find besides $t_x = a$ a second solution $t_x = \mathit{Int}$.

The danger then is that the meaning of programs may become ambiguous. This is a well-recognized problem [11, 24]. For this reason, Haskell demands that programs must be unambiguous. We therefore follow Haskell and rule out ambiguous programs. In terms of implication constraints, the unambiguity condition says that all existentially quantified type variables which do not appear in the final type must be unique. It is certainly not a coincidence that unambiguity also prevents us from guessing solutions.

For our specific case, we could argue that the declaration `instance StackImpl s Int` itself is illegal because it overlaps with the one from before. Hence, there should be only one valid solution $t_x = a$. The point is that the unambiguity check is a conservative check and does not take into account any of the specific conditions which we impose on instances. Hence, the program without the annotation `x::a` fails not because it does not type check, the program is simply plain illegal.

Let's consider the implication constraint for the "devious" program

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

We find that $t_f = Bar\ t_x \rightarrow t_r \wedge \forall t_y.(Foo\ t_x\ t_y \supset t_r = Int)$ where t_f, t_x and t_y are respectively the types of `f`, `x` and `y` respectively, and t_r is the result type. The implication constraint restricts the set of solutions that can be given to these variables. The function body demands that $t_r = Int$ and the call `foo x y` demands $Foo\ t_x\ t_y$. The universal quantifier $\forall t_y$ captures the fact that variable `y` is abstract.

In the previous example, we only had to check that the implication constraint is correct. Here, we actually need to find a solution. The problem becomes now apparent. The constraint $t_f = Bar\ [Int]^n \rightarrow Int$ is a solution of the above implication constraint for any $n \geq 0$. More formally,

$$\forall t_f.(t_f = Bar\ [Int]^n \rightarrow Int) \supset$$
$$(\exists t_r.\exists t_x.t_f = Bar\ t_x \rightarrow t_r \wedge \forall t_y.(Foo\ t_x\ t_y \supset t_r = Int))$$

is a true statement under the assumption that $Foo\ [Int]^n\ t_y$ holds for any t_y, which is implied by the above instances (F1) and (F2). Each one of maximal types $\texttt{f} :: Bar\ [Int]^n \rightarrow Int$ corresponds to one of the solutions $t_f = Bar\ [Int]^n \rightarrow Int$.

A naive "solution" would be to consider the implication constraint itself as the solution. Although, we (trivially) obtain complete inference, this approach is not practical. First, types become unreadable. In the type system, we now admit implication constraints (and not only sets of primitive constraints). Second, type inference becomes intractable. The implication constraints arising from the program text may now have implication constraints on the left-hand side of $\supset$. But then solving these "extended" implication constraints is very close to solving of first-order formulae. Previous work [14] shows that solving of first-order formula with subtype constraints is decidable but has a non-elementary complexity. Note that via Haskell type classes we can encode complex relations such as subtyping. Hence, we abandon this path and consider how to solve implication constraints in terms of sets of primitive constraints.

3.3 Highlights of MPTC implication solver

In its simplest form, we need a solving procedure for implication constraints of the form $D \supset C$ where D consists of sets of type class constraints whereas C additionally contains Hindley/Milner constraints (i.e. type equations). Before we attempt solving, let's consider how to check that $D \supset C$ holds. Checking is a natural first step to achieve solving.

We apply the law that $D \supset C$ iff $D \leftrightarrow D \wedge C$. Thus, checking can be turned into an equivalence test among constraints. The standard method to test for equivalence is to build the canonical normal forms of D and $D \wedge C$ and check whether both forms are identical. In case of type equations, we can build canonical normal forms by building most general unifiers. Here, we additionally find type classes.

The meaning of type classes is specified by instance declarations which effectively define a rewrite relation among constraints. For example, the `instance StackImpl [a] a` declaration from Section 2 implies that the $StackImpl\ [a]\ a$ constraint can be rewritten to $True$. In Haskell speak, this process is known as context reduction, although, we will use the term constraint rewriting/solving

here. In Section 4.1, we formalize how to derive these rewriting steps from instance declarations. For the moment, let's assume a rewrite relation $\rightarrowtail^*$ among constraints where we exhaustively apply instance rules on type classes and rewrite type equations into most general unifiers.

Based on this assumption, we check $D \leftrightarrow D \wedge C$ by executing $C \rightarrowtail^* C'$ for some final constraint C' and testing whether D and C' are identical. Notice that we do not rewrite D which is due to the GHC assumption that constraints D are already in canonical normal form. If D and C' are identical, the check succeeds. Otherwise, we need to infer some missing hypotheses, i.e. constraint. The obvious approach is to take the set difference between C' and D. Recall that we can treat a conjunction of primitive constraints as a set. Then, $C' - D$ is a solution of $D \supset C$. We have that $(C' - D) \supset (D \supset C)$ iff $((C' - D) \wedge D) \supset C$ iff $C' \supset C$ which clearly holds. To summarize, the main idea behind our solving procedure is to rewrite constraints to some canonical normal form. We take the set difference between canonical normal forms to infer the missing assumptions.

To illustrate this solving procedure, we consider a simple example.

```
class F a
class B a b where b :: a -> b
instance F a => B a [a]
data T a = F a => Mk a      -- (T)
f (Mk x) = b x
```

In the data type definition (T), `F a` constrains the type of the constructor `Mk`. Function `g` gives rise to the following implication constraint

$$t_f = T\ t_x \rightarrow b \wedge (F\ t_x \supset B\ t_x\ b)$$

This case is slightly more general than above. Constraints $t_f = T\ t_x \rightarrow b$ will be definitely part of the solution. Solving of $(F\ t_x \supset B\ t_x\ b)$ yields the solution $B\ t_x\ b$. There are no instance rules applicable to $B\ t_x\ b$. Hence, the difference between $B\ t_x\ b$ and $F\ t_x$ is $B\ t_x\ b$. Hence, $t_f = T\ t_x \rightarrow b \wedge B\ t_x\ b$ is a solution. Hence, `f` can be given the type $\forall t_x, b.B\ t_x\ b \Rightarrow t_x \rightarrow b$.

In general, our solving procedures needs to deal with multiple branches (i.e. conjunctions of implications). Universally quantified variables refer to type annotations and abstract variables whereas existentially quantified variables refer to Hindley/Milner constraints. Universal variables are more "problematic" because they cannot be instantiated and are not allowed to escape. In the following, we give an informal discussion of how our solving procedure deals with such cases. The exact details are presented in the upcoming section.

For example, $B\ a\ b \wedge t_r = Int$ is not a valid solution of

$$\forall b.True \supset (B\ a\ b \wedge t_r = Int)$$

because the variable b escapes. We will check for escaping of universal variables by applying a well-known technique known as Skolemization [18]. Skolemization of $\forall b.True \supset (B\ a\ b \wedge t_r = Int)$ yields $True \supset (B\ a\ Sk \wedge t_r = Int)$. The constraint $B\ a\ Sk \wedge t_r = Int$ is clearly not a valid solution because of the Skolem constructor Sk.

We explore solving of multiple branches. The idea is consider one branch at a time.

```
class Foo a b where foo::a->b->Int
instance Foo Int b -- (F)
class Bar a b where bar :: b->a->a
data Erk a = forall b. Bar a b => K1 (a,b)
           | forall b. K2 (a,b)
g (K1 (a,b)) = bar b a
g (K2 (a,b)) = foo a b
```

Function g's program text gives rise to

$$\begin{array}{ll} t = \mathit{Erk}\ a \rightarrow t_3 \wedge t_3 = t_1 \wedge t_3 = t_2 \wedge & (C_0) \\ (\mathit{Bar}\ a\ \mathit{Sk}_1 \supset \mathit{Bar}\ a\ \mathit{Sk}_1 \wedge t_1 = a) \wedge & (F_1) \\ (\mathit{True} \supset \mathit{Foo}\ a\ \mathit{Sk}_2 \wedge t_2 = \mathit{Int}) & (F_2) \end{array}$$

where each branch corresponds to a pattern clause. Universal quantifiers have already been replaced by fresh Skolem constructors.

We start solving the first branch F_1. Based on our method for solving for single implications, we find that $C_0 \wedge t_1 = a$ is a solution for $C_0 \wedge F_1$. We make this solving step explicit by writing

$$C_0 \wedge F_1 \wedge F_2 \gg C_0 \wedge t_1 = a \wedge F_2$$

We will formally define this rewriting relation $\gg$ among implication constraints in the upcoming section. Each time we solve a single implication constraint we replace the implication constraint with its solution. Thus, we incrementally build up the solution for the entire set of implication constraints. Solving of the remaining second branch yields the solution $C_0 \wedge t_1 = a \wedge t_2 = \mathit{Int}$. Hence, we find that $C_0 \wedge F_1 \wedge F_2 \gg^* C_0 \wedge t_1 = a \wedge t_2 = \mathit{Int}$. Notice that $C_0 \wedge t_1 = a \wedge t_2 = \mathit{Int}$ implies $a = \mathit{Int}$ and therefore we can rewrite $\mathit{Foo}\ a\ \mathit{Sk}_2$ to True and thus solve (F_2). We obtain that g has type `Erk Int->Int`.

If we start solving F_2 first, we cannot immediately "fully" solve this implication constraint. The constraint $\mathit{Foo}\ a\ \mathit{Sk}_2 \wedge t_2 = \mathit{Int}$ is not a valid solution because of the Skolem constructor. We can only infer, i.e. add, the *partial* solution $t_2 = \mathit{Int}$. That is, we make the following progress

$$C_0 \wedge F_1 \wedge F_2 \gg C_0 \wedge t_2 = \mathit{Int} \wedge F_1 \wedge F_2$$

If we continue solving F_2 we are stuck. No further constraints can be added at this stage. Our solving method only observes the canonical normal forms of the constraints involved. Based on this information, we cannot infer the missing information $t_1 = a$. Hence, we consider solving of F_1. We find that $C_0 \wedge t_2 = \mathit{Int} \wedge F_1 \wedge F_2 \gg C_0 \wedge t_2 = \mathit{Int} \wedge t_1 = a \wedge F_2$. Finally, we can verify that $C_0 \wedge t_2 = \mathit{Int} \wedge t_1 = a \wedge F_2 \gg C_0 \wedge t_2 = \mathit{Int} \wedge t_1 = a$.

The point is that it may not be possible to solve a single implication without solving other implications first. In case we cannot make progress, i.e. no further constraints can be added, we consider a different branch. In general, a different solving order may yield a different result. Under the conditions imposed by GHC, we can verify that we always obtain the same result. The above example satisfies the GHC conditions and indeed we infer both times the same result.

4 Inferring Principal Types under the GHC Conditions

In our approach, type inference boils down to solving of implication constraints. In a first step, we review some material on type class constraint solving, i.e. solving of sets of primitive constraints. Then, we formalize the MPTC implication

solver. Along the way, we introduce the conditions imposed by GHC sufficient to verify our main result: The MPTC implication solver computes principal solutions, therefore type inference computes principal types, under the GHC MPTC Conditions.

For space reasons, we omit the details of how to generate implication constraints out of the program text. This is by now a standard exercise. For full details see the technical report version of this paper [26].

4.1 Type class constraint solver

In case we only consider multi-parameter type classes (i.e. no existential types and type annotations are involved), type inference boils down to solving of sets of primitive constraints. Instance declarations define a rewrite relation among type class constraints. Hence, the type class constraint solver is parameterized in terms of these rewrite relations.

Following our earlier work [24], we formally define these rewrite relations in terms of Constraint Handling Rules (CHRs) [4]. For each declaration

$$\texttt{instance}\ D\ \Rightarrow\ TC\ \bar{t}$$

we introduce the single-headed CHR $\texttt{rule}\ TC\ \bar{t}\ \iff\ D$. In case, the context D is empty, we generate $\texttt{rule}\ TC\ \bar{t}\ \iff\ \mathit{True}$. The set of all such generated constraint rules is collected in the *MPTC program logic* P.

Logically, the symbol $\iff$ corresponds to Boolean equivalence. Operationally, we can apply a renamed $\texttt{rule}\ TC\ \bar{t}\ \iff\ D$ to a set of constraints C if we find a matching copy $TC\ \bar{s} \in C$ such that $\phi(\bar{t}) = \bar{s}$ for some substitution ϕ. Then, we replace $TC\ \bar{s}$ by the right-hand side under the matching substitution $\phi(D)$ More formally, we write $C \rightarrowtail (C - \{TC\ \bar{s}\}) \cup \phi(D)$ to denote this derivation step. We write $C \rightarrowtail_P^* C'$ to denote the exhaustive application of all rules in P, starting with the *initial* constraint C and resulting in the *final* constraint C'. If the program logic P is fixed by the context, we sometimes also write $C \rightarrowtail^* C'$.

Here is an example to show some CHRs in action. Under the CHRs

```
rule StackImpl (Tree a) a <==> Eq a
rule Eq [a] <==> Eq a
```

we find that $\mathit{StackImpl}\ (\mathit{Tree}[a])[a] \rightarrowtail \mathit{Eq}\ [a] \rightarrowtail \mathit{Eq}\ a$.

We repeat the CHR soundness result [4] which states that CHR rule applications perform equivalence transformations. Recall that $P \models F$ means that any model M satisfying P (treating $\iff$ as Boolean equivalence) also satisfies F.

Lemma 1 (CHR Soundness [4]). *Let* $C \rightarrowtail_P^* C'$. *Then* $P \models C \leftrightarrow \bar{\exists}_{fv(C)}.C'$.

We say P is *terminating* if for each initial constraint we find a final constraint. We say P is *confluent* if different derivations starting from the same point can always be brought together again.

We will demand that CHRs resulting from instances satisfy these properties. Termination obviously guarantee decidability. Confluence guarantees canonical normal forms. Otherwise, we may need to back-track and exhaustively explore all possibilities during solving which may increase the complexity of the solver significantly.

To guarantee confluence and termination, GHC imposes the following conditions on programs.

Definition 1 (Well-Behaved Instances).

Termination Order: *The context of an instance declaration can mention only type variables, not type constructors, and in each individual class constraint all the type variables are distinct.*

In an instance declaration `instance` $D \Rightarrow TC\ t_1 \ldots t_n$*, at least one of the types* t_i *must not be a type variable and* $fv(D) \subseteq fv(t_1, \ldots, t_n)$*.*

Non-Overlapping: *The instance declarations must not overlap: For any two declarations* `instance` $D \Rightarrow TC\ t_1 \ldots t_n$ *and* `instance` $D' \Rightarrow TC\ t'_1 \ldots t'_n$ *there is no substitution* ϕ *such that* $\phi(t_1) = \phi(t'_1), \ldots, \phi(t_n) = \phi(t'_n)$*.*

From now on we assume that the MPTC program logic satisfies the Well-Behaved Instances Conditions. They are sufficient, but not necessary[3] conditions for the essential property that the type class constraint solver is terminating and confluent.

4.2 MPTC implication solver

Solutions and Normalization. We first apply three normalization steps to the implication constraints for convenience.

In the first normalization step, we flatten nested implications and pull up quantifiers, based on the following first-order equivalences: (i) $(F_1 \supset Qa.F_2) \leftrightarrow Qa.(F_1 \supset F_2)$ where $a \notin fv(F_1)$ and $Q \in \{\exists, \forall\}$; (ii) $(Qa.F_1) \wedge (Qb.F_2) \leftrightarrow Qa, b.(F_1 \wedge F_2)$ where $a \notin fv(F_2)$, $b \notin fv(F_1)$ and $Q \in \{\exists, \forall\}$; and (iii) $C_1 \supset (C_2 \supset C_3) \leftrightarrow (C_1 \wedge C_2) \supset C_3$. We exhaustively apply the above identities from left to right until we reach the *pre-normal* form

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

where $\mathcal{Q}$ is a mixed prefix of the form $\exists \overline{b_0}.\forall \overline{a_1}.\exists \overline{b_1} \ldots \forall \overline{a_n}.\exists b_n$. Variables in C_0 are free. Our goal is to find solutions (in terms of types) to these variables.

Definition 2 (Solutions for Fixed Assumption Constraints). *Let* P *be a MPTC program logic,* $F \equiv C_0 \wedge \mathcal{Q}.((D_1 \supset C_1) \wedge \ldots \wedge (D_n \supset C_n))$ *an implication constraint and* C *a constraint. We say that* C *is a* solution *of* F *w.r.t.* P *iff*

1. $C, C_0 \rightarrowtail \ldots \rightarrowtail C$*,*
2. $\models \mathcal{Q}.(C \wedge D_i \leftrightarrow C'_i)$ *where* $C, C_i \rightarrowtail^*_P C'_i$ *for* $i = 1, \ldots, n$*, and*
3. $C \wedge \mathcal{Q}.(D_i \wedge C_i)$ *is satisfiable in* P *for each* $i = 1, \ldots, n$*.*

In such a situation, we say that C *satisfies the* Fixed Assumption Constraint Condition*.*

We say that C *is a* principal solution *iff (i)* C *is a solution, and (ii) for any other solution* C' *we have that* $P \models C' \supset \overline{\exists}_{fv(F)}.C$*.*

The first two conditions define solutions in terms of the operational reading of instances as CHRs. They imply the logical statement $P \models C \supset F$. This

[3] There are other more liberal instance conditions [25] which guarantee the same.

can be verified by straightforward application of the CHR Soundness Lemma. The reason for defining solutions operationally rather than logically is due to the type-preserving dictionary-passing translation scheme [6] employed in GHC. Briefly, assumption constraints D are taken literally and turned into dictionaries. Rewriting them would break separate compilation. Hence, in our definition of solutions we guarantee that assumption constraints are fixed. Interestingly, the Fixed Assumption Constraint Condition is essential to guarantee principal types as we will see later.

The last condition demands that for each particular branch the constraints arising do not contradict each other (i.e. they must be satisfiable). In particular, we reject thus the always false constraint $Int = Bool$ as a solution. Such solutions are clearly non-sensical because they solve any implication constraint. In terms of the GHC translation scheme, unsatisfiable branches represent dead-code, hence, we can ignore them.

In the second normalization step we eliminate all universally quantified variables by Skolemization [18]. That is, we transform $\exists\bar{b}.\forall\bar{a}.F$ into $\exists\bar{b}.[\overline{Sk_a(\bar{b})}/\bar{a}]F$ where Sk_{a_i}'s are some fresh Skolem constructors. We apply this step repeatedly on implication constraints in pre-normal form until we reach the *Skolemized, pre-normal* form

$$C_0 \wedge \exists\bar{b}.((D'_1 \supset C'_1) \wedge \ldots \wedge (D'_n \supset C'_n))$$

For solutions C of Skolemized implication constraints, we additionally demand that no Skolem constructor appears in C.

The Skolemization preserves the set of solutions. It is sufficient to verify this statement for a single branch.

Lemma 2 (Solution Equivalence). *Let P be a MPTC program logic, S a constraint, $\mathcal{Q}.(D \supset C)$ a implication constraint and $\exists\bar{b}.(D' \supset C')$ its Skolemized form. Then, S is a solution of $\mathcal{Q}.(D \supset C)$ iff S is a solution of $\exists\bar{b}.(D' \supset C')$.*

In the last normalization step we drop the outermost existential quantifier $\exists\bar{b}$. However, the choice of variables $\bar{b}$ may not be unique. If this is the case we face the ambiguity problem mentioned in Section 3. Therefore, we only consider unambiguous implication constraints where we can safely drop the existential quantifier.

We say that $C_0 \wedge \exists\bar{b}.((D_1 \supset C_1) \wedge ... \wedge (D_n \supset C_n))$ is *unambiguous* iff $fv(\phi(D_i), \phi(C_i)) \subseteq fv(\phi(C_0))$ for each $i = 1, ..., n$ where ϕ is the m.g.u. of type equations in C_0.[4] The above says that fixing the variables in C_0 will fix the variables in each branch. Checking for ambiguity is obviously decidable.

Next, we introduce a solving procedure for implication constraints in *normal* form, i.e. unambiguous, Skolemized, pre-normal implications constraints of the form

$$C_0 \wedge (D_1 \supset C_1) \wedge ... \wedge (D_n \supset C_n)$$

Solving Method. We formalize the solving method motivated in Section 3.3. In Figure 1, we define a solver $F \gg^*_P C$ for implication constraints F in normal

[4] We assume that $fv(a = Int) = \emptyset$ because a type is bound by the monomorphic type Int.

Primitive: We define $F \gg^*_P C'$ where $C \rightarrowtail^*_P C'$ if $F \equiv C$.

General: Otherwise $F \equiv C_0 \wedge (D \supset C) \wedge F'$. We assume that the most general unifier of type equations in C_0 has been applied to D and C. We execute $C_0, D, C \rightarrowtail^*_P C'$ for some C'. We distinguish among the following cases:

Fail: If $\mathit{False} \in C'$ we immediately fail.

Solved: If $C' - (C_0 \wedge D)$ yields the empty set (i.e. C' and $C_0 \wedge D$ are logically equivalent), we consider $D \supset C$ as solved. We define $F \gg^*_P C''$ if $C_0 \wedge F' \gg^*_P C''$.

Add: Otherwise, we set S to be the subset of all constraints in $C' - (C_0 \wedge D)$ which do not refer to a Skolem constructor.

(a) In case S is non-empty, we define $F \gg^*_P C''$ if $C_0 \wedge S \wedge (D \supset C) \wedge F' \gg^*_P C''$.

(b) In case S is empty, we pick some $(D_1 \supset C_1) \in F'$ and define $F \gg^*_P C''$ if $C_0 \wedge (D_1 \supset C_1) \wedge (F' - (D_1 \supset C_1)) \wedge (D \supset C) \gg^*_P C''$.

(c) Otherwise, we fail.

Fig. 1. MPTC Implication Solver

form w.r.t. the program logic P which, if successful, yields a solution C. The case **Add** subcase **(b)** deals with the situation where we cannot make any further progress, hence, we switch to a different branch. We assume that if none of the branches makes progress we reach subcase **(c)**.

We can establish soundness by a straightforward application of the CHR Soundness and Solution Equivalence Lemma.

Lemma 3 (Soundness of Solving). *Let P be a program logic If $F \gg^*_P C$ for some C then C is a solution of F.*

4.3 Main result

In addition to the Well-Behaved Instances and the Fixed Assumption Constraint Conditions, GHC imposes a third condition on programs.

Definition 3 (GHC MPTC Conditions). *We say a program satisfies the* GHC MPTC Conditions *iff*

- *Instances are well-behaved (see Definition 1).*
- *Each implication constraint in normal-form arising out of a program is unambiguous and has a solution which satisfies the Fixed Assumption Constraint Conditions (see Definition 2).*
- *Each data type definition satisfies the* Bound Type Class Context Condition. *That is, for any*

```
data T a1 ... am = forall b1,...,bn. D => K t1 ... tl
```

and each TC $\overline{t'} \in D$ we have that $fv(\overline{t'}) \cap fv(\bar{b}) \neq \emptyset$.

In fact, GHC 6.4.1 accepts `data T a = forall b. F a => Mk a b` which breaks the Bound Type Class Context Condition. However, in GHC such declarations are interpreted as `data F a => T a = forall b. Mk a b`. That is, `F a` needs to be satisfied when building any value of type `T a`, but `F a` will not appear in a local assumption constraint.

Our main result says:

Theorem 1 (Principal Types for GHC MPTC Programs). *If successful, our solving method computes principal solutions for programs satisfying the GHC MPTC Conditions.*

Before we explain the proof steps necessary to verify the above result, we highlight the importance of the GHC MPTC Conditions.

The Well-Behaved Instances Conditions are not essential. We could replace them with alternative conditions as long as we the type class constraint solver remains confluent and terminating.

GHC imposes the Fixed Assumption Constraint Condition because of dictionary-passing translation scheme. The next example shows that without this condition we may infer non-principal types.

```
class Bar a b c d where bar ::d->c->a->b
class Bar2 a b
class Foo a b d
class Foo2 a
instance Bar2 a b => Bar a b c T2 -- (B)
instance Foo2 a => Foo a b T2     -- (F)
instance Foo2 a => Bar2 a [a]     -- (B2)
data T2 = K
data Erk a d = forall c. Foo a c d => Mk a c d
f (Mk a c K) = bar K c a
```

The program logic P consists of the following rules.

```
rule Bar a b c T2 <==> Bar2 a b  -- (B)
rule Foo a b T2   <==> Foo2 a    -- (F)
rule Bar2 a [a]   <==> Foo2 a    -- (B2)
```

The program text of `f` yields the (simplified) implication constraint ($Foo\ a\ Sk\ T2 \supset Bar\ a\ b\ Sk\ T2$).

Application of our solving method yields the solution $Bar2\ a\ b$ which implies the type $\forall a, b.Bar2\ a\ b \Rightarrow Erk\ a\ T2 \rightarrow b$ for `f`. However, this solution is not principal. We claim there is another incomparable solution $b = [a]$ which corresponds to the type $\forall a.Erk\ a\ T2 \rightarrow [a]$. Both solutions (types) are incomparable and there is no more general solution (type).

We verify that $b = [a]$ is indeed a solution by checking that $b = [a] \wedge (Foo\ a\ Sk\ T2 \supset Bar\ a\ b\ Sk\ T2)$ holds w.r.t. P. From the earlier Section 3, we know that the checking problem $b = [a] \wedge (Foo\ a\ Sk\ T2 \supset Bar\ a\ b\ Sk\ T2)$ can equivalently be phrased as an equivalence testing problem $(b = [a] \wedge Foo\ a\ Sk\ T2) \leftrightarrow (b = [a] \wedge Foo\ a\ Sk\ T2 \wedge Bar\ a\ b\ Sk\ T2)$. Then, we rewrite the left-hand and

right-hand side and check whether resulting constraints are logically equivalent.

$$
\begin{array}{lll}
(1) & b = [a], \mathit{Foo}\ a\ \mathit{Sk}\ \mathit{T2} & \\
\rightarrowtail_F & b = [a], \mathit{Foo2}\ a & (*)
\end{array}
\qquad
\begin{array}{lll}
(2) & b = [a], \mathit{Foo}\ a\ \mathit{Sk}\ \mathit{T2}, \mathit{Bar}\ a\ b\ \mathit{Sk}\ \mathit{T2} & \\
\leftrightarrow & b = [a], \mathit{Foo}\ a\ \mathit{Sk}\ \mathit{T2}, \mathit{Bar}\ a\ [a]\ \mathit{Sk}\ \mathit{T2} & \\
\rightarrowtail_B & b = [a], \mathit{Foo}\ a\ \mathit{Sk}\ \mathit{T2}, \mathit{Bar2}\ a\ [a] & \\
\rightarrowtail_{B2} & b = [a], \mathit{Foo}\ a\ \mathit{Sk}\ \mathit{T2}, \mathit{Foo2}\ a & \\
\rightarrowtail_F & b = [a], \mathit{Foo2}\ a &
\end{array}
$$

The final constraints $b = [a], \mathit{Foo2}\ a$ are equivalent. Hence, $b = [a]$ is a solution. However, $b = [a]$ is not a valid solution under the GHC MPTC Conditions. To obtain the solution $b = [a]$, it is crucial to rewrite the assumption constraint, see the derivation step $(*)$. This violates the Fixed Assumption Constraint Condition.

The Bound Type Class Context Condition is essential as well. Here are excerpts of an example which we have seen earlier in Section 3.3.

```
data T a = F a => Mk a      -- (T)
f (Mk x) = b x
```

The definition (T) violates the Bound Type Class Context Condition. Variable `a` is not bound by the `forall` quantifier. Our solving procedure infers the type $\forall t_x, b.B\ t_x\ b \Rightarrow T\ t_x \rightarrow b$. But this type is not principal. Function `f` can also be given the incomparable type $\forall a.T\ a \rightarrow [a]$ and there is no more general type.

We conclude this section by stating the essential result to verify the above theorem. The crucial observation is that under the GHC MPTC Conditions, the "incremental" solutions S which we compute in solving step **Add** are part of the principal solution (if one exists). Here is the formal result.

Lemma 4 (Principal Progress). *Let P be a program logic derived from instance declarations which satisfy the GHC MPTC Conditions. Let $(D \supset C)$ be a implication constraint in normal form such that (a) $D \rightarrowtail^* D$ and (b) each primitive constraint in D contains at least one Skolem constructor. Let S be a Skolem-free subset of $C' - D$ where $C \rightarrowtail_P^* C'$ from some C' and $\mathit{False} \notin C' - D$. If $(D \supset C)$ has a principal solution, then S is a subset of this principal solution.*

Assumption (a) effectively represents the Fixed Assumption Condition and assumption (b) represents the Bound Type Class Context Condition. The Bound Type Class Context Condition guarantees that for all implication constraints $(D \supset C)$ in normal form we have that each type class constraint in D contains at least one Skolem constructor. Implication constraints resulting from type annotations always satisfy this property.

In combination with Lemma 3, the above results guarantee that our solving method makes progress towards a principal solution. Thus, we can verify the above theorem.

Under the GHC Conditions, we can also verify that the final result is independent of the order of solving. Recall that in solver case **Add**, subcase **(b)** the choice which implication $(D_1 \supset C_1)$ to consider next is not fixed. Effectively, the result below is saying that the implication solver is confluent.

Lemma 5 (Deterministic Progress). *Under the GHC MPTC Conditions, different runs of the MPTC implication solver will yield the same result where we either report a solution or reach one of the failure states. Every implication constraint is considered at most twice.*

5 Conclusion

We have pointed out subtle problems when performing type inference for multi-parameter type classes with existential types and type annotations. In general, we lose principality and decidability of type inference. Under the GHC MPTC Conditions, we give a procedure that infers principal types. To the best of our knowledge, there is no formal description available of the GHC type inference engine or any of the other systems which we have mentioned. Nevertheless, we believe that our procedure is fairly close to the actual GHC implementation. Formalizing the GHC type inference engine based on the principles and methods introduced in this paper is something which we plan to pursue in the future.

Our main result guarantees that every inferred type is principal. The question is whether failure of our inference method implies that no principal type exists?

```
class Foo a b where foo :: a->b
data Bar a = forall b. Foo b a => Mk b
f (Mk x) = foo x
```

Function `f`'s program text generates $t = Bar\ a \rightarrow c \wedge (Foo\ Sk\ a \supset Foo\ Sk\ c)$. Our solving method fails (and so does GHC). It almost seems that $t = Bar\ a \rightarrow a$ is a principal solution. Hence, `f` has the principal type $\forall a.Bar\ a \rightarrow a$. But this is only true if we assume a "closed" world where the set of instances (here none) are fixed. Haskell type classes follow the open world assumption. At some later stage, we may introduce `instance Foo b Int`. Then, `f` can be given the incomparable type $Bar\ a \rightarrow Int$. The point is that the principal types inferred by our MPTC implication solving method are "stable". That is, they remain principal if we add further instances (which must satisfy the GHC MPTC Conditions of course). Failure of our inference method seems to imply that no stable principal type exists. This is something which we plan to investigate further.

Acknowledgments

We thank the reviewers for their comments.

References

1. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. of POPL'82*, pages 207–212. ACM Press, January 1982.
2. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, LNCS, pages 174–188. Springer-Verlag, October 1999.
3. K. F. Faxén. Haskell and principal types. In *Proc. of Haskell Workshop'03*, pages 88–97. ACM Press, 2003.
4. T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.
5. Glasgow haskell compiler home page. http://www.haskell.org/ghc/.
6. C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

7. F. Henderson et al. The Mercury language reference manual, 2001. http://www.cs.mu.oz.au/research/mercury/.
8. Fritz Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems*, 15(1):253–289, April 1993.
9. Hugs home page. haskell.cs.yale.edu/hugs/.
10. D. Jeffery, F. Henderson, and Z. Somogyi. Type classes in Mercury. In J. Edwards, editor, *Proc. Twenty-Third Australasian Computer Science Conf.*, volume 22 of *Australian Computer Science Communications*, pages 128–135. IEEE Computer Society Press, January 2000.
11. M. P. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, Department of Computer Science, September 1993.
12. M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.
13. S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.
14. V. Kuncak and M. Rinard. Structural subtyping of non-recursive types is decidable. In *Proc. of LICS'03*, pages 96–107. IEEE Computer Society, 2003.
15. J. Lassez, M. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.
16. K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, 1996.
17. K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, 1994.
18. Dale Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, 1992.
19. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.
20. M. Odersky and K. Läufer. Putting type annotations to work. In *Proc. of POPL'96*, pages 54–67. ACM Press, 1996.
21. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
22. M.J. Plasmeijer and M.C.J.D. van Eekelen. Language report Concurrent Clean. Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, June 1998. ftp://ftp.cs.kun.nl/pub/Clean/Clean13/doc/refman13.ps.gz.
23. J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.
24. P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.
25. M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 2006. To appear.
26. M. Sulzmann, T. Schrijvers, and P.J.Stuckey. Principal type inference for GHC-style multi-parameter type classes. Technical report, The National University of Singapore, 2006.
27. M. Sulzmann, J. Wazny, and P.J.Stuckey. A framework for extended algebraic data types. In *Proc. of FLOPS'06*, volume 3945 of *LNCS*, pages 47–64. Springer-Verlag, 2006.