To What Extend Is Type Inference for Parameteric Polymorphism Possible in the Presence of Ad-Hoc and Subtype Polymorphism

Daniel Marty
University of Applied Sciences Rapperswil
Supervised by Prof. Dr. Farhad D. Mehta
Seminar Spring 2017

Abstract
The aim of this paper is to compare the capabilities of type inference in different contexts. While type inference is complete in functional programming, mainstream languages still lack this feature and only slowly make progress in this area. The problems and limitations arising if subtype polymorphism and overloading, basic features in object-oriented programming, are components in such languages, are captured and analyzed in this paper. Finally, the type inference capabilities of modern multi-paradigm languages on the basis of Scala and Swift are studied and compared to type inference available in Haskell.

1 Introduction

Type inference for functional languages is common nowadays and is finding a way into mainstream languages with subtypes like Java (SE 7) and C++ (11) to get rid of the tedious task to add type annotations to every identifier [Pot01]. Languages with a shorter history than above mentioned languages, for example Scala and Swift, offer type inference since their first release. Even thought, their type systems were constructed with type inference in mind, they do not provide the same extent as representatives of pure functional programming languages.

Definition 1 (Type Inference)
Reconstructing the type information of terms without type annotations in their external form (the language programmers use to write the code) at compile time, is called type reconstruction or type inference. In explicitly typed languages, where the external and the internal form both contain type annotations, type inference is equivalent to the identity function. [PT00]

Modern programming languages offer different kinds of polymorphism to write generic code and therefore avoid code duplication. The term polymorphism is confusing and refers to different forms of polymorphism depending on the context. In functional programming the term mostly is associated with parametric polymorphism while object-oriented programmers link it to subtype polymorphism and call parametric polymorphism generics [Pie02]. It is common to differentiate three kinds of polymorphism [Sut15].

Definition 2 (Parametric Polymorphism)
If a function is defined over a range of types and acts the same for each type, it is called parametric polymorphism [WB89].

Definition 3 (Ad-hoc Polymorphism) In contrast to parametric polymorphism, a different meaning for the same function name can be achieved by overloading a meaning for each combination of the arguments. That means the function is acting differently for each type [CW85].
Definition 4 (Subtype Polymorphism)
Subtype polymorphism introduces a hierarchy 
$S <: T$ that allows substituting values of a 
supertype $T$ by values of a subtype $S$ which is 
also known as the Liskov substitution principle 
[Lar11].

The following sections cover the question how 
polymorphism affects the possibility to infer types 
and how modern programming languages combine 
different kinds of polymorphism. With Haskell as 
the representative for functional languages and 
Scala and Swift for multi paradigm languages, 
the capability of type inference depending on the 
presence of subtype polymorphism is scrutinized.

2 Hindley-Milner

This section serves to give an introduction to the 
Hindley-Milner which is named after its inventors 
J. Roger Hindley and Robin Milner.

With the Hindley-Milner type system an approach to parametric polymorphism is available 
that infers the most general type of a given program if such a type exists. It first appeared in 
Standard ML and can be found in Haskell and other functional languages in an extended form 
[Wie11].

Definition 5 (Principal Type) Alongside the 
interpreted base types, an infinite collection of 
uninterpreted types, also called principal types, 
exist. Principal types serve as type variables that 
can be instantiated with other types. It has all 
other typings that are possible as instances. The 
function $\lambda x : A . x$ has the type $A \rightarrow A$ and 
represents the identity function for all elements of $A$, 
independently of what $A$ may be [Pie02].

Following example shows how parametric polymorphism works in Haskell without going into 
detail, because how Hindley-Milner is inferring a most general type is not a topic of this paper.

1. `mymap f [] = []`
2. `mymap f (x:xs) = f x : mymap f xs`

Everytime an unkown type is encountered, a 
new principal type gets introduced (in this example, a new type is introduced in alphabetical 
order, but it does not matter how the names are chosen).

The function has two parameters, so its type 
has to be $a \rightarrow b \rightarrow c$. The second argument has 
to be a list (denoted by [] and (x:xs)). With this 
information the principal type $c$ has to be a list 
type $[d]$. Further, $f$ has to be a function that takes 
an element of that list and returns an element of 
type $e$. The type returned by the map function 
has to be a list of type $e$. The most general type 
therefore is $(d \rightarrow e) \rightarrow [d] \rightarrow [e]$.

3 Extend Hindley-Milner with 
ad-hoc polymorphism

In this section, problems arising by adding ad-hoc polymorphism to Hindley-Milner and the solution 
provided with type classes are presented.

Type classes are an approach to add ad-hoc polymorphism to the Hindley-Milner type system. Before the time of Haskell, there was no standard approach to overloading.

3.1 Early Approaches

The equality operation is used in this section to 
show the limitations of the ad-hoc polymorphism 
before Haskell introduced type classes.

A simple approach is to overload equality without adding abstract or function types as it was 
done in the first version of Standard ML. With 
this approach it is possible to check for equality of integers (2*8 == 16) or characters (’A’ == 
’B’) but it is impossible to define a function

1. `member [] y = False`
2. `member (x:xs) y = (x == y) || member xs y`

and call it with

1. `member [4,9,8] 9`
2. `member ”Member?” ’b’`

Another approach is to make it work with all types 
(fully polymorphic). The type of the equality 
operation conforms to

1. `== : a -> a -> Bool`

the type of the function above to

1. `member :: [a] -> a -> Bool`. 
However, not all types possess a meaningful implementation for equality.

The last approach is to limit polymorphism instead of making the equality fully polymorphic. This is done by adding an equality type variable and is the current approach in Standard ML.

Applying equality on a function type or an abstract type causes a type error because there is no substitution available [WB89].

Definition 6 (Equality Type Variable)
Other than principal types, equality type variables can only be substituted by types that admit equality.

3.2 Type Classes
Type classes bring parametric and ad-hoc polymorphism, two kinds of polymorphism that were separated, together.

Type signatures and names of the class are provided in a class declaration.

It means that a belongs to the class Eq if the equality operation (==) or (/=) with the signature a → a → Bool is defined on it. It also provides a default implementation of (/=) or (==) in case that one operation is implemented. This reduces the operations that must be implemented to conform to the type class.

Additionally to the class declaration there is a instance declaration in which implementations of all class operations for a given type are provided.

The instance declaration says that the type Int belongs to the type class Eq and the implementation of the equality is given by the function primEqInt which has to be of type (Int → Int → Bool). If the type is Char or Bool the implementation can be found in primeEqChar respectively primEqBool instead. The function primEqBool could be implemented like following:

If the first argument is true, the id function is called on the second argument, else the not function is called which returns true if the argument is false and vice versa.

The member function mentioned earlier has following type in Haskell:

The type variable a can only be substituted with types that belongs to the class Eq.

3.2.1 Superclasses
Sometimes it makes sense to require type classes that were already defined when constructing new type classes.

The term (Eq a) is called a context of the type and limits the type of a to those admitting equality. Thus, if the operations (==), (<) and (<=) are defined on a type, the type is an instance of the type class Ord. This allows simpler signatures to be inferred. Without superclasses the context would be (Eq a, Ord a). The hierarchy formed by superclasses must be a directed acyclic graph [HHPJW96].

3.2.2 Translation
It is possible to translate a program with type classes to an equivalent program without overloading during the inference process. This program without overloading is then typable in the Hindley-Milner type system. All the instance declarations generate a corresponding dictionary declaration. Overloaded functions will be translated to functions without overloading using this dictionary. The class dictionary contains tuples of functions and sub-dictionaries for all its functions and operators. This allows to call specific
functions depending on the type of the input and the operator.

1. `dictEqInt = ⟨⟨⟩⟩`
2. `dictOrdInt = ⟨⟨primEqInt⟩⟩`

   explain selector

1. `(==) = \langle ⟨⟩, == \rangle − > ==`
2. `getEqFromOrd = \langle ⟨dictEq⟩, <, <= \rangle − > <=
3. `(<) = \langle ⟨dictEq⟩, <, <= \rangle − > <=`
4. `(<=) = \langle ⟨dictEq⟩, <, <= \rangle − > <=`

[HHPJW96].

3.3 Java Generics

Maybe I will show the similarities to Java interfaces here. It is not surprising that Java interfaces Where the equality could be implemented like this:

1. `public interface Eq {
2.     default public bool equalTo(Eq a) {
3.         return ! this.notEqualTo(a);
4.     }
5.     default public bool notEqualTo(Eq a) {
6.         return ! this.equalTo(a);
7.     }
8. }
9.}

Or maybe show an example in Haskell and how it could be done in Java.

1. `public static <T extends Comparable<T>> T max(List<T> elms) {
2.     T cand = elms.get(0);
3.     for (T elm : elms) {
4.         if (cand.compareTo(elm) < 0) {
5.             cand = elm;
6.         }
7.     }
8.     return cand;
9. }

10. `maxTwo :: Ord a ⇒ a → a → a
11. maxTwo x y | x < y = y
12. | otherwise = x
13.}
14. `maxList :: Ord a ⇒ [a] → a
15. maxList [x] = x
16. maxList (x:xs) = maxTwo x (maxList xs)

4 Type Inference with Subtype Polymorphism

The type inference done in Hindley-Milner is undecidable in presence of subtyping or overloading which are basic features of typed, object-oriented programming languages. In section 3, a solution to enrich the Hindley-Milner type system with ad-hoc polymorphism was presented. This section is focusing on obtaining most general types of expressions including additional subtype assumptions.

Deciding if an expression can be typed for any given subtype partial order is equivalent to determining whether a form of satisfiability problem over this given partial order is solvable, which has been shown PSPACE-hard. That means it can be solved by a Turing machine using polynomial amount of space.

Definition 7 (Substitution) TODO [Pie02]

If subtyping is not a component of a language, an instance can be .. with substitution with the consequence that no substitution decreases the size of an expression. The most general typing therefore is also the syntactically shortest.

[HM95].

Definition 8 (Satisfiability Problem) TODO

Definition 9 (Partial Order) A partial order on a set $S$ is given if the relation $\leq$ has following properties:

(i) Reflexivity: $x \leq x$ for all $x \in S$
(ii) Antisymmetry: $a \leq b$ and $b \leq a$ implies $a = b$
(iii) Transitivity: $a \leq b$ and $b \leq c$ implies $a \leq c$

Definition 10 (SSI) Given a finite partial order $\mathcal{B}$ and a System $I$ of inequalities, is $I$ satisfiable in $\mathcal{B}$?

Definition 11 (TIS) Given a signature $\Sigma$ and a term $M$, is $M$ typable over the signature $\Sigma$?
4.1 Reduction from TIS to SSI

Not sure if I should show that or just mention (Type Inference Reduced to Inequality Satisfaction)

4.2 Reduction from SSI to TIS

Not sure if I should show that or just mention (Reducing Inequality Satisfaction to Typability)

4.3 Most General Typings

Definition 12 (Coercion Set C) A Coercion set C is a set of coercions. The coercions in this case are subtype assertions between arbitrary types.

Definition 13 (Type Substitution S) If a type substitution S exists, a typing statement $C', \Gamma' \triangleright M : \sigma$ is called an instance of $C, \Gamma \triangleright M : \sigma$.

(i) $C' \vdash SC$

(ii) $C' \vdash S\sigma$

(iii) $\forall x \in \text{Dom}(\Gamma), \Gamma'(x) = S\Gamma(x)$

Let $M = \lambda f. \lambda x. \lambda y. K(fx)(fy)$. Using type s for x and type t for y, a derivable typing for M is:

$s \preceq u, t \preceq u, v \preceq w, \emptyset \triangleright M : (u \rightarrow v) \rightarrow s \rightarrow t \rightarrow w$

A typing for the term M with empty coercion set is: $\emptyset, \emptyset \triangleright M : (s \rightarrow t) \rightarrow s \rightarrow s \rightarrow t$ There can be no substitution S such that $S\sigma = \sigma'$

(i) $C' \vdash SC$

(ii) $C' \vdash S\sigma \preceq \sigma'$

(iii) $\forall x \in \text{Dom}(\Gamma), C' \vdash \Gamma'(x) \preceq S\Gamma(x)$
5 Local Type Inference

This section focuses on local type inference which is only recovering type annotations with information from adjacent nodes in the syntax tree. Because only local information is used, long distance constraints such as unification variables are not taken into consideration [PT00].

"The reason Scala does not have Hindley/Milner type inference is that it is very difficult to combine with features such as overloading (the ad-hoc variant, not type classes) [...] and subtyping. [...] I'm just saying it's very difficult to make this work well in practice, where one needs to have small type expressions, and good error messages." - Martin Odersky, designer of Scala [Ode08]

Scala’s type inference aims to provide the same level of comfort as languages like Haskell with complete type inference. The restriction to a local context allows to combine subtype polymorphism with type inference in the style of Hindley-Milner. The idea behind this is to use type annotations in some situations which are not common and rarely occur but provide type inference for desirable cases such as:

- Types of local bindings.
- Types of parameters of anonymous functions.
- Type arguments in applications of polymorphic functions.

Bound variables of top-level functions should remain explicitly annotated because type annotation in that position serve as useful documentation which also is considered good practice in Haskell or other languages with complete type inference [Wie11].

5.1 Subtyping and Type Classes

Subtype polymorphism and type classes both solve similar problems.

"Polymorphism captures similar structure over different values, while type classes capture similar operations over different structures" [?].

Scala tries to combine both concepts by introducing a powerful mechanism called implicits which provides the missing link to use type classes in object oriented programming. A variable can be declared as implicit which permits it to be passed as a parameter into a function.

Definition 14 (trait)  **TODO [sca]**

Definition 15 (implicit)  **TODO [sca]**

1 trait Eq[A] {
  2  def equ(x: A, y: A): Boolean
  3  def neq(x: A, y: A): Boolean
  4 }

5 implicit val floatPairEq = new Eq[(Float, Float)] {
  6  def equ(x: (Float, Float), y: (Float, Float)) =
  7      (x._1 == y._1) && (x._2 == y._2)
  8 }

10 def contains[A](e: A, list: List[A]) {
  11    implicit cmp: Eq[A] = list match{
  12      case Nil => false
  13      case x :: xs => cmp.equ(a, x) || contains(e, xs)
  14    }

5.2 Bidirectional Typechecking

maybe something about bidirectional typechecking because it is what scala and swift do

6 Conclusion

The Hindley-Milner algorithm allows global type inference for functional programming and deduces a general type if such a type exists. However, Hindley-Milner is not decidable if subtype polymorphism or overloading is present. There is always a trade-off between supporting those feature vs. more powerful type inference. Languages like Scala are offering type inference with a reduced scope while supporting subtyping and overloading at the same time.
References


