Literature Survey of Type Inference Algorithms for Statically Typed Languages

Saulius Jakovonis[1]

Supervisor(s): Jesper Cockx[1]**, Bohdan Liesnikov**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Saulius Jakovonis
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Bohdan Liesnikov, Annibale Panichella

Abstract

The success of dynamically typed languages such as Python has resulted in an increased interest in supporting type inference in statically typed languages. Type inference refers to automatic type detection based on surrounding context and allows retaining the type safety (and other advantages) of static types, while matching the ease of use of dynamically typed languages. Unfortunately, implementing type inference can be tricky. Researchers have been proposing various methods for type inference ever since the 1970s, however there is no single solution that works for all languages. This paper presents and analyses the proposed methods together with motivations, intuitions, use-cases and examples from practice with the aim of helping new programming language developers understand type inference principles and choose the right technique for their needs.

1 Introduction

Type-checking is a verification mechanism that occurs during the compilation phase of a statically typed language and is not present in dynamically typed languages [1]. It works by scanning the entire program and verifying that all expressions are well-typed. In case any type errors are detected, the program is rejected.

Enforcing well-typedness provides statically typed languages several advantages over dynamically typed languages. First, it guarantees that a program is type safe during runtime. This means that no type errors can occur in a program that has successfully compiled greatly reducing the potential of bugs and also removes many potential test cases that a programmer would have to write otherwise. Second, due to the nature of type checking the types of all expressions are known in advance. As a result programming environments (IDEs) can provide more powerful automatic refactoring, automatic documentation and other quality of life features, enhancing the programming experience of a language [2]. Furthermore, knowing the types in advance allows for better performance optimization and avoids the need for dynamic type checking, which can result in an increased run-time efficiency [2].

However, in many statically typed languages these advantages usually come with a significant cost - a requirement for explicit type annotations, which increases the verbosity of a language and burdens the programmer to know the type system well before writing a program (e.g. in Java, should you choose "ArrayList" or "LinkedList"?). This has led to a belief that dynamically typed languages offer a better experience for programmers, making many programmers (new and experience) forgo of the aforementioned benefits of static typing in favor of more user-friendly atmosphere of dynamically typed languages.

1.1 Type Inference

Fortunately, statically typed programming languages do not inherently need type annotations to make type-checking possible - most type annotations can be omitted by taking advantage of type inference. In programming languages, type inference refers to automatic type detection based on surrounding context. More specifically, it applies when there is a known expression with an unknown type which needs to be determined. Thus having type inference in theory removes the need of explicit type annotations when declaring variables and functions, while preserving the stated benefits of statically typed languages.

However, implementing type inference is not a trivial task and has been studied extensively for over 40 years since one of the first and most influential type inferences algorithms (algorithm W) was first formalized in 1982 [3]. The largest proportion of research was directed towards investigating possible ways to achieve type inference for complex type systems, which allow for features not supported by algorithm W (e.g. subtyping, type classes or first-class polymorphism), a smaller share of research was also investigating the possible ways to improve error reporting for languages which use algorithm W for type inference.

Nevertheless, even after decades of research there still is no single technique that would cover all the complex features commonly used in modern languages. There are a plethora of techniques, all of which prioritize some features, while sacrificing others. Thus a programming language developer has to choose wisely how to design a programming language so that it allows for powerful type inference and what type inference algorithms to use in different scenarios.

1.2 Research Goals and Structure

This paper will focus on providing a comprehensive literature survey of such type inference techniques used in statically typed programming languages by examining a wide range of literature and presenting their key advancements, challenges and use-cases. More specifically the following questions will be answered:

- What are the common issues to implementing type inference?
- What are the proposed solutions to these issues?
- How these solutions compare? Identify advantages and limitations.
- How were these methods adopted in practice?

The rest of this paper is divided as follows. Section 2 will describe how the information was collected (methodology), section 3 will discuss how this paper adheres to the responsible research principles, section 4 will delve deeper presenting an array of type inference algorithms present across literature together with their motivations and specifics. It will also perform comparisons of competing approaches, identifying their main strengths and weaknesses, section 5 will provide some discussion and advice for programming language developers, section 6 will conclude.

2 Methodology

The information present in this literature survey was gathered with a strong emphasis on peer reviewed research papers and primary sources. However, some alternative sources

like pre-prints (e.g. a very recent algorithm FreezeML [4]), articles published by reputable sources (e.g. an article published by Microsoft about C# [5]) or official documentation were used if they provided valuable additional information not present in the peer review research papers covered by the scope of this survey.

Relevant literature sources were gathered by using various search engines (Google, ACM Digital Library, Research Gate) with keywords appropriate to the research topic such as "Type Inference", "Type Systems", "Hindley-Milner" or specific type system features like "First-Class Polymorphism" or "Subtyping". A lot of literature was also gathered through the many related works sections and references used by initial set of literature.

The algorithms present in this survey were compared based on the information composed of the limitations and advantages identified by the original research, together with second-hand evaluations of successors in this field and other sources such as the success of adoption in practice.

3 Responsible Research

The data presented in this paper adheres to the principles of responsible research, since all the information was presented in an honest way i.e. exactly as found in the (identified) references, especially if said research conflicted with our narratives. In cases where our personal opinions were stated, they were clearly identified as such (e.g. section 5). Furthermore, the methods used to gather information were also identified (section 2) to enhance the reproducibility of our research.

4 Type Inference Techniques

The aim of this section is to introduce the different type inference techniques with real world examples of where they have been applied and the motivation behind each technique. First, subsection 4.1 will briefly discuss how type inference is implemented and to what extent it is supported in most popular languages. Then, subsection 4.2 will focus on describing the original Algorithm W, followed by 4.3 which will present the various extensions of this algorithm. Lastly, subsection 4.4 will introduce bidirectional type checking - an alternative approach to algorithm W. More comparisons between these techniques will follow in section 5 discussions.

4.1 Type Inference in most Popular Languages

Although type inference for statically typed languages seems especially advantageous (as described in section 1), in practice languages which are currently the most popular are slow to adopt it. For example, C offers no type inference [6]. Other very popular static languages such as C++, Java and C# offer some type inference, however only in a very limited form (e.g. `var` in Java or C# and `auto` in C++). For example, the `auto` keyword in C++ might be used in a "for-loop" when iterating over a list of a known type without explicit annotations, however when creating a constructor for such a list the return type must be annotated. In essence type inference in these languages boils down to matching the known type on the right side of the initialization expression to the left side or vice-versa [5]. Thus, in these languages type inference alleviates the programmer from only the most primitive type annotations, resulting in most expressions still needing explicit type annotations.

The reasons for this limited adoption might vary, however it is safe to assume that it is difficult to introduce type inference for a language which was not not designed with it in mind. It took years for all of these languages to receive any form of type inference. For example, for Java (released in 1995) type inference for Generics was only introduced in Java 7 (2011), while `var` keyword only in Java 11 (2018). On the other hand, newer statically typed languages like Swift (2014) or Kotlin (2016) were released with type inference from the very beginning and offer a much greater support for it. For example, Swift is able to infer that a function {`x in return x + 1`} is of type $Int \rightarrow Int$, which is not possible in the previously mentioned older languages.

On the other hand, many functional programming languages such as ML, Haskell, OCaml, F# (although not as popular) showcase the true power of type inference. Having their grounds in mathematics, these languages prioritized abstract reasoning and were built with type inference in mind from the very beginning and thus allow for nearly all expressions to be annotation free. In fact, the most well-known and influential of type inference algorithms - algorithm W (or Damas-Milner) was designed for ML in 1982 [3] and since has been used in various forms in many other functional languages (e.g. Haskell, OCaml or F#.). The following section will delve deeper into Algorithm W and its extensions.

4.2 Algorithm W

Algorithm W [7] [3] (sometimes called Damas-Milner or simply W) is a type inference algorithm created by R. Milner and L. Damas for the ML (Meta-Language) programming language in 1982 [3] and since has been used as a base point for type inference algorithms in many functional programming languages, most notably OCaml, Haskell and F# (although its influence extends much further beyond these three languages). Its main power stems from its simplicity and the ability to infer the most general type for any well-typed expression under the Hindley-Milner type system [3], a property not many other type systems enjoy.

This subsection will first introduce the Hindley-Milner type system used in Algorithm W and then the Algorithm W itself.

Type System

The specific type system used by algorithm W is called Hindley-Milner (HM in short), named that way because R. Hindley and R. Milner both discovered and formalized this system independently [8] [7]. Hindley was the first one to discover it in 1969 [8] and called it the "principal type scheme". Milner's work came later in 1978 [7], where he called it the "polymorphic type system". In this paper Milner acknowledged that he only became aware of Hindley's system after "doing the work", nevertheless Milner still mentioned that his type system for Algorithm W could be viewed as an extension of Hindley's system that is specifically tailored for programming languages [7]. Formally HM can be viewed either as

```haskell
id :: a -> a
id = (\x -> x)

-- Compiles ("id" becomes Int->Int)
ex1 = (\f -> (f 5, "text")) id
-- Compiles ("id" becomes String->String)
ex2 = (\f -> (5, f "text")) id
-- Error (Integer 5 is not a String)
ex3 = (\f -> (f 5, f "text")) id

-- Compiles (polymorphic binds allowed)
ex4 = let f = id in (f 5, f "text")
```

Figure 1: Haskell example, demonstrating the restriction of HM that lambda functions can only be applied to monomorphic arguments (see "ex3"), but let bindings don't have this restriction (see "ex4").

an extension of simply-typed λ-calculus or as a restriction of System F (polymorphic λ-calculus):

HM is an extension of simply-typed λ- calculus, because unlike the latter, HM allows for parametric polymorphism and does not require type annotations. As an example consider the identify function - $\lambda x \rightarrow x$. Such a function is not possible in simply-typed λ-calculus since a concrete type would have to be specified for its argument (e.g. $\lambda x : Int \rightarrow x$ or $\lambda x : String \rightarrow x$), yet it is possible in HM.

HM is a restriction of System F because unlike in System F, in HM polymorphism is limited. More specifically it is limited in two ways - first, λ-expressions are not allowed to be applied to polymorphic values, second, polymorphic quantifiers such as $\forall$ can only occur at the outermost level (first-class polymorhism is not supported) [9]. These limitations are necessary to maintain a very important property that HM enjoys, which is that for any expression, most general (principal) types can always be inferred and thus type annotations are never needed [10]. System F, on the other hand, while enjoying less restrictions, loses the principal type property, causing type inference in this system to be undecidable [11].

To lessen the impact of not allowing lambda expressions to be applied to polymorphic values, HM introduced let-bindings, which "cheat" around this issue and formally make HM let-polymorphic (as opposed to fully polymorphic). As an example, consider a code-snippet written in Haskell in Figure 1. In this snippet four expressions are presented together with the aforementioned polymorphic identity function. In examples "ex1" and "ex2" lambda expressions are allowed to be applied to "id" because internally "ex1" and "ex2" only apply it to either Int or String (not both), thus "id" is instantiated as a monomorphic type of $Int \rightarrow Int$ in "ex1" or $String \rightarrow String$ in "ex2", however "ex3" fails because the function is applied to both Int and String, requiring id to be polymorphic. Fortunately, "ex4" shows that an equivalent expression that does use "id" in a polymorphic fashion can be achieved using the let-binding.

Formally HM type system consists of 6 typing rules - variable (Var), function application (App), lambda abstraction (Abs), let generalization (Let), type instantiation (Inst) and type generalization (Gen). All these rules are presented in a common formal type system notation in Figure 2.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ [\text{VAR}] \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}\ [\text{ABS}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ [\text{APP}]$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}\ [\text{LET}]$$

$$\frac{\Gamma \vdash e : \sigma_2 \quad \sigma_2 \sqsubseteq \sigma_1}{\Gamma \vdash e : \sigma_1}\ [\text{INST}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma}\ [\text{GEN}]$$

Figure 2: Declarative representation of the Hindley-Milner type system rules in formal notation. Adapted from Damas and Milner [3].

The following subsection will introduce the actual type inference algorithm W for this type system.

Type Inference Algorithm

In addition to extending Hindley's type system, Milner's original paper publised in 1978 [7] also laid the groundwork for algorithm W by introducing a type-checking algorithm for Hindley-Milner type system. This work was later used in the subsequent paper by R. Milner's and L. Damas paper published in 1982 [3] where the inference part of algorithm W was finalized by R. Milner and the proofs for the decidability, completeness and soundness of algorithm W were established and presented by L. Damas. This paper showed that under the Hindley-Milner type system, any well-typed expression can be inferred a principal (most general) type using algorithm W. Principle type property is very important and suggests that for a language using Hindley-Milner type system together with Algorithm W, type annotations are never needed. However, in many cases adding new features to increase the expressiveness of HM results in losing the principal types property as shown by some extensions presented in the following subsection.

At its core Algorithm W type inference relies on type unification [7]. J. Robinson was the first to provide a formal algorithm for unification together with proofs of its completeness in 1965 [12] and it is his algorithm that is used as a subprocedure in the original algorithm W.

Unification can be viewed as a function that takes in two type expressions (e.g. "e_1" and "e_2") and provides a set of substitutions S, so that when when these substitutions are applied to both expressions "e_1" and "e_2", the resulting expressions are equal (i.e. $S(e_1) = S(e_2)$), we say that S unifies "e_1" and "e_2". Furthermore, Robinson's algorithm produces substitutions that are the most general [12], meaning that the resulting type has the least amount of constraints. As an ex-

ample consider

$$f_1 : \alpha \to Int \quad and \quad f_2 : \alpha \to \beta,$$

where $\alpha \to Int$ denotes a type of a function that takes a value of any type α and returns a value of concrete type Int. Similarly $\alpha \to \beta$ denotes a type of a function that takes a value of any type α and returns a value of any type β (f_2 is more general than f_1). If we were to unify f_1 and f_2, the resulting most general set of substitutions S would be

$$S = \{\beta \mapsto Int\}.$$

Here we place a constraint on β, saying that β has to be of type Int, so that

$$S(f_1) = S(\alpha \to Int) = \alpha \to Int$$
$$S(f_2) = S(\alpha \to \beta) = \alpha \to Int$$
$$\text{thus} \quad S(f_1) = S(f_2).$$

It also easy to see that S is also the most general unifier, yet not the only one. Consider another possible unifier S', where

$$S' = \{\alpha \mapsto Int, \beta \mapsto Int\}.$$

S' still unifies f_1 and f_2 (i.e. $S'(f_1) = S'(f_2) = Int \to Int$), yet the resultant type is not the most general since S' imposes an unnecessary additional constraint $\alpha \mapsto Int$ and thus over-restricts the type.

Algorithm W works by combining this unification algorithm together with the type system rules shown in 2. It applies these rules bottom-up, meaning that it first recursively traverses the abstract syntax tree (AST) and applies new type variables to the leaves as a base case. Then, the recursive tree is collapsed by unifying the "child" types of each expression into a single "parent" type until either at some stage the unification algorithm fails to unify (there doesn't exist a substitution that would make the types equal) or the root of the recursive tree is reached and the resultant type of the expression together with the set of substitutions is returned. In the case where unification fails a type error is returned.

For example, given an application expression

$$e' = e_1\ e_2,$$

with the following types

$$e' : \tau_2, \quad e_1 : \alpha \quad and \quad e_2 : \tau_1,$$

it is known from the application rule ([APP] in Figure 2) that α needs to unify with some type $\tau_1 \to \tau_2$. In this case the algorithm assigns the parent expression e' a new (unused) variable name τ_2, since due to this bottom-up approach the expression e' has not yet been processed by the algorithm and has no type attached to it. The types α and τ_1, on the other hand, have already been inferred to the most general types in their respective sub-trees, thus these concrete types are used for the unification. After unifying α with $\tau_1 \to \tau_2$, if a solution exists, a new set of substitutions is obtained for all three type variables α, τ_1 and τ_2, that when applied result in these types being the most most general in this new broader scope of the AST that includes expression e'. If e' was the root, algorithm W halts and returns the most recent set of subsitutions together with the type τ_2 belonging specifically to e'. If e' was not the root of the AST the algorithm continues traversing the AST upwards and unifying expressions based on the type rules in Figure 2 similarly to this function application example until the root is reached. If at any point of the algorithm no successful set of unifying substitutions exists, the algorithm halts with a type error, notifying that some problematic expression could not be unified to some other expression.

Although simple, efficient and enjoying the principal type property, algorithm W is limited to Hindley-Milner type system, thus it is not very expressive. Due to this and other problems over the years many extensions to algorithm W were proposed that either improve it in some way or adapt it to handle more complex language features. This will be the focus of the following subsection.

4.3 Extensions of Algorithm W

Algorithm W is very efficient and guarantees that for any well-typed expression, it will produce the most general type (i.e. type annotations are never needed). However, this guarantee would only work in a type system which does not allow for more complex type system features such as type classes, subtyping, first-class polymorphism and others. In addition, algorithm W suffers from poor error localization, meaning that when type related errors occur the error location is frequently reported far away from the actual source. As a result, over the years many extensions to the original Algorithm W were proposed. OutsideIn(X) (2011) [13] for type classes, MLF (2003) [14], HMF (2008) [9], FreezeML (2020) [4] for first-class polymorphism, MLsub (2017) [15], Simple-sub (2020) [16] for subtyping and SOLVE [17] (2002), "Practical Error Localization" [18] (2015) for error localization.

Subtyping

Subtyping is an essential type system feature for many programming languages. For example, subtyping shows up in object-oriented programming (OOP) languages such as Java, C# and C++ through the support of inheritance[1] [20] (widely considered a fundamental aspect of OOP paradigm). Other examples include traits in Scala, union types in TypeScript, and objects in OCaml (implemented as records with subtyping).

In general, implementing subtyping means allowing for types to be related to each other in terms their constraints (or conversely in terms of features they offer), such that more precise types (e.g. Dog) could be used in places where more general types (e.g. Mammal, Animal, etc.) are expected, which makes for a more expressive and convenient language. Formally we express this relationship as $Dog \leq Animal$ or generally $\tau \leq \tau'$, meaning that the type τ is a subtype of τ' (conversely τ' is a supertype of τ) and implies that τ might be used anywhere where τ' is expected (the opposite does not hold unless types are equal). In functional languages that do not have objects, a more standard example to explain subtyping could be integers and natural numbers. Since each natural number is itself an integer, we can say that $Nat \leq Int$,

[1] Inheritance does not always imply subtyping as remarked by Cook et al. [19], but in these language it does [20].

and thus we can use naturals for any function/list/record that expects integers. The opposite does not hold since not all integers are naturals.

Unfortunately, even though subtyping is very useful, combining it with type inference has been a long lasting issue. Hindley-Milner type system (that Algorithm W is built for) does not feature subtyping for a very good reason - since at its core its type inference algorithm relies heavily on unification. Unification equates two types by generating a set of substitutions, which simplifies/merges constraints on types into a concrete type in an efficient way (as explained in subsection 4.2). However, in the presence of subtyping unification no longer works, since instead of type equalities i.e. $\alpha = \beta$, implementing subtyping would result in type inequalities i.e. $\alpha \leq \beta$, for which there is no way to perform unification [21].

The most natural approach to avoid unification is to collect all of the subtyping constraints while traversing the AST and solve them in one go at the the end of the algorithm. If there exists no type satisfying the set of constraints imposed to an expression, then it means this expression is ill-typed. This problem of checking whether there exists a type which satisfies a set of subtyping constraints is known as subtype satisfiability [22].

However, collecting constraints without simplifying them is not a good idea. Without simplification, as the size of the program grows these constraints would accumulate and the types of expressions would quickly become too large to comprehend for programmers (e.g. when shown in error messages). Furthermore, since the time-complexity of subtype satisfiability is high (currently the most efficient algorithms are $O(x^3)$ at best for most type systems [22]) such an approach would not be practical at all for large applications. Thus some sort of constraint simplification is necessary for type inference under subtyping to be practical. Pottier discusses this issue in detail and proposes a simplification algorithm in his 1996 [21] and 2001 [23] papers.

Due to the difficulties mentioned above, extending Algorithm W to handle subtyping has been a real struggle. Researchers have been proposing various methods ever since the early 1980s. Some of the earliest examples are Mitchell (1984) [24], Stanisfer (1988) [25] and Fuh and Mishra (1989) [26]. These early attempts at supporting type inference under subtyping were not satisfying due to requiring heavy restrictions, for example none of these algorithms allowed let-polymorphism (property of Algorithm W) or function overloading (ad-hoc polymorphism). Other algorithms focused solely on enabling overloading without subtyping [27] [28] (the topic of subsection 4.3).

In 1994, G. S. Smith combined these two lines of work and proposed a sound and complete algorithm that would infer principal types in the presence of both - subtyping and overloading [29], however with some restrictions yet again (though not as severe as the previous work) such as only allowing global overloading, since local overloading would break the principal type property. Although Smith's work looked promising, we could not find practical applications of his type system and algorithm, which could be due to the high cost of inferring types for larger programs (as he mentions himself in the future work section). There are many papers that continued the work, much more than we can cover given the scope of our paper. Some papers not mentioned already are Aiken and Wimmers (1993) [30], F. Pottier (1998) [31][32].

The most recent subtyping extensions of Algorithm W are MLSub (2017) [15] by Dolan and Mycroft and SimpleSub (2020) by Parreaux [16]. These two algorithms are very related, since SimpleSub was released as a simplification of MLSub that was implemented in only 500 lines of Scala code and was much easier to understand. Parreaux critiqued MLSub as being hard to understand even by experts, mentioning that reading the related lengthy Dolan's Ph.D. thesis was necessary to understand MLSub in full. He showed that his simplified version, although not as "mathematically elegant" inferred equivalent types for a "million randomly generated expressions" [16]. Source code for both algorithms has been published on GitHub [33][34] and both also feature fully working demos online [35][36].

MLSub proposed a sound and complete algorithm that would infer principal types in the presence of subtyping for an ML-like language. This language included everything that the HM type system included with an addition of records. Unlike a lot of previous attempts, MLsub's greatests strengths were allowing let-polymorphism and inferring types which are small/compact (thus easy to read). To acomplish compact types, it applied constraint simplification methods inspired by Pottier's (1998) work [32]. Furthermore, MLSub differed from most previous work in that it introduced a concept of "biunification". Previous algorithms usually traversed the application collecting a set of constraints and then solved them as a separate step all at once, while MLsub merged constraints while traversing the AST using "biunification" in a similar way how unification was applied in Algorithm W [6].

The greatest weaknesses of MLSub is that it does not allow for function overloading, restricts function inputs to union types (intersection types not allowed) and outputs to intersection types (union types not allowed) and is not very efficient (possibly exponential time [37]). The first two restrictions are needed to achieve the compact types that the algorithm is known for. This is because the simplification algorithm of MLsub exploits the assumption that the types $(int \to int) \cap (nat \to nat)$ and $(int \cup nat) \to (int \cap nat)$ are equivalent, which is only true if those restrictions mentioned above hold [16]. As with other algorithms mentioned previously, we are could not find any integrations of MLsub or SimpleSub with widespread languages.

First-Class Polymorphism

First-class polymorphism is another feature not supported by Algorithm W. It means allowing polymorphic quantifiers $\forall\alpha$ to be first-class citizens. For quantifiers to be first-class citizens they have to have the ability to appear anywhere within the type signature [38]. Another term that is sometimes used for this concept is impredicative polymorphism. In HM, quantifiers are second-class/predicative because they are restricted to appear only in the outermost level of a type [4], for example the type $\forall\alpha.[\alpha] \to [\alpha]$ (a function that takes a list of any type and returns a list of the same type) is allowed, however $[\forall\alpha.\alpha \to \alpha]$ (a list of functions that take an expres-

sion of any type and return an expression of the same type) is not since the quantifier $\forall\alpha$ appears inside the list constructor, not as a predicate.

As already discussed in subsection 4.2, the reason why HM does not support first-class polymorphism is that allowing it would make HM equivalent to System F (for which type inference becomes undecidable [11]). As a consequence, any extension of HM enabling first-class polymorphism is guaranteed to lose the principle type property, meaning some type annotations will become necessary [2].

Losing principal types is a considerably large trade-off, which might not be justifiable for some programming languages. Especially since first-class polymorphism occurs infrequently in practice and can usually be avoided [39]. However, there certainly are rare cases [40] where it becomes unavoidable or the workarounds required are very impractical as explained by Serrano et al. [38]. Consequently, some programming languages might consider it a valuable feature to have e.g. Haskell's impredicative types extension (will be briefly discussed in subsection 4.4) and thus there has been a considerable amount of research into this topic.

Examples of extensions to algorithm W which support first-class polymorphism are MLF (2003) [14], HMF (2008) [9], FreezeML (2020) [4].

Type Classes (Function Overloading)

Type classes are another type system feature not present in HM type system, yet in practice supported and commonly used in languages such as Haskell or Scala. The purpose of type classes is to allow function overloading [28]. Function overloading is a form of ad-hoc polymorphism and is useful when seeking non-uniform behavior for functions of the same name. For example a function $show$ that converts its input expression into a string needs different implementations for different types.

Overloading functions using type classes works in a similar way to interfaces or abstract classes common in OOP languages [41]. Type classes impose a set of requirements (i.e. a set of function overloadings) that have to be implemented for some type, so that this type could become a member of this type class. Then any function wishing to use these overloaded functions have to restrict their inputs to be of the typeclass that specifies those functions. For example, in Haskell, for a type to be a member of type class Num it has to implement arithmetic operations $+$, $-$, $*$, etc. and then any function that uses these operations has to restrict its input to be of typeclass Num.

All numeric types (Int, Nat, Float, etc.) are members of Num typeclass in Haskell, which means that the mentioned arithmetic operations are defined for all of them, something not possible using only parametric polymorphism present in HM. In HM, the function $+$ would only be able to be defined for all types or for a single specific type like Int, thus it is either too general or to specific for this desired purpose. Furthermore, it would need to have an identical implementation, irrespective of the input types. All of these limitations restrict the expressiveness of a programming language, thus it is valuable for many programming languages to support overloading whether through type classes or otherwise.

The creation of type classes as an extension to Hindley-Milney type system is often accredited to Wadler and Blott (1989) [28]. Their idea was modified (to work with more complex language features than intended) and integrated in the early versions of Haskell, the specific changes are explained in a paper by Hall et al (1996) [42]. This system was innovative, however it imposed strong restrictions and even allowed for some ill-typed programs to typecheck like `true + true` for the overloaded operator $+$ not defined for booleans as noted by Smith [29]. More recently, Haskell adopted a new type inference algorithm for type classes called OutsideIn(X) (2011) [13]. OutsideIn(X) allowed combining type classes with other more complex features like GADTs (Generalized Algebraic Data Structures) and itself was an extension of HM(X) (1999) [11]. HM(X) created a general type inference framework for Hindley-Milner extensions that was easy to adapt to arbitrary constraint systems and offered a universal type inference algorithm that did not depend of the specific constraint logic used [43].

Error Localization

Finally, we would like to discuss another group of related Algorithm W extensions, which do not introduce any new features but address an important problem associated with this algorithm - error localization.

Algorithm W has been known to produce uninformative errors almost since its invention in the early 80s. In 1986 Wand was one of the first researchers that addressed this problem and proposed an improvement to the original algorithm [44]. Wand explained that the errors shown by Algorithm W were usually indicating error locations far away from the actual source, which was a real struggle for programmers trying to diagnose their code. The reason for this issue was due to unification, which propagated the errors in subexpressions upwards the AST until unification was not possible anymore and only then reported this location as being problematic. The algorithm that Wand proposed returned all possible error locations instead of the last one that caused the issue, which was an improvement however not an optimal one since the number of errors reported could become large and cause confusion once again.

In 1998 Lee and Yi formalized Algorithm M [45] [3] - an analogous algorithm to W that worked top-down, rather than bottom-up [4]. In addition to formalizing and proving the algo-

[2] Requiring annotations might also lead to issues if subtyping is also desired, because now the compiler would have to verify whether the user supplied annotation is not more general than the inferred type (i.e. verifying that for every type allowed by annotation, it is also allowed by the inferred most general type), a problem known as subsumption checking [16].

[3] Lee and Yi noted that this algorithm was "folklore" and quite common in practice, however it was never properly formalized before their work

[4] Instead of taking an expression as input and producing an inferred type like Algorithm W, Algorithm M took an expression and a type and produced a set of substitutions to make this type compatible with an actual type of the expression. After applying the resultant set of substitutions (produced by algorithm M) to the input type, the same exact type would be obtained as W.

rithm, they proved an interesting property of M, which stated that M would always stop earlier than W in case of an error, resulting in detected error locations being closer to the actual source and easier to identify. They also noted that error localization could be further improved by using both algorithms M and W in conjuction and taking into account both of the generated error locations when determining the most probable one.

Other notable algorithms that took upon improving error localization of Algorithm W are SOLVE [17] proposed in 2002 by Heeren et al. and a more recent one by Pavlinovic et al. [18] proposed in 2015.

4.4 Bidirectional Type Checking

In addition to Algorithm W and its various extensions, another approach to type inference called Bidirectional Type Checking "has become one of the most popular techniques for implementing typecheckers in new languages" [46]. It utilizes a bidirectional approach in a sense that it combines type checking and type inference into one process. The reasoning for this is that some expressions might be "typeable in checking mode when it is not typable in inference mode" [47]. For example, consider Figure 3 which returns to the example "ex3" from Figure 1 that did not compile with a unidirectional type inference approach. It shows that this expression successfully compiles if an explicit annotation is provided and bidirectional type checking extension is enabled. This approach was first formalized by Pierce and Turner in 1999 [48] and has been applied by many various languages ever since, most notably in Scala. Type inference in Scala uses an improved version of Pierce and Turner's algorithm called "Colored Local Type Inference" as formalized in a paper by Scala's creator in 2001 [43] to allow for subtyping. Another notable algorithm that follows this technique is QuickLook (2020) [38] which was designed specifically to support first-class polymorphism and has been adopted by Haskell [49]. In 2021 Dunfield and Krishnaswami made an excellent survey analysing how this approach evolved over the years and mentioning many more bidirectional type checking algorithms that are not covered by our paper [46]. The remainder of this section will introduce the main techniques of this approach together with a few notable algorithms that implement these techniques.

Bidirectional type-checking can be classified as partial type inference (as opposed to Algorithm W which showcases complete type inference), meaning that explicit annotations are required for some expressions. In their original paper [48] Pierce and Turner motivated this choice by recognizing that complete type inference is really difficult to achieve for more complex features such as subtyping or impossible in the case of first-class polymorphism [11] and although there were numerous attempts in combining Hindley-Milner type inference with subtyping, there was no widespread adoption of these techniques in practice.

Furthermore, Pierce and Turner argued that complete type inference might not be as beneficial in practice as it would seem. For example, requiring type annotations for top-level function definitions could be seen as a form of documentation, showing the intent of the programmer. In their view,

```haskell
ex3' :: (forall a. a -> a) -> (Int, String)
ex3' = \f-> (f 5, f "text")
-- Compiles:
-- >>> ex3' id
-- (5,"text")
```

Figure 3: Another Haskell example, showing how a similar expression to Figure 1 compiles, given ImpredicativeTypes extension is enabled (QuickLook [38]) and explicit type annotation is provided.

inferring "common and silly" annotations should be enough in practice. Specifically, after analysing a large amount of code written in ML, they determined that the most important cases where types should be inferred are applications of polymorphic functions, parameters of anonymous functions and local bindings of variables. They argued that function definitions do not need to be inferred since from their results local function definitions are rare and (as noted) top-level function definitions serve as documentation.

Their technique takes advantage of explicit type annotations by propagating them downwards the AST (type-checking) and synthesizes unknown types carrying them upwards (type inference similar to Algorithm W). For this reason, the formal descriptions bidirectional type systems usually split the type system rules (such as seen in Figure 2 for Algorithm W) into two rules - inferring a type (when "child" types in AST are known) and checking (when "parent" types in AST are known) [43].

Subtyping

A notable bidirectional type checking algorithm for Subtyping is "Colored Local Type Inference" (2001) [43] which has been designed for Scala by Odersky et al. This algorithm borrows heavily from the original "Local Type Inference" (2000) [48], yet improves it considerably. Odersky noted that the original algorithm propagates types downwards in the AST (using type checking) only if they are fully known. If the types are only partially known, then the algorithm disables type-checking and relies solely on type inference (propagating types upwards in the AST). This means that given the type of f is (partially) known as

$$f : (Int \to \alpha) \to \alpha$$

the function application

$$f\ (\lambda x \to x + 1)$$

is ill-typed in the original algorithm [43], but not in algorithm by Odersky.

First-Class Polymorphism

Another noteworthy bidirectional type checking algorithm is QuickLook (2020) [38]. QuickLook focuses on combining first class polymorphism with type inference. One of the main strengths of QuickLook is its ease of integration with existing type inference systems. This is achieved due to their localized approach, where all first-class polymorphism is resolved before the constraint solving stage begins and thus a standard constraint solver can be used [50]. This is especially advantageous for complex production languages like Scala,

OCaml and Haskell, where any big changes to the existing type system are impractical. In fact, to backup their claims the authors of QuickLook have integrated their system with GHC (Haskell compiler) with only 1% change of the original codebase, preserving all the extensions that Haskell already implemented such as Type Classes and GADTs [38]. Furthermore, in 2021 the implementors of Haskell officially integrated QuickLook with Haskell as an optional "ImpredicativeTypes" extension, avalable as part of Haskell 9.21 [49]. In their documentation, it is noted that for the first time Haskell can enjoy a "robust" implementation of first-order polymorphism, unlike previous attempts which were unstable and unreliable.

5 Discussion and Advice

From the previous sections it should be apparent that there is a lot of research concerning type inference, resulting in a wide array of techniques, all of which have their own subtle nuances or restrictions that a developer of a new language should be aware of and unfortunately due to the scope of this paper it is not possible to cover each technique fully in as much detail as they require.

Thus, in this last section we wish to provide further discussion and advice for any new programming language developers regarding which techniques we believe are more suitable in what contexts, which we hope will help determining which techniques are worth investigating further based on the goals of the programming language developers.

First, we will have a general discussion between the Hindley-Milner and Bidirectional approaches, then we will address subtyping, first-class polymorphism and finally the quality of error messages.

Best Approaches Overall

As we have seen there are two main families type inference techniques - Hindley-Milner based and bidirectional. Choosing one over the other will have certain consequences for the language.

HM approaches usually support global type inference together with principal type property, which results in type inference being complete, where type annotations are never needed, but comes with a consequence of being inflexible. It is more difficult to add new features in HM than it is with bidirectional type checking, since each new feature might break the principal type property, rendering type inference unreliable as Odersky mentions for his motivations to not use HM approaches in Scala [43].

On the other hand, bidirectional type checking comes with greater freedom of type system features that one might add to programming language, however it means sacrificing complete type annotations, which might worsen the experience for programming in such language. Fortunately, as argued by Pierce [48], enforcing some annotations might be even desirable for some cases like top-level function definitions since they serve as documentation and help understanding the intent of the programmer.

Thus the choice of the type inference often comes down to whether more language expressiveness or more type inference is preferred. However, as certain bidirectional techniques such as QuickLook [38] have shown, it could be possible to combine both approaches into a hybrid system, where the bidirectional inference algorithm serves as a "preprocessing step" that reduces the program into one that is compatible with a HM based approach, in the end achieving best of both worlds - complete type inference (using HM algorithms) for simple programs and partial type inference (using bidiretional algorithms) when more complex features are involved.

Subtyping

Although there are HM based approaches that offer principal type inference under subtyping such as MLSub [15] or SimpleSub [16] they come at a cost of imposing certain restrictions to subtyping such as inability to support function overloading and are inefficient as noted by Melo [6] (some sources suggest that these algorithms might even be exponential [37]). Furthermore, we were unable to find any successful applications of these systems in practice. On the other hand, widespread languages like Scala serve as evidence that bidirectional type checking is a safe choice in the presence of subtyping. Thus we recommend choosing this approach as well.

First-Class Polymorphism

Since First-Class Polymorphism requires breaking the principal type property anyway, we believe that choosing a hybrid approach of HM together bidirectional algorithm like QuickLook [38] is the safest choice, which has also proven itself by being integrated in Haskell.

Good Error Messages

Finally, we believe that although the original Algorithm W did suffer from uninformative error messages, there are enough extensions addressing the problem like Algorithm M [45], SOLVE [17] or the method proposed by Pavlinovic et al. [18] that in our opinion makes it a tie between HM based approaches and bidirectional type checking, where other aspects such as completeness of type inference or expressiveness of the language should sway the choice one way or the other.

6 Conclusion

This paper presented a literature survey of type inference techniques for statically typed languages by analysing a wide range of research papers concerning this topic. Two main families of algorithms for type inference have been established - Hindley-Milner based and bidirectional type checking. The paper broke down each technique, presenting most notable algorithms belonging to each approach and intuitions behind them. Furthermore, the different algorithms where contrasted and compared, identifying their unique advantages and weaknesses and where possible presenting their application in real-world programming languages. Finally, the paper presented some advice and discussion about the presented techniques with the aim of helping programming language developers make the right choice of type inference algorithms when designing a new language.

References

[1] J. Siek and M. Vachharajani, "Gradual typing with unification-based inference," p. 7, July 2008.

[2] E. Meijer and P. Drayton, "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages," Jan. 2004.

[3] L. Damas and R. Milner, "Principal type-schemes for functional programs," *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212, Jan. 1982.

[4] F. Emrich, S. Lindley, J. Stolarek, J. Cheney, and J. Coates, *FreezeML: Complete and Easy Type Inference for First-Class Polymorphism*. Apr. 2020.

[5] B. Wagner, "Type Inference." https://download.microsoft.com/download/5/4/B/54B83DFE-D7AA-4155-9687-B0CF58FF65D7/type-inference.pdf, May 2012.

[6] L. T. C. Melo, R. G. Ribeiro, B. C. F. Guimarães, and F. M. Q. Pereira, "Type Inference for C: Applications to the Static Analysis of Incomplete Programs," *ACM Transactions on Programming Languages and Systems*, vol. 42, pp. 1–71, Sept. 2020.

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

[8] J. Hindley, "The principal type-scheme of an object in Combinatory Logic," *Trans AMS*, vol. 146, pp. 29–60, Jan. 1969.

[9] D. Leijen, "HMF: simple type inference for first-class polymorphism," *ACM SIGPLAN Notices*, vol. 43, pp. 283–294, Sept. 2008.

[10] M. Odersky and K. Laufer, "Putting Type Annotations to Work," *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ser. POPL*, Jan. 1996.

[11] J. B. Wells, "Typability and type checking in System F are equivalent and undecidable," *Annals of Pure and Applied Logic*, vol. 98, pp. 111–156, June 1999.

[12] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM*, vol. 12, pp. 23–41, Jan. 1965.

[13] D. Vytiniotis, S. P. Jones, T. Schrijvers, and M. Sulzmann, "OutsideIn(X) Modular type inference with local assumptions," *Journal of Functional Programming*, vol. 21, pp. 333–412, Sept. 2011. Publisher: Cambridge University Press.

[14] D. Le Botlan and D. Rémy, "ML $^{\mathrm{f}}$: raising ML to the power of system F," in *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, (Uppsala Sweden), pp. 27–38, ACM, Aug. 2003.

[15] S. Dolan and A. Mycroft, "Polymorphism, subtyping, and type inference in MLsub," *ACM SIGPLAN Notices*, vol. 52, pp. 60–72, Jan. 2017.

[16] L. Parreaux, "The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl)," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–28, Aug. 2020.

[17] B. Heeren, J. Hage, and S. Swierstra, "Generalizing Hindley-Milner Type Inference Algorithms," Aug. 2002.

[18] Z. Pavlinovic, T. King, and T. Wies, "On Practical SMT-Based Type Error Localization," Aug. 2015. arXiv:1508.06836 [cs].

[19] W. R. Cook, W. Hill, and P. S. Canning, "Inheritance is not subtyping," in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, (New York, NY, USA), pp. 125–135, Association for Computing Machinery, Dec. 1989.

[20] M. A. AbdelGawad, "Why Nominal-Typing Matters in OOP," Dec. 2017. arXiv:1606.03809 [cs].

[21] F. Pottier, "Simplifying subtyping constraints," in *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, (New York, NY, USA), pp. 122–133, Association for Computing Machinery, June 1996.

[22] J. Niehren and T. Priesnitz, "Non-Structural Subtype Entailment in Automata Theory," 2003.

[23] F. Pottier, "Simplifying Subtyping Constraints: A Theory," *Information and Computation*, vol. 170, pp. 153–183, Nov. 2001.

[24] J. C. Mitchell, "Coercion and type inference," in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, (New York, NY, USA), pp. 175–185, Association for Computing Machinery, Jan. 1984.

[25] R. Stansifer, "Type inference with subtypes," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, (New York, NY, USA), pp. 88–97, Association for Computing Machinery, Jan. 1988.

[26] Y.-C. Fuh and P. Mishra, "Polymorphic Subtype Inference: Closing the Theory-Practice Gap," in *Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages*, TAPSOFT '89, (Berlin, Heidelberg), pp. 167–183, Springer-Verlag, Mar. 1989.

[27] S. Kaes, "Parametric overloading in polymorphic programming languages," in *ESOP '88* (H. Ganzinger, ed.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 131–144, Springer, 1988.

[28] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th*

ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '89, (New York, NY, USA), pp. 60–76, Association for Computing Machinery, Jan. 1989.

[29] G. S. Smith, "Principal type schemes for functional programs with overloading and subtyping," *Science of Computer Programming*, vol. 23, pp. 197–226, Dec. 1994.

[30] A. Aiken and E. L. Wimmers, "Type inclusion constraints and type inference," in *Proceedings of the conference on Functional programming languages and computer architecture*, (Copenhagen Denmark), pp. 31–41, ACM, July 1993.

[31] F. Pottier, "A framework for type inference with subtyping," in *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, (New York, NY, USA), pp. 228–238, Association for Computing Machinery, Sept. 1998.

[32] F. Pottier, *Type Inference in the Presence of Subtyping: from Theory to Practice*. report, INRIA, 1998.

[33] S. Dolan, "MLsub Demo." https://web.archive.org/web/20200429043036/https://www.cl.cam.ac.uk/~sd601/mlsub/, Apr. 2020.

[34] L. Parreaux, "Simple-sub Demo." https://lptk.github.io/simple-sub/.

[35] S. Dolan, "MLsub Source Code." https://github.com/stedolan/mlsub, June 2023. original-date: 2014-11-17T20:46:32Z.

[36] L. Parreaux, "Simple-sub Source Code." https://github.com/LPTK/simple-sub, May 2023. original-date: 2020-02-12T12:25:10Z.

[37] E. Jones and S. Ramsay, *Intensional Datatype Refinement*. Aug. 2020.

[38] A. Serrano, J. Hage, S. Peyton Jones, and D. Vytiniotis, "A quick look at impredicativity," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 89:1–89:29, Aug. 2020.

[39] F. Henglein and J. Rehof, "The complexity of subtype entailment for simple types," pp. 352–361, Aug. 1998.

[40] L. Augustsson, "Impredicative polymorphism, a use case." http://augustss.blogspot.com/2011/07/impredicative-polymorphism-use-case-in.html, 2011.

[41] "OOP vs type classes - HaskellWiki." https://wiki.haskell.org/OOP_vs_type_classes#Type_classes_are_like_interfaces.2Fabstract_classes.2C_not_classes_itself.

[42] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler, "Type classes in Haskell," *ACM Transactions on Programming Languages and Systems*, vol. 18, pp. 109–138, Mar. 1996.

[43] M. Odersky, C. Zenger, and M. Zenger, "Colored local type inference," in *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (London United Kingdom), pp. 41–53, ACM, Jan. 2001.

[44] M. Wand, "Finding the source of type errors," in *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '86, (New York, NY, USA), pp. 38–43, Association for Computing Machinery, Jan. 1986.

[45] O. Lee and K. Yi, "Proofs about a folklore let-polymorphic type inference algorithm," *ACM Transactions on Programming Languages and Systems*, vol. 20, pp. 707–723, July 1998.

[46] J. Dunfield and N. Krishnaswami, "Bidirectional Typing," *ACM Computing Surveys*, vol. 54, pp. 1–38, May 2021.

[47] S. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields, "Practical type inference for arbitrary-rank types," *Journal of Functional Programming*, vol. 17, pp. 1–82, Jan. 2007. Publisher: Cambridge University Press.

[48] B. C. Pierce and D. N. Turner, "Local type inference," *ACM Transactions on Programming Languages and Systems*, vol. 22, pp. 1–44, Jan. 2000.

[49] "6.4.18. Impredicative polymorphism — Glasgow Haskell Compiler 9.7.20230603 User's Guide." https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/impredicative_types.html, Oct. 2021.

[50] F. Emrich, J. Stolarek, J. Cheney, and S. Lindley, "Constraint-based type inference for FreezeML," July 2022. arXiv:2207.09914 [cs].