IB Computer Science

Extended Essay

Topic:

Investigating type inference for object-oriented programming languages through the lambda calculus.

Research Question:

To what extent is type inference possible for statically typed polymorphic object-oriented programming languages?

Word count: 3996

Candidate Code: `jky029`

Contents

1 Introduction

1.1 Background

The design and implementation of programming languages has come a long way since the use of machine code for directly programming computers. These improvements have allowed the development of software systems larger than ever before, without sacrificing on correctness or robustness. **Type systems** are a vital part of programming language design that have allowed computer programmers to meet the demands for correct and properly functional programs. One definition of type systems for programming languages would be: "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute" (Pierce 1). In other words, type systems assign a *type* to different expressions and constructs in a programming language based on properties such as the values they compute. The type system can then reason about the behavior of a program by methodically looking at how different types interact with each other, and then based on the rules of the type system, decide whether the program is well-behaved (it adheres to the rules of the system) or not. As an example, the following Java program fails to compile due to an error in the type system:

Listing 1: Ill-behaved Java program

```
int x = 1;
String y = "2";
// error: bad operand types for binary operator '/'
System.out.println(x / y);
```

While the error is obvious in this example, in larger codebases it is not always easy for the programmer to notice small errors. This is where the type checker comes in. The job of a type checker (which is usually included

in the compiler itself) is to ensure that the source code adheres to the rules of the type system.

1.2 Type Inference

Due to its vast benefits, most compiled languages offer some form of type checking in order to aid the development of software. However, type checking has certain drawbacks that make it an inconvenient feature for programmers in certain situations. One major drawback is that sound type checking requires that all expressions appearing in the source code must belong to a certain type. In order to satisfy this requirement, the programmer has to manually annotate every expression with its type, such as in the first two lines of Listing 1. This can often lead to overly verbose code and hinder productivity. For example, one criticism of Java that influenced the development of the Kotlin programming language was its verbosity, which in turn leads to poor readability (Breslav).

The solution to this is to have the compiler analyze a program to automatically infer the types of expressions appearing in it. This is known as **type inference** (Krishnamurthi). Type inference eliminates the need for programmers to explicitly annotate expressions. Moreover, type inference can be integrated within tools such as Integrated Development Environments (IDEs) to further aid development by, for example, providing documentation or catching errors before executing the program. Due to its close ties with the type system, powerful type inference is often also indicative of a powerful type system.

1.3 Type Inference for OOP Languages

Languages belonging to the functional programming (FP) paradigm, such as Standard ML or Haskell, have included type inference as a feature for

a long time. However, it has been largely absent from most common object-oriented programming (OOP) languages for quite some time, and has only recently started to become a common feature of some (Melo). This is because the strong type systems of FP languages lend themselves well to type inference, whereas for OOP languages certain features (in particular, polymorphism) make type inference a much more difficult task.

Given the benefits of type inference and type checking, and considering the popularity of OOP languages, the question of to what extent type inference is possible for statically typed polymorphic OOP languages is an interesting one which I was interested to study further.

Note that because OOP is merely a paradigm, languages are free to choose how closely they adhere to OOP principles. As such, this essay does not focus on any particular language. Rather, it develops the minimal lambda calculus (λ-calculus) for reasoning about programming languages, and later extends it with features of polymorphism common (but not strictly unique) to OOP languages. This is then used to explore and understand barriers to type inference in OOP languages.

2 The Lambda Calculus

2.1 Untyped λ-calculus

The untyped λ-calculus (Church, *Introduction to Mathematical Logic*; *The Calculi of Lambda-Conversion*) is a minimal yet Turing-complete programming language that can be used to model computation, using only function abstraction and application. Its usefulness comes from the fact that it can be considered not only as a programming language, but also a formal system for making and proving logical statements (Pierce). It is an important tool in programming language design, and will also help us in exploring type inference for OOP languages. In order to illustrate its use, we will begin with the untyped λ-calculus and extend it to obtain the Simply-Typed Lambda Calculus (STLC).

The syntax of the untyped λ-calculus is as follows, using Backus-Naur form (BNF) notation (see Appendix A for a summary of this notation):

`t ::=`	*terms:*
x	*variable*
$\mid \lambda x.$`t`	*abstraction*
`|` `t t`	*application*
`v ::=`	*values:*
$\lambda x.$`t`	*abstraction value*

As shown above, the syntax comprises of just three terms. Variables, such as x are terms; abstraction of a variable x over a term `t` (this is akin to a function definition with one parameter x, which returns `t`); and application of a term to another term `t` (this is akin to function application). The only "value" in λ-calculus (that is, an expression that cannot be evaluated further) is abstraction itself. Note that unlike other programming languages, λ-calculus does not have any built-in constants or primitives such as numbers or conditionals.

"Computation" is reflected by evaluating the terms of an expression to obtain a simpler expression. The primary evaluation rule in λ-calculus is an evaluation of function application. If an expression contains a function application of some term $\texttt{t}_1$ to a lambda abstraction $\texttt{t}_2$, then this can be evaluated by replacing all occurrences of the abstraction variable in $\texttt{t}_2$ with $\texttt{t}_1$. This substitution is written as $[x \mapsto \texttt{t}_1]\texttt{t}_2$, which reads as "replace all free occurrences of x in $\texttt{t}_2$ by $\texttt{t}_1$" (Pierce). For example, the term $(\lambda x.x)y$ consists of an application of the variable y to the function $\lambda x.x$. This function simply returns the argument provided, and so the term $(\lambda x.x)y$ would evaluate to just y.

This evaluation rule can be more formally expressed using inference rules (see Appendix A for a summary of this notation):

$$\text{E-App1: } \frac{\texttt{t}_1\ \texttt{t}_2}{\texttt{t}_1 \longrightarrow \texttt{t'}_1} \qquad \text{E-App2: } \frac{\texttt{v}_1\ \texttt{t}_2}{\texttt{t}_2 \longrightarrow \texttt{t'}_2}$$

$$\text{E-AppAbs: } (\lambda x.\texttt{t}_{12})\texttt{v}_2 \longrightarrow [x \mapsto \texttt{v}_2]\texttt{t}_{12}$$

The notation $\texttt{t} \longrightarrow \texttt{t'}$ means that the term $\texttt{t}$ can be evaluated to $\texttt{t'}$. Essentially, "$\longrightarrow$" represents a 'computation' step. Here, the rule **E-App1** tells us that the term $\texttt{t}_1\ \texttt{t}_2$, where $\texttt{t}_1 \longrightarrow \texttt{t'}_1$, evaluates to $\texttt{t'}_1\ \texttt{t}_2$. Although it appears to be obvious, it is important because it specifies the order of computation: before the function application $\texttt{t}_1\ \texttt{t}_2$ can be performed, the term $\texttt{t}_1$ needs to be fully evaluated.

Similarly, the rule **E-App2** tells us that the expression $\texttt{v}_1\ \texttt{t}_2$, where $\texttt{t}_2 \longrightarrow \texttt{t'}_2$, evaluates to $\texttt{v}_1\ \texttt{t'}_2$. Here, the (meta-)variable $\texttt{v}_1$ stands for a *value* rather than a term, meaning that it cannot be evaluated any further. In other words, before the function application $\texttt{v}_1\ \texttt{t}_2$ can be performed, the term $\texttt{t}_2$ needs to be fully evaluated. Finally, the rule **E-AppAbs** tells us how to perform the function application, which is to perform a substitution.

2.2 Simply-Typed Lambda Calculus

λ-calculus can be extended with types to obtain the STLC. To demonstrate typing, we will add the boolean type to it. First, we extend the syntax:

`t ::=`	*terms:*
...	*(previous terms)*
| true	*true*
| false	*false*
$\mid \lambda x : \tau.\texttt{t}$	*lambda abstraction*
`v ::=`	*values:*
...	*(previous values)*
| true	*true value*
| false	*false value*
τ `::=`	*types:*
$\tau \to \tau'$	*function type*
| `Bool`	*boolean type*

We have added the two new boolean terms, true and false, both of which are values as well. Aside from that, the lambda abstraction term is also different now; rather than simply x, the argument is now written $x : \tau$. The new symbol τ ranges over the types available in the STLC: the function type $\tau \to \tau'$, and the newly-added `Bool` type. An example of a concrete function type would be `Bool` $\to$ `Bool`. An example of a value belonging to this type would be $(\lambda x : \texttt{Bool}.x)$.

We can now use typing rules to define the behavior of our new terms and types (see Appendix A for a summary of this notation):

$$\text{T-True: } \frac{}{\vdash true : \texttt{Bool}} \qquad \text{T-False: } \frac{}{\vdash false : \texttt{Bool}} \qquad \text{T-Var: } \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\text{T-Abs: } \frac{\Gamma, x : \tau \vdash \mathtt{t} : \tau'}{\Gamma \vdash \lambda x : \tau.\mathtt{t} : \tau \to \tau'} \qquad \text{T-App: } \frac{\Gamma \vdash \mathtt{t}_1 : \tau \to \tau' \quad \Gamma \vdash \mathtt{t}_2 : \tau}{\Gamma \vdash \mathtt{t}_1 \mathtt{t}_2 : \tau'}$$

The first three rules are straightforward: true and false are of type `Bool`, and if $x : \tau$ is in the context Γ, then x is of type τ under that context. The rule **T-Abs** describes typing for lambda abstractions: the abstraction variable $x : \tau$ is added to the context Γ (because the variable might appear in the term `t`), and given that the term `t` is of type τ', then we can conclude that a lambda abstraction of the form $(\lambda x : \tau.\mathtt{t})$ is of type $\tau \to \tau'$ (since `t` is the return value of the abstraction). **T-App** states that a term of type $\tau \to \tau'$ can be applied to a term of type τ, resulting in a term of type τ'.

In this manner, by extending the syntax and grammar to add new terms and types, and defining typing rules for the behavior of those terms and types, we can extend the STLC to add new features to it. Before extending it with OOP features, we will first consider how type inference may be performed on the STLC developed so far.

2.3 Type Inference as Constraint Satisfaction

One common technique for performing type inference on the STLC (and FP languages derived from STLC) is to model it as a Constraint Satisfaction Problem. A *constraint* between two types τ and τ' (denoted $\tau \sim \tau'$) simply states that the two types must be *unified*, or in other words, they should be equal to each other. A *constraint set* (usually denoted C) is a list of such constraints for a given program (where type annotations may be partially or completely absent). A *unification* function is one which generates a substitution that satisfies all constraints in C (Suidman, "Introduction to Type Systems: Type Inference"). In other words, it finds a solution to the constraint satisfaction problem.

For example, consider the following function written without type anno-

tations (in a common FP language):

Listing 2: Constraint-based Inference

```
f x = x + x
```

The job of the inference algorithm is to infer the type of the variable x and the function f. First, because we do not yet know the types of these terms, we will assign type variables to them, so that $x : \tau_0$ and $f : \tau_1$. Next, we analyze the body of the function, which consists of a single operation, $x + x$. We assign this expression a type variable as well, so that $(x + x) : \tau_2$. Since f is a function whose argument is x and return value is $(x + x)$, we can generate the constraint $\tau_1 \sim (\tau_0 \rightarrow \tau_2)$. We know that the operator '+' takes two values of type `Int`, so we can generate the constraint $\tau_0 \sim$ `Int` (in fact, we would generate two constraints for both the left-hand and right-hand side of the operator. However, this is a special case since the same variable appears on both sides). We also know that the '+' operator returns a value of type `Int` (the arithmetic sum of the left and right hand sides), so we have the constraint $\tau_2 \sim$ `Int`. Altogether, we have the following constraints:

$$C = \{\tau_1 \sim (\tau_0 \rightarrow \tau_2), \tau_0 \sim \texttt{Int}, \tau_2 \sim \texttt{Int}\}$$

To solve these, we can substitute the type `Int` for τ_0 and τ_2 to satisfy all constraints. Since $x : \tau_0$ and $f : \tau_1$, we can infer that $x :$ `Int` and $f :$ `Int` $\rightarrow$ `Int`.

A constraint-based inference algorithm is one which can be employed to generate constraints in this manner and perform unification to infer types (Krishnamurthi, Lerner, and Politz). An example is the Damas-Hindley-Milner algorithm (Damas; Hindley; Milner), which forms the basis of most inference algorithms used in statically typed FP languages. One important characteristic of Damas-Hindley-Milner inference is its completeness - it can infer the types of *all* terms within a given program, without any annotations or hints. This is known as *global* or *full* type inference, as opposed

to *partial* type inference which can only infer the types of some terms. The other notable characteristic of Damas-Hindley-Milner inference is that it infers the *principal type*, that is, the most general type that encompasses all possible types for a given expression. Both properties are considered desirable in any type system.

3 Features of Object-Oriented Programming

One feature common to many OOP languages (such as Java, Scala, TypeScript, Kotlin, etc.) is polymorphism. Polymorphism is commonly defined as types (or entities) whose operations are applicable to values of more than one type (Cardelli and Wegner). Interestingly, this definition is broad enough to include most of the distinguishing features of typed OOP languages, such as generic programming, subtyping and operator/function overloading, as we will see later.

This definition, however, is too broad. Polymorphism can be further divided into different forms with more precise definitions, allowing us to better understand how they may act as barriers to type inference. In particular, three forms of polymorphism are discussed: parametric, subtype and ad-hoc polymorphism.

4 Parametric Polymorphism

4.1 System F

Parametric polymorphism is when a data type, such as a function, can be written generically such that it can handle values independent of their type (Pierce). Such types are also known as generic data types. The following is an example of parametric polymorphism in Java:

Listing 3: Generic Programming in Java

```
public <T> ArrayList<T> wrap(T value) {
    ArrayList<T> list = new ArrayList<T>();
    list.add(value);
    return list;
}
```

This function simply wraps a value of type `T` into an `ArrayList` and returns the list. Note that the `ArrayList` data structure is itself a generic data type. This demonstrates the expressiveness and power of parametric polymorphism: functions no longer need to be bound by any one specific type. Rather, they can be expressed for any type `T`. We call `T` a *type variable*.

STLC can be extended to support parametric polymorphism. STLC with parametric polymorphism is also known as System F (Girard; Reynolds). We have seen in the example above that parametric polymorphism is simply the introduction of a special variable that ranges over *types* instead of *terms*. Thus, the formulation of System F can be achieved by introducing a new form of abstraction that takes a type variable as its argument, and returns a concrete type formed using that variable. A complete understanding of System F is not necessary to understand the following section (although for completeness, the formal syntax and rules can be found in Appendix B).

4.2 Type Inference in System F

The concepts of reducibility and undecidability are important to understand why type inference is difficult (in fact, impossible) in System F. A problem A is **reducible** to B if there is a way to convert any given instance of A to an instance of B. A problem is **undecidable** if there is no general algorithm to determine the answer for a given instance of the problem. Furthermore, if A is reducible to B, and B is undecidable, then A is also undecidable (Hopcroft, Motwani, and Ullman).

It has been proven that, in System F, the problem of type inference is undecidable. This was proven by Wells by showing that type inference in System F can be reduced to another problem known as the semi-unification problem, which had already been proven to be undecidable (Kfoury, Tiuryn, and Urzyczyn). While the proof itself is beyond the scope of this essay, this reduction implies that global type inference is not possible in a type system that supports parametric polymorphism. Not only that, it has also been shown that in many cases, even partial type inference is undecidable for System F (Boehm, "Partial polymorphic type inference is undecidable"; "Type Inference in the Presence of Type Abstraction").

Given the benefits and expressiveness of parametric polymorphism, most typed OOP languages choose to give up on global type inference in favor of this feature. Other languages, such as Haskell, allow a restricted form of parametric polymorphism where type inference is still possible (Pierce).

5 Subtype Polymorphism

5.1 Subtyping

Subtyping is a major characteristic of the OOP paradigm. Like parametric polymorphism, it allows writing code in a more abstract manner. For example, consider the following Java program:

Listing 4: Subtyping in Java

```
public class Main {
   public static void main(String[] args) {
      byte x = 100;
      short y = 1000;
      int result = add(x, y);
      System.out.println(result); // outputs 1100
   }

   public static int add(int x, int y) {
      return x + y;
   }
}
```

Although the method `add` is defined for two parameters of type `int`, it works for `byte` and `short` as well. This is because `byte` and `short` are both *subtypes* of the *supertype* `int`. This means that both `byte` and `short` (and all other subtypes of `int`) can be substituted for `int` without compromising on the correctness of the program. This particular subtyping relation is commonly known as the Liskov substitution principle (Liskov and Wing), and is an important design principle of OOP. More generally, a type `S` is a subtype of some type `T` if any term of `S` can safely be used in a context where a term of `T` is expected (Pierce).

5.2 Record Types

Subtype polymorphism is specifically important in OOP because of how it affects the way objects and object types behave. Commonly, an *object* is a "data structure encapsulating some internal *state* and offering access to this state to clients via a collection of *methods*" (Pierce 228). To better understand subtype polymorphism through the STLC under this context, we first need to extend it with a datatype similar to these objects. To do so, we will add the record type to the STLC. Records are simply a collection of terms identified by some label. Although proper object types in more complete programming languages are more complex and nuanced than this, it is sufficient for our purposes.

First, the syntax for record literals and accessing fields of a record (projection):

`t ::=`	*terms:*
...	*(previous terms)*
$\mid$ `{`$\mathtt{l}_i$`=`$\mathtt{t}_i$`}`$^{i \in 1..n}$	*record*
$\mid$ `t.l`	*projection*
`v ::=`	*values:*
...	*(previous values)*
$\mid$ `{`$\mathtt{l}_i$`=`$\mathtt{v}_i$`}`$^{i \in 1..n}$	*record value*
τ `::=`	*types:*
...	*(previous types)*
$\mid$ `{`$\mathtt{l}_i$`:`τ_i`}`$^{i \in 1..n}$	*record type*

The notation `{`$\mathtt{l}_i$`=`$\mathtt{t}_i$`}`$^{i \in 1..n}$ is used for a record with n fields, each uniquely labeled. The remaining grammar is straightforward. An example of a record using this syntax would be `{x=true, y=false}`. Relevant evaluation rules can be found in Appendix C.

The relevant typing rules for records are as follows:

$$\text{T-Rcd:}\ \frac{\text{for each } i\ \ \Gamma \vdash \mathtt{t_i} : \tau_i}{\Gamma \vdash \{\mathtt{l}_i{=}\mathtt{t}_i{}^{\,i \in 1..n}\} : \{\mathtt{l}_i{:}\tau_i{}^{\,i \in 1..n}\}} \qquad \text{T-Proj:}\ \frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i{:}\tau_i{}^{\,i \in 1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{l}_j : \tau_j}$$

The rule **T-Rcd** tells us that a record $\{\mathtt{l}_i{=}\mathtt{t}_i{}^{\,i \in 1..n}\}$ where each term of a field $\mathtt{t}_i$ has some type τ_i will have the type $\{\mathtt{l}_i{:}\tau_i{}^{\,i \in 1..n}\}$. For example, the record `{x=true, y=false}` has two fields `x` and `y` of type `Bool`. The type of this record would then be `{x:Bool, y:Bool}`. The rule **T-Proj** simply tells us that the type of a projection will be the type of the corresponding field.

For convenience, we will make use of `let`-syntax to bind terms to a name, like so: `let id` $= \lambda x.x$, and similarly `type` to bind types to a name.

With the addition of records, the benefits of subtyping in STLC become more apparent. Consider the following function:

$$\texttt{let f} = (\lambda r : \{\texttt{x:Bool}\}.\ r.\texttt{x})$$

The function `f` takes a record containing the field x : `Bool` and returns the value of that field. However, the following well-behaved term would be considered invalid under the typing rules for function application (see rule **T-App** in subsection 2.2):

```
f {x:true, y:false}
```

The above application fails because, according to our typing rules, the function `f` can only take terms of the type `{x:Bool}`, while in this case it is being applied to the type `{x:Bool, y:Bool}`. Clearly, the term is well-behaved, because it only requires the argument to have a field `x` of type `Bool`. To overcome this, we introduce subtyping in the lambda calculus.

5.3 Rule of Subsumption

Although there is much more to the formalization of subtyping in lambda calculi, in our case we only focus on the aspects relevant to our discussion, namely the subsumption rule. We will assume that a record type `S` is a subtype of some other record type `T` if it contains *at least* all the fields contained in `T`. This is denoted as `S <: T`. Then, we can add the rule of subsumption:

$$\text{T-Sub:}\ \frac{\Gamma \vdash \texttt{t} : \texttt{S} \qquad \texttt{S} <: \texttt{T}}{\Gamma \vdash \texttt{t} : \texttt{T}}$$

This rule simply tells us that a term of some subtype `S` also belongs to its supertype `T`. For example, `{x:true, y:false}` belongs to both `{x:Bool, y:Bool}` *and* `{x:Bool}`, because `{x:Bool, y:Bool} <: {x:Bool}`. This solves the problem encountered in the previous section, because now the function `f` can accept not only the type `{x:Bool}`, but also all of its subtypes.

5.4 Inference under Subtyping

A closer look at the subsumption rule also gives us an idea of why inference may be problematic under the presence of subtyping: the term `t` now belongs to not just the type `S`, but also all subtypes of `S`. In other words, the same term belongs to multiple types.

Let us consider the function `f` defined earlier, but this time without any type annotations:

$$\texttt{let f = } (\lambda r.r.\texttt{x})$$

Now, due to the rule of subsumption, the variable r could belong to any of an infinite number of types. Given this program, without any type

annotations from the programmer, the most we can infer is that `f` is a function type that takes some record type as its argument, which contains a field `x`. Needless to say, this is not very practical or useful.

The kind of subtyping discussed so far falls largely under the notion of *structural subtyping*, where the subtype relation between two types is based entirely on their *structure* (for example, the type and number of fields in two record types). A different notion is that of *nominal* subtyping, where a subtype relation may only exist if the programmer explicitly declares one so (Pierce). This is a common feature in many mainstram OOP languages, and usually includes special syntax as well, for example the `implements` keyword in Java. Under nominal subtyping, inference becomes even more difficult. Consider the following example:

```
type A = {x:Bool}
type B = {x:Bool}
let f = (λr.r.x)
```

In this example, even though the types `A` and `B` are identical, they are seen as entirely different types under nominal subtyping, because we did not declare any subtype relation between them. Thus, even if we were able to infer that r is of type `{x:Bool}`, it would be impossible to determine whether it is of type `A` or `B` without explicit annotations from the programmer.

6 Ad-Hoc Polymorphism

Ad-Hoc polymorphism allows overloading of functions and operators to different types (Pierce). A common example would be overloading the '+' operator to work on both integers (as an arithmetic operator) as well as String types (as a concatenation operator).

The issues this poses to type inference are straightforward. Consider the function from Listing 2:

```
f x = x + x
```

Assuming the '+' operator is overloaded for both integers and Strings, both of the following lines are valid:

```
f 1 // == 2
f "hi" // == "hihi"
```

The function `f` can be applied to both `integers` and `strings`. In other words, the variable `x` can now belong to two different types depending on the argument provided. This is similar to the problem faced with subtype polymorphism, making type inference difficult.

Ad-hoc polymorphism is another feature popular with OOP languages, but can also be found in some strongly typed functional programming languages, such as Haskell. These languages introduce more complex constructs such as typeclasses (O'Sullivan, Stewart, and Goerzen) that allow overloading of functions without sacrificing type inference. However, typeclasses are not very suitable for type systems supporting the OOP paradigm.

7 Alternative Techniques

Despite these challenges, there are several techniques for performing (partial) type inference in OOP languages. One increasingly common technique is *local type inference* (Pierce and Turner). Compared to the full type inference of Damas-Hindley-Milner, local type inference is very much limited. This technique recovers type information from adjacent nodes of the abstract syntax tree (the internal representation of a program, as a tree structure) after the parsing stage of the compiler (Pierce), when possible. This allows it to perform simple inferences, such as in variable declarations. Numerous modern OOP languages, such as Java, Scala, Visual Basic, C# etc. use a form of local type inference. For example, Java introduced the `var` keyword which enables local type inference in variable declarations, so that code such as the following:

```
HashMap<String, String> map = new HashMap<String, String>();
```

Can be shortened to:

```
var map = new HashMap<String, String>();
```

In such cases, the inference engine can observe the type of the expression on the right-hand side of the assignment operator (provided the expression is simple enough) and assign it to the variable on the left-hand side, without requiring any annotations for it. Even though local type inference lacks completeness and the principal type property, it can still provide surprisingly sufficient inference in many cases, and reduce verbosity.

Aside from this, other more powerful techniques have also been developed. For example, *bidirectional type inference* (Pierce and Turner), implemented in the Swift programming language (Suter). In this technique, type information is propagated further in both the backwards and forwards direction from the nodes of a syntax tree, allowing for more powerful type inference.

Algebraic subtyping (Dolan) is another powerful type inference technique that allows inference under the presence of subtyping. Although type inference for simple subtypes had been possible to some extent (Mitchell), algebraic subtyping is notable in that it extends Damas-Hindley-Milner inference to provide full support for subtyping, while retaining the principal type property (Parreaux).

Although a detailed discussion of these techniques is beyond the scope of this essay, they highlight the possibilities of type inference in OOP languages.

8 Conclusion

In conclusion, global type inference for statically typed OOP languages supporting parametric polymorphism is not possible, while global type inference for OOP languages supporting ad-hoc or subtype polymorphism is possible but complex. However, simple partial type inference for typed polymorphic OOP languages is still possible to a large extent, through techniques such as local type inference and its variants. In most cases, this partial type inference is sufficient to a large degree.

In fact, it is important to note that while type inference has many benefits in terms of programming language design, there are also cases where (global) type inference may not be desirable. Even in languages that do support global type inference, it is often discouraged in practical settings. This is due to the fact that type annotations can also serve the purpose of documenting code, and making it easier to understand. Moreover, type annotations also aid the compiler in analyzing the code and providing more useful and readable error messages (as opposed to error messages containing obscure type variables that are not very helpful to the programmer) (Pierce and Turner). In these cases, type annotations are encouraged, while type inference is relied upon only when code becomes verbose or unreadable. This situation is ideal for partial type inference techniques, making them quite useful and practical.

9 Appendices

9.1 Appendix A: Notation

Backus-Naur Form

BNF is a commonly used notation for expressing the syntax of programming languages (called a **grammar**). Essentially, it is a set of rules that can be used to generate strings that follow the syntax defined by the grammar (or in other words, syntactically valid strings) (Nystrom). BNF consists of:

- **Terminal** symbols, which are literal values of the string generated.
- **Non-terminal** symbols, which are used to reference other rules of the grammar.
- **Productions**, which are the rules themselves. Each production is of the form LHS ::= RHS;, where the LHS is a non-terminal symbol, and the RHS is a sequence of either terminal or non-terminal symbols.

For example, the following could be a BNF grammar defining the signature for methods in a Java-like programming language:

```
        Sig  ::= Visibility Access Type Name '(' Params ');' ;
Visibility   ::= 'public' | 'protected' | 'private' ;
    Access   ::= 'static' | '' ;
      Type   ::= 'boolean' | 'byte' | 'char' | 'int' | 'float' ;
      Name   ::= 'A' | ...  | 'Z' | 'a' | ...  | 'z' | Name ;
    Params   ::= Name | Name ',' Params ;
```

The pipe symbol ('|'), read as 'or', separates the sequences of a production. We are allowed to choose whichever we want. Terminal symbols are enclosed within quotes to differentiate them from non-terminal symbols.

Note that this change is for simplicity only; in the grammar for λ-calculus (and use of this notation elsewhere in this essay), quotes do not enclose non-terminal symbols.

To generate a string from this grammar, we start from the rule `Sig`. It tells us to go to the rule `Visibility`, which has three non-terminals. We choose, for example, the first one, 'public'. Now our string is: `public Access Type Name '(' Params ');'`. We then look at `Access`, which is either 'static' or the empty string. Choosing the first one again, we have: `public static Type Name '(' Params ');'`. Similarly for `Type`, if we choose 'boolean', we then have: `public static boolean Name '(' Params ');'`. For `Name`, notice that it contains the entire english alphabet, and then recursively references itself. This allows us to generate any sequence of letters to form a name. We can choose, say, 'isEqual'. We thus have: `public static boolean isEqual '(' Params ');'`. For `Params`, we can have either one `Name`, or a recursively generated sequence of `Names` separated by a comma, for example 'a, b'. We then have: `public static boolean isEqual '(' a, b ');'`. Putting it together, we get: `public static boolean isEqual(a, b);`.

Inference Rules

In logic, inference rules are a form of syntactic expressions which take a number of *premises*, written above a horizontal bar, and return a conclusion based on those premises (Suidman, "Introduction to Type Systems: Simply Typed Lambda Calculus"). The prime example is the *modus ponens*, which takes two premises, p and $p \implies q$ (if p then q), to return the conclusion, q:

$$\text{modus ponens: } \frac{p \qquad p \implies q}{q}$$

This rule can be read as "given p and $p \implies q$, then we can conclude q". If there are no premises given above the horizontal bar, then the conclusion under the bar is considered an *axiom*, that is, it is considered to always hold.

Turnstile

The turnstile ('$\vdash$') is also a feature from a logic system (namely sequent calculus) which is used to separate assumptions (appearing on the left) from propositions (appearing on the right). It means that the proposition on the right can be derived or deduced from the assumptions on the left. The symbol $\vdash$ can be read as 'yields' or 'entails'. Multiple assumptions may appear on the left side, but only one proposition can appear on the right (Kleene). As an example: $p, p \implies q \vdash q$ reads 'p and $p \implies q$ yields q', which is valid since q can be directly derived from the assumptions.

Typing Context

A typing context, usually denoted Γ, is a set containing the declarations of variables and their types. For example, $\Gamma = \{x : \texttt{Int}, f : \texttt{Int} \to \texttt{Bool}\}$ is an example of a typing context (Pierce).

Typing Judgments and Typing Rules

A typing judgment is essentially an assertion telling us the type of an expression once it is fully evaluated (Pierce). For example, $1 + 1 : \texttt{Int}$ is a typing judgment telling us that the expression $1 + 1$, once evaluated, has the type `Int`.

However, we often come across expressions such as $x + 1$, where we have a variable whose type we do not know. These expressions have to be

considered with respect to some typing context which can tell us the types of the variables appearing in it.

We can define the following inference rule to say that if the variable x and its type τ is in the context, then we are allowed to conclude that in an expression containing the variable x, the type of x is τ:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Inference rules such as this which combine typing judgments are commonly referred to as typing rules.

Going back to our example of the expression $x + 1$, if we have the context $\Gamma = \{x : \texttt{Int}\}$, then we can assert the type of $x + 1$ with the judgment $\Gamma \vdash x + 1 : \texttt{Int}$, since Γ now contains the type of x.

In this manner, by combining typing judgments with the turnstile notation, we can write the judgment $\Gamma \vdash e : \tau$ to say that, under the assumption that Γ contains the types of all variables occurring in e, we can assert that e, once evaluated, has the type type τ.

9.2 Appendix B: System F Rules

To extend λ-calculus with polymorphism, we make two new additions: **type variables**, which are similar to normal variables except that they range over all *types* rather than *values*, and **polymorphic types**, which are types that contain type variables (Sørensen and Urzyczyin).

These additions require new syntax for types:

τ `::=`	*types:*
`X`	*type variable*
$\mid \forall$`X.`τ	*polymorphic type*
$\mid \tau \to \tau'$	*function type*
`| Bool`	*boolean type*

Listing 3 already gives an example of a type variable. An example of a polymorphic type would be $\forall$`X.X` $\to$ `X`. A function with such a type would accept a value of any type, and return a value of the same type (this is similar in nature to the identity function - in fact, it turns out that any function with the type $\forall$`X.X` $\to$ `X` will be equivalent to the identity function!).

We also update the syntax for terms in order to allow type abstraction and application:

`t ::=`	*terms:*
...	*(previous terms)*
$\mid \Lambda$`X.t`	*type abstraction*
`| t` τ	*type application*

Type abstractions are again similar to normal abstractions, but with type variables instead of normal variables - the term Λ`X.t` introduces a new type variable `X`, abstracted over the term `t` (similar to the type variable `T` abstracted over the method `wrap` in Listing 3). We can now write a *generic* identity function using type abstractions:

$$\texttt{let id} = \Lambda\texttt{X.}\lambda x : \texttt{X.}x$$

This is equivalent to the following Java program:

Listing 5: Generic Identity Function in Java

```
public <T> T identity(T x) {
   return x;
}
```

This type abstraction can then be applied to *instantiate* the polymorphic type with a concrete type. For example, '`id Int`' applies the type `Int` to our generic `id` function. The type variable is then replaced with the specific type, in this case giving us `id : Int` $\rightarrow$ `Int`. In Java, this is analogous to supplying our generic method with a specific type when calling it:

Listing 6: Generic Identity Function in Java

```
identity<int>(1); // The type variable T is replaced with 'int'.
```

The typing rules for type abstraction and application are as follows:

$$\text{Ty-TyAbs: } \frac{\Gamma \vdash \mathtt{t} : \tau}{\Gamma \vdash \Lambda \mathtt{X}.\mathtt{t} : \forall \mathtt{X}.\tau} \qquad \text{Ty-TyApp: } \frac{\Gamma \vdash \mathtt{t} : \forall \mathtt{X}.\tau}{\Gamma \vdash \mathtt{t}\ \tau' : [\mathtt{X} \mapsto \tau']\tau}$$

The first rule describes type abstraction: $\Lambda\mathtt{X}.\mathtt{t}$ has type $\forall\mathtt{X}.\tau$ if $\mathtt{t}$ has type τ. The second describes type application: if a term $\mathtt{t}$ has the type $\forall\mathtt{X}.\tau$, then the type application $\mathtt{t}\ \tau'$ has the type $[\mathtt{X} \mapsto \tau']\tau$, that is, the type obtained when τ' is substituted for the type variable $\mathtt{X}$.

These rules form the basis of System F, and outline how it is equivalent to generic programming patterns in mainstream programming languages.

9.3 Appendix C: Subtyping for Record Types

Evaluation Rules for Records

The relevant evaluation rules for records are as follows:

$$\text{E-Proj: } \frac{\texttt{t}_1 \longrightarrow \texttt{t'}_1}{\texttt{t}_1\texttt{.l} \longrightarrow \texttt{t'}_1\texttt{.l}} \qquad \text{E-ProjRcd: } \{\texttt{l}_i\texttt{=}\texttt{v}_i{}^{i \in 1..n}\}\texttt{.l}_\texttt{j} \longrightarrow \texttt{v}_j$$

E-Proj tells us to fully evaluate a record term before applying projection, while **E-ProjRcd** tells us that a projection of some label $\texttt{l}_j$ on a record evaluates to the corresponding value $\texttt{v}_j$ of the field identified by the label. For example, `{x:true}.x` evaluates to `true`.

Basic Subtyping Rules

The following transitivity and reflexivity rules can be derived straightforwardly from the Liskov substitution principle (that a supertype may be safely substituted with its subtype in any context):

$$\text{S-Trans: } \frac{\texttt{S <: T} \qquad \texttt{T <: U}}{\texttt{S <: U}} \qquad \text{S-Refl: } \texttt{S <: S}$$

Works Cited

Boehm, Hans-J. "Partial polymorphic type inference is undecidable". Proc. of 26th Annual Symposium on Foundations of Computer Science. IEEE, 1985. Web.

——."Type Inference in the Presence of Type Abstraction". Proc. of ACM SIG-PLAN Conference on Programming Language Design and Implementation. Portland, Oregon, USA: Association for Computing Machinery, 1989. Web. PLDI '89.

Breslav, Andrey. "JVM Languages Report: Extended Interview With Kotlin Creator Andrey Breslav". 2013. Web.

Cardelli, Luca and Peter Wegner. "On understanding types, data abstraction, and polymorphism". *ACM Computing Surveys* (1985). Print.

Church, Alonzo. *Introduction to Mathematical Logic*. 1936. Print.

——.*The Calculi of Lambda-Conversion*. 1941. Print.

Damas, Luis. "Type Assignment in Programming Languages". University of Edinburgh, 1985. Print.

Dolan, Stephen. "Algebraic Subtyping". University of Cambridge, 2016. Print.

Girard, Jean-Yves. "Interprétation fonctionnelle et élimination des coupures de l'arith-métique d'ordre supérieur". Université Paris VII, 1972. Print.

Hindley, J. Roger. "The Principal Type Scheme of an Object in Combinatory Logic". *Transactions of the American Mathematical Society* (1969). Print.

Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd. 2007. Print.

Kfoury, A. J., J. Tiuryn, and P. Urzyczyn. "The undecidability of the semi-unification problem". *Information and Computation 1* (1993). Print.

Kleene, Stephen C. *Introduction to Metamathematics*. 1952. Print.

Krishnamurthi, Shriram. *Programming Languages: Applications and Interpretations (PLAI)*. 2012. Web.

Krishnamurthi, Shriram, Benjamin S. Lerner, and Joe Gibbs Politz. *Programming and Programming Languages (PAPL)*. 2019. Web.

Liskov, Barbara H. and Jeannette M. Wing. "A Behavioral Notion of Subtyping". *ACM Trans. Program. Lang. Syst.* (1994). Web.

Melo, Leandro T. C. "The Basics of a Standard Type Inference Technique". 2020. Web.

Milner, Robin. "A Theory of Type Polymorphism in Programming". *Journal of Computer and System Sciences* (1978). Print.

Mitchell, John C. "Type inference with simple subtypes". *Journal of Functional Programming* (1991). Print.

O'Sullivan, Bryan, Don Stewart, and John Goerzen. *Real World Haskell*. 2008. Web.

Parreaux, Lionel. "Demystifying MLsub — the Simple Essence of Algebraic Subtyping". Mar. 2020. Web.

Pierce, Benjamin C. *Types and Programming Languages*. The MIT Press, 2002. Print.

Pierce, Benjamin C. and David N. Turner. "Local Type Inference". *ACM Trans. Program. Lang. Syst.* 22 (Jan. 2000). Web.

Reynolds, John C. "Towards a theory of type structure". *Proc. Collaque sur la Programmation* (1974). Print.

Sørensen, Morten Heine and Pawel Urzyczyin. "Chapter 11 - Second-order logic polymorphism". *Lectures on the Curry-Howard Isomorphism*. Edited by Morten Heine Sørensen and Paweł Urzyczyin. Vol. 149. Elsevier, 2006. 269–302. Web. Studies in Logic and the Foundations of Mathematics.

Suidman, Splinter. "Introduction to Type Systems: Simply Typed Lambda Calculus". 2020. Web.

——."Introduction to Type Systems: Type Inference". 2020. Web.

Suter, Toni. "Bi-directional Type Inference in Swift". 2017. Web.

Wells, Joe B. "Typability and type checking in System F are equivalent and undecidable". *Annals of Pure and Applied Logic* (1999). Web.