Polymorphic Type Inference

In this, we will discuss a typed λ-calculus, which allows for polymorphic functions. To be specific, we will discuss a kind of polymorphism where function parameters specify the types of some of the identifiers. This is called *parametric polymorphism*. We will discuss Milner's algorithm to *infer* types when they are not given.

1 A Typed λ calculus

There are several variants of λ-calculus with types. A standard way to introduce this is to consider λ-calculus where every identifier is annotated with a type. Since the λ-calculus we will consider for this section adopts a different approach, we will give only a brief introduction to the simply typed λ-calculus with type annotations. This discussion is taken from [2].

For example, we will consider a single *ground type* $\{o\}$. We could, of course, start with a richer type system, but it is unnecessary for now.

Given the ground types, we could consider more sophisticated types. For example, we could consider types of functions mapping some type to another. So the grammar for types is:

$$\tau ::= \{o\} \mid \tau \to \tau.$$

Given this grammar, the following expressions are valid types.

$$\{o\} \to \{o\}, \{o\} \to \{o\} \to \{o\}.$$

Let $\mathcal{V}$ be a countably infinite set of variables. We will consider *pre-terms* defined by the grammar. One noticeable difference is that the λ-term now contains an annotation declaring that the variable has a certain type.

$$\mathcal{T} = \mathcal{V} \mid \mathcal{T}\mathcal{T} \mid \lambda V : \tau \ \cdot \ \mathcal{T}.$$

Not every preterm is considered a valid term of the typed λ-calculus. We will give a set of inference rules which will finally specify which λ-pre-terms are valid *terms* of the simply typed λ-calculus. This step is one crucial difference between the untyped and the simply typed calculi.

A *context* is a mapping of a finite set of variables to types. They are written in the form $x_i : T_i \ldots x_n : T_n$. In this expression, $x_i : T_i$ indicates that in the context, the variable x_i has type T_i.

A judgment is a ternary relation $\Gamma \vdash s : B$ indicating that in the context Γ, we can *deduce* that s has type B.

$$\frac{}{x_1 : T_1 \ldots x_n : T_n \vdash x_i : T_i \;\; 1 \leq i \leq m}$$

$$\frac{\Gamma \vdash f : S \to T \quad \Gamma \vdash x : S}{\Gamma \vdash fx : T}$$

Judgment for application

$$\frac{\Gamma x : A \vdash e : B}{\Gamma \vdash (\lambda x : S \cdot e) : S \to T}$$

Judgment for λ-abstraction

We will deal with a slightly different typed system in which the terms do not have explicit type annotations.

2 The Language

We follow Cardelli [1] in introducing the following language:

EXP :=VAR |
if EXP **then** EXP **else** EXP |
λVAR $\cdot$ EXP |
EXP (EXP) |
let DECL **in** EXP

DECL :=VAR = EXP |
DECL **then** DECL
rec DECL
(DECL)

Data types are introduced into the language by having a predefined set of identifiers in the initial environment.

An Example program:

```
        let rec factorial =
                λn·
                if zero(n)
                then succ(0)
                else times(n)(factorial(pred(n)))
in factorial(0)
```

Our goal is to introduce a system where types can contain *type variables*, which will be determined by the compiler. For example, in

```
let id =(λa : α · a)                    //α is a type variable
   id(3)
id(true)
  id(id)                                          //type error
```

we introduce a polymorphic identity expression. It is more general than the identity function in a language which allows parameters of only a specific type.

A similar declaration is possible in a dynamically typed language like Oz. If the invocation involves a wrong type, then we would get an exception at runtime. What is different about the type inferencing algorithm is that at compile-time, we can infer the types.

We have the best of both - the flexibility of dynamic types, and the safety of a static typed system. The algorithm we will describe is due to Milner. It can infer types in the absence of type declarations. It is sound (), efficient (linear time in the size of the source code), and supports a sophisticated type system. We will present only the language above. Sophisticated type inferencing with exception handling, overloading, mutable data, records and union types are not covered.

3 Types

A type can be

- A Type variable: e.g. α, β etc. standing for an arbitrary type.
- A Type operator. These can be nullary operators like INT, BOOL, or operators taking arguments, like $\rightarrow$ (the function operator) and $\times$ (the cartesian product). These are described below.

  $\alpha \rightarrow \beta$ is an arbitrary function mapping domain α to a Ottoman β. $\alpha \times \beta$ is a type of any pair of values.

Types (like $\alpha \rightarrow \beta$) containing type variables are called polymorphic types. Types (like INT $\rightarrow$ BOOL) devoid of type variables are called monomorphic types.

Expressions involving multiple occurrences of the same type variable (like $\alpha \to \alpha$) are *contextual dependencies*.

The underlying algorithm for instantiation is *unification*. The typechecking unification fails when either it tries to match different type operators (as in INT and BOOL) or it tries to resolve circular type definitions (as in α and $\alpha \to \beta$.)

Since circular types are forbidden, functions such as

$$(\lambda x \cdot xx)$$

are invalid in our language.

4 Initial Type Environment

The initial type environment consists of certain identifiers. For our discussion, it suffices to consider the following.

5 The Algorithm, Part I

There are the following cases.

1. When a variable x is introduced by a λ-binder, it is assigned a new type variable α. This is stored in an environment.
2. In a conditional, if the **if** condition expression is matched to BOOL, the **then** and the **else** clauses are unified to determine a unique type for the whole expression.
3. In $(\lambda x \cdot e)$, the type of e is inferred in an environment where x is associated with a new type variable.
4. In an application fe where the type of e is α, we can infer that $f : A \to \beta$ where *beta* is a new type variable.

6 An Example: Compose

We will apply the algorithm to infer the type of the λ-expression for composition defined as

$$(\lambda g \ \cdot \ (\lambda f \ \cdot \ (\lambda x \ \cdot \ f(gx))))$$

Even though the algorithm has been described in a top-down fashion, it works in a bottom-up manner. We will now derive the type of the composition expression in a bottom-up fashion. We do not use any free variables in this definition, hence the initial environment can be empty.

$x : \alpha$	
$g : \varepsilon_1$	
$(gx) : \beta$	
$f : \varepsilon_2$	
$f(gx) : \gamma$	
$E = (\lambda x \ \cdot \ f(gx)) : \alpha \to \gamma$	Evaluating $f(gx)$ in the environment with $x : \alpha$
	Unify(ε_1, $\alpha \to \beta$)
$F = (\lambda f \cdot E) : (\beta \to \gamma) \to (\alpha \to \gamma)$	E evaluated with $f : \varepsilon_2$
	Unify(ε_2, $\beta \to \gamma$)
$G = (\lambda g \ \cdot \ F) : (\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$	F evaluated with $g : \alpha \to \beta$

7 The Algorithm, Part II: Let expressions and Declarations

The type inferencing of let expressions involves the concept of *generic* variables. To motivate this concept, let us consider λ bound variables.

Consider the following function.

$$(\lambda f \ \cdot \ pair(f(0), f(true)))$$

What should be the type of f? The first instance, according to our algorithm, would suggest a type of $\textsc{Int} \to \beta$. The second instance would suggest a type of $\textsc{Bool} \to \beta$. However, unification of these two types will fail. This is a limitation of Milner's type inferencing algorithm.

Type variables occurring in the type of a λ expression are called non-generic. They are shared between all of their occurrences in the λ expression, and this could lead to conflicts.

How is the above case different from the following code?

$$\begin{aligned} &\textbf{let } id = (\lambda a \ \cdot \ a) \\ &\textbf{in } pair(id(0), id(true)) \end{aligned}$$

In this case, we know what id is. We can think that unlike the λ-bound variable f which was available to us only as a "black box", we have the definition of id available to us. This allows us to infer its type in a more general manner than in the previous example.

The type of id is inferred as $\alpha \to \alpha$ by our algorithm. Every instance of id which occurs in the body of the **let** expression can be instantiated to a different type.

For example, in the occurrence $id(0)$, following the algorithm, the instance of id has type $\textsc{Int} \to \textsc{Int}$. In its second occurrence, id has type $\textsc{Bool} \to \textsc{Bool}$. This can be achieved by making a copy of the type

of *id* for every distinct occurrence, and *then* instantiating it. We have to allow these different instances. Otherwise, polymorphism would be absent.

Definition 1. Type variables which occur in the type of a **let** expression, and which do not occur in an enclosing λ-definition are called *generic* variables.

We will see what the problem of enclosing λ definitions are.

$$
\begin{array}{l}
\lambda g\cdot \\
\quad \textbf{let } id = g \\
\quad \textbf{in } pair(id(0), id(true))
\end{array}
$$

This example should also cause a unification failure. Non-generic variables are shared between occurrences. We should not make new copies of the non-generic type variable for g. Hence this case is the same as the first.

Our algorithm should therefore consider whether a variable is generic before deciding the instantiating mechanism. To incorporate this facility into our algorithm, we can maintain a list of non-generic variables. Any variable which is absent from the list is a generic variable. This list is augmented when entering a λ-declaration, and it automatically is restored to the previous list when leaving the λ-expression. (This is similar to how we determine the list of bound variables in a procedure.) Now we can discuss the algorithm for **let** expressions.

5. To infer the type of **let**, we infer the type of its declaration part. This yields a new environment of identifiers and types, which we can use to infer the type of the body of **let**.
6. Suppose a declaration is of the form $x_i = t_i$. Then we introduce $\langle x_i, T_i \rangle$, where $t_i : T_i$.

In the case of mutually recursive declarations, we first create an environment with $\langle x_i, \alpha_i \rangle$ where α_i are non-generic type variables. In this environment, we infer the types of t_i. Their types T_i are unified with α_i.

8 Inferring the type of the Length

We will consider an example of inferring types in presence of let expressions. The following expression defines a λ-term to evaluate the length of a homogeneous list. The type of a homogeneous list is denoted αLIST.

$$
\begin{array}{l}
\textbf{let rec } length = \\
\qquad \lambda l \cdot \\
\qquad\quad \textbf{if } null(l) \\
\qquad\quad \textbf{then } 0 \\
\qquad\quad \textbf{else } succ(length(tail\ l))
\end{array}
\qquad .
$$

It has the following free variables. We will start with the environment containing the corresponding types.

$$\begin{aligned} null &: \alpha\textsc{List} \to \textsc{Bool} \\ 0 &: \textsc{Int} \\ succ &: \textsc{Int} \to \textsc{Int} \\ tail &: \alpha\textsc{List} \to \alpha\textsc{List} \end{aligned}$$

The crucial steps in inferring the type of *length* are as follows.

$$\begin{array}{rr} l : \gamma & \\ null(l) : \textsc{Bool} & \text{Type of null} \\ l : \alpha\textsc{List} & \text{Unify}(\gamma,\ \alpha\textsc{List}) \end{array}$$

Since $0 : \textsc{Int}$ and the rule for the **if** block enforces that both the **then** clause and the **else** clause have the same type, after unification, we obtain

$$succ(length(tail\ l)) : \textsc{Int}$$

Unifying $l : \alpha$ with the type of $tail : \beta\textsc{List} \to \beta\textsc{List}$ (Important: We pick new variable names for the type of tail.) gives us that $\beta = \alpha$. Further,

$$length : \alpha\textsc{List} \to \delta.$$

Further unifying with *succ* resolves δ, giving us that

$$length : \alpha\textsc{List} \to \textsc{Int}$$

What's left

We have not talked about the formal semantics of type checking. We could define a system of axioms and inferences (as in the case of the simply typed λ calculus) and prove that type of every well-formed expression is correctly inferred by the algorithm. This lies outside our scope. Interested readers might refer to [1].

References

[1] Luca Cardelli. Basic polymorphic type checking. *Science of Computer Programming*, 8(2):147–172, 1987.

[2] Ralph Loader. Notes on simply typed lambda calculus. 1998.