Type Inference

SPRING 2020

Overview

- Briefly describe type inference
- Generally talk how to use type inference to determine types
- *Objective*
  - Gain insight of what the type inference is
  - How to use type inference for determining a type by given function
  - How to write a function by given a type signature

What is type inference?

- *Def.* --- given an expression, type inference statically analyzes the expression and determines its type
- Objective --- determine the best type for an expression, based on syntax and semantic information in the expression
  - Type inference guarantees to produce the most general type
- Type variable --- a variable represents unspecified types that can be instantiated by any type to satisfy the type constraint. That is, a general type!!!
  - Represented as a prime (') followed by a lowercase alphabet
  - For instance, 'a, 'b , 'c … in type signature

```
fun id x = x  (* val id = fn : 'a -> 'a *)
val one = id 1
val hello = id "hello"
```

Type inference vs. Type checking

Type inference

```
fun is_large x =
    if x > 37 then true
    else false
```

- Infer the type without the type declaration
- Use Hindley–Milner (HM) type inference algorithm to infer type consistent at compile time
- `is_large` takes an integer as argument and returns a Boolean

Type checking (static)

```
bool is_large (int x) {
    if (x > 37) {
        return true;
    }
    else {
        return false;
    }
}
```

- Analyze type and body of each function at compile time
- Use explicit type declaration to verify type consistency

Q: What is HM type inference?

How to do type inference by yourself?

- Consider a function:

```
fun is_large x =
    if x > 37 then true
    else false
```

  - **val is_large = fn : int -> bool**
- How to determine its type by using HM algorithm?

- Based on the header, assume the type of function arguments and output (denote them with type variables)
- Walk through the body of the function, determine type of inputs and outputs

1. is_large: $'a \rightarrow 'b$
2. Look at if expression
   I. x > 37, x must have type of int
   II. true, function returns value with type of Boolean
   III. false, function returns value with type of Boolean
3. DONE!!! is_large has type of int -> bool

More examples…

High order function

```
fun comp f g x = g (f x)
```

- Firstly looking at the the header, comp returns a type of $'d$ where inputs three arguments $'a$, $'b$ and $'c$:

$$\texttt{comp: } 'a \rightarrow 'b \rightarrow 'c \rightarrow 'd$$

- Now walking through the body, the variable f takes an argument of x, we assume it returns a result of new type which we annotate as $'e$:

$$\texttt{f: } 'a = 'c \rightarrow 'e$$

- The variable g takes an argument of which function f returns, and g returns a result which type is same as the function returns:

$$\texttt{g: } 'b = 'e \rightarrow 'f, 'd = 'f$$

- Plug all sub-types into the function signature:

$$\texttt{comp: } ('c \rightarrow 'e) \rightarrow ('e \rightarrow 'f) \rightarrow 'c \rightarrow 'f$$

- After renaming:
  val comp = fn : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c

Pattern matching

```
fun append l1 l2 =
  case l1 of
    [ ] => l2
  | h1::t1 => h1 :: append t1 l2
```

- Look at the the header, append returns a type of $'c$ where inputs two arguments $'a$ and $'b$:

$$\texttt{append: } 'a \rightarrow 'b \rightarrow 'c$$

- Considering the first case matching, we infer the variable of l1 is a list by [ ], and function returns a result of l2:

$$\texttt{append: } 'd\ list \rightarrow 'b \rightarrow 'b$$

- Now examining the second case, we already know the type of l1 is a list, and function returns a list that takes elements of l1. Thus, the result list must have the same type of l1 that also includes type of l2:

$$\texttt{append: } 'd\ list \rightarrow 'd\ list \rightarrow 'd\ list$$

- After renaming: **val append = fn : 'a list -> 'a list -> 'a list**

Type error detection by type inference

- Restriction with polymorphism
  - By type inference, compiler does not believe programmer given input parameter with a correct type
  - The parameter will be treated as monomorphically (consistently). Once the specific type of one variable is determined, the compiler will not be allowed the type to be generalized
  - Example

```
fun g (f: 'a -> 'a) = (f true, f 0)
```

    - In this example, the compiler will raise an error because f is taking either an Boolean argument or an integer.

More details related to type inference

- Fully Hindley–Milner (HM) type inference algorithm
  - Generate a sequence of constraints -> Solve the constraints -> Find a unifier
- Unification
  - Intuition --- find the best solution that could generate all solutions
  - More details discussed at Prolog
- Type system and subtype
- High order function
- Pattern matching