

COSE212: Programming Languages

Lecture 16 — Let-Polymorphic Type System

Hakjoo Oh
2023 Fall

Motivation

- Our type system is useful but it is not as expressive as we would like it to be. In particular, it does not support *polymorphism*¹. For example, it rejects the following program:

```
let f = proc (x) x in
  if (f (iszero (0))) then (f 11) else (f 22)
```

- Polymorphic functions are widely used in practice, so OCaml supports polymorphism:

```
# let f = fun x -> x in
  if (f (0=0)) then (f 11) else (f 22);;
- : int = 11
```

- Let's extend our type system to the let-polymorphic type system, the ML-style polymorphism.

¹Polymorphism refers to the language mechanisms that allow a single part of a program to be used with different types in different contexts

What went wrong?

```
let f = proc (x) x in
  if (f (iszero (0))) then (f 11) else (f 22)
```

- We assign type $t \rightarrow t$ to f , generating the constraint that the argument and return types are the same.
- Intuitively, the program can be well typed because the all usages of f satisfy the required constraint:
 - ▶ In $(f (iszero 0))$, we can assign $bool \rightarrow bool$ to f .
 - ▶ In $(f 11)$ and $(f 22)$, we can assign $int \rightarrow int$ to f .
- However, our type checking algorithm uses the same type variable t in both cases and generates the spurious constraint that $bool = int$.
- Any idea to fix this problem?

A Simple Solution

Associate a *different* variable t with each use of f . This is easily accomplished by substituting the body of f for each occurrence of f . For example, convert the program

```
let f = proc (x) x in
  if (f (iszero (0))) then (f 11) else (f 22)
```

into the following before type-checking:

```
if ((proc (x) x) (iszero (0)))
then ((proc (x) x) 11)
else ((proc (x) x) 22)
```

which is accepted by our type system as we can generate different type variables for different copies of the procedure.

Typing Rule

Instead of the ordinary typing rule for let:

$$\frac{\Gamma \vdash E_1 : t_1 \quad [x \mapsto t_1]\Gamma \vdash E_2 : t_2}{\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : t_2}$$

we use the new typing rule:

$$\frac{\Gamma \vdash [x \mapsto E_1]E_2 : t_2}{\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : t_2}$$

The corresponding algorithm for generating type equation:

$$\mathcal{V}(\Gamma, \text{let } x = e_1 \text{ in } e_2, t) = \mathcal{V}(\Gamma, [x \mapsto e_1]e_2, t)$$

The ordinary unification algorithm does the rest.

Flaws

This simplistic method has some flaws that need to be addressed before we can use it in practice.

- 1 Unused definitions are not type-checked, so a program like
`let x = <unsafe code> in 5`
will pass the type-checker. (This can be easily fixed. See Exercise 1)
- 2 The method is not efficient if the body of `let` contains many occurrences of the bound variables:

```
let a = <complex code> in
  let b = a + a in
    let c = b + b in
      let d = c + c in
        ...
```

The typing rule can cause the type-checker to perform an amount of work that is exponential in the size of the original code.

Exercise 1

Fix the typing rule and \mathcal{V} to repair the first problem.

Let-Polymorphic Type Checking Algorithm

To avoid the re-computation, practical implementations of languages with let-polymorphism use a more clever algorithm. In outline, the type-checking of

$$\text{let } x = e_1 \text{ in } e_2$$

proceeds as follows:

- We find the most general type t of e_1 by running the ordinary type-checking algorithm.
- We *generalize* any variables remaining in the type, obtaining the *type scheme* $\forall \alpha_1 \dots \alpha_n. t$, where $\alpha_1 \dots \alpha_n$ appear in t .
- We extend the type environment to record the type scheme for the bound variable x , and start type-checking e_2
- Each time we encounter an occurrence of x , we generate fresh type variables $\beta_1 \dots \beta_n$ and use them to instantiate the type scheme.

Example 1

```
let f = proc (x) 1 in (f 1) + (f true)
```

Example 2

```
let f = proc (x) x if (f true) then 1 else ((f f) 2)
```

Generalization Is Not Always Safe

Care is needed when generalizing types because doing so is not always safe. For example, consider the program:

```
proc (c)
  (let f = proc (x) c in
    if (f true) then 1 else ((f f) 2))
```

- The most general type for f is $t_1 \rightarrow t_2$.
- Generalizing the type, we obtain the type scheme $\forall t_1, t_2. t_1 \rightarrow t_2$.
- The body of `let` is well-typed by instantiating t_2 to `bool` for the first occurrence of f and to some function type for the second occurrence of f . The type system accepts the program.
- However, the program produces runtime error because no value c can be both a boolean and a procedure.
- To fix this problem, we disallow generalization for any type variables that are mentioned in the type environment. The safe type scheme for f is $\forall t_1. t_1 \rightarrow t_2$. With this generalization the program gets rejected.

Efficiency

- The algorithm is much more efficient than the simplistic approach.
- In practice, its time complexity is almost linear.
- However, the worst-case time complexity is still exponential.
- For example, try to evaluate the following OCaml program. It takes a very long time to typecheck.

```
let f0 = fun x -> (x,x) in
  let f1 = fun y -> f0 (f0 y) in
    let f2 = fun y -> f1 (f1 y) in
      let f3 = fun y -> f2 (f2 y) in
        let f4 = fun y -> f3 (f3 y) in
          let f5 = fun y -> f4 (f4 y) in
            f5 (fun z -> z)
```

Summary

- We extended our type system (called *simple type system*) to *let-polymorphic type system*, the core of ML type system.
- The extension is conservative:

$$\Gamma \vdash_{\text{simple}} E : T \implies \Gamma \vdash_{\text{poly}} E : T$$

Let-polymorphic type system accepts all programs acceptable by the simple type system.