ENE4014: Programming Languages

Lecture 16 — Let-Polymorphic Type System

Woosuk Lee 2024 Spring

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 polymorphism¹. 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
```

• Lets 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 $t \to t$ 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 ${\tt f}$. This is easily accomplished by substituting the body of ${\tt f}$ for each occurrence of ${\tt f}$. 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:

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

we used the new typing rule:

$$rac{\Gamma dash [x \mapsto E_1] E_2 : t_2}{\Gamma dash$$
let $x = E_1$ in $E_2 : t_2$

Here, $[x \mapsto E_1]E_2$ denotes an expression obtained by replacing each occurrence of x by E_1 in E_2 .

The corresponding algorithm for generating type equation:

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

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.

• 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 ${\cal V}$ to repair the first problem.

We can fix the problem by adding a premise to the typing rule:

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

and a corresponding premise to the algorithm:

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

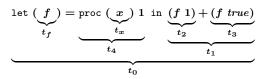
Let-Polymorphic Type Checking Algorithm

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

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

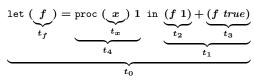
proceeds as follows:

- We find the most general type t of e_1 by running the ordinary type-checking algorithm (i.e., compute $\mathcal{U}(\mathcal{V}(\Gamma,e_1,t))$ where Γ is the type environment embracing e_1).
- 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.
- ullet 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.

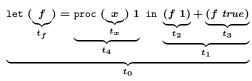


		Equations	Substitution
t_f	=	$orall t_x.\ t_x o {int}$	
t_1	=	int	
$\boldsymbol{t_2}$	=	int	
	=		
t_f	=	$int \to t_{2}$	
t_f	=	bool $ ightarrow t_3$	
t_0	=		

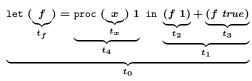
$$\mathcal{U}(\mathcal{V}(\emptyset, exttt{proc } (x) \ 1, t_4)) = t_x o ext{int.}$$



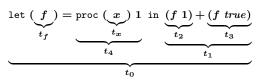
	Equations			bstitution
$egin{array}{lll} t_f &=& \ t_f &=& \ t_0 &=& \end{array}$	$egin{array}{l} int ightarrow t_2 \ bool ightarrow t_3 \ t_1 \end{array}$	t_f t_1 t_2 t_3	= = =	$orall t_x.\ t_x ightarrow $ int int int



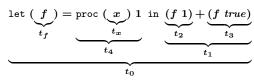
Equations	Substitution
$eta_1 o ext{int} egin{array}{ll} eta_1 o ext{int} &= & ext{int} o t_2 \ t_f &= & ext{bool} o t_3 \ t_0 &= & t_1 \ \end{array}$	$egin{array}{lll} t_f &=& orall t_x.\ t_x ightarrow { m int} \ t_1 &=& { m int} \ t_2 &=& { m int} \ t_3 &=& { m int} \ \end{array}$



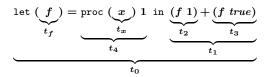
Equations	Substitution
$eta_1 = ext{int}$ int $= t_2$ $t_f = ext{bool} o t_3$ $t_0 = t_1$	$egin{array}{lll} t_f &=& orall t_x.\ t_x ightarrow { m int} \ t_1 &=& { m int} \ t_2 &=& { m int} \ t_3 &=& { m int} \ \end{array}$



Equations			Substitution			
				$egin{array}{c} t_f \ t_1 \end{array}$	=	$orall t_x \cdot t_x ightarrow ext{int}$ int int int int int int
				t_2	=	int
				t_3	=	int
				$oldsymbol{eta_1}$	=	int
				t_2	=	int
t_f	=	$egin{array}{c} bool ightarrow t_3 \ t_1 \end{array}$				
t_0	=	t_1				



Equations				Substitution		
				t_f t_1 t_2 t_3	= = = = =	$orall tution \ orall t_x \cdot t_x ightarrow ext{int} \ ext{int$
$eta_2 o \operatorname{int} \ t_0$	=	$egin{array}{c} bool ightarrow t_3 \ t_1 \end{array}$		t_2	=	int

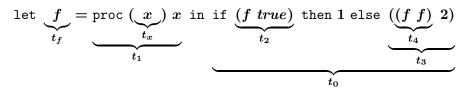


Equations	Substitution			
	t_f	=	$\forall t_x.\ t_x o int$	
	4.			
	t_2	=	int	
	t_3	=	int	
	β_1	=	int	
	t_2	=	int	
	eta_2	=	bool	
	t_3	=	int	
	t_0	=	int int int int bool int int	

$$\underbrace{f}_{t_f} = \underbrace{\operatorname{proc}\left(\underbrace{x}_{t_x}\right) x}_{t_1} \text{ in if } \underbrace{\left(f \ true\right)}_{t_2} \text{ then 1 else } \underbrace{\left(\left(f \ f\right)}_{t_4} \ 2\right)}_{t_0}$$

		Equations	Substitution
t_f	=	$orall t_x.\ t_x o t_x$	
$\boldsymbol{t_2}$	=	bool	
t_3	=	int	
$\boldsymbol{t_4}$	=	$int \to t_3$	
t_f	=	$bool \to t_2$	
$oldsymbol{t_f}$	=	$t_f o t_4$	
t_0	=	t_3	

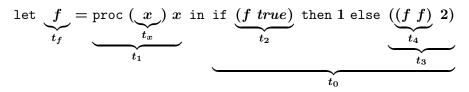
$$\mathcal{U}(\mathcal{V}(\emptyset, \text{proc } (x) | x, t_1)) = t_x \to t_x.$$



Equations	Substitution		
	$egin{array}{lll} t_f &=& orall t_x. \ t_x ightarrow t_x \ t_2 &=& {\sf bool} \ t_3 &=& {\sf int} \ t_4 &=& {\sf int} ightarrow {\sf int} \end{array}$		
$egin{array}{lll} t_f &=& bool ightarrow t_2 \ t_f &=& t_f ightarrow t_4 \ t_0 &=& t_3 \end{array}$	$t_4 = int o int$		

$$\underbrace{f}_{t_f} = \underbrace{\operatorname{proc}\left(\underbrace{x}_{t_x}\right) x}_{t_1} \text{ in if } \underbrace{\left(\underbrace{f \ true}_{t_2}\right)}_{t_2} \text{ then 1 else } \underbrace{\left(\underbrace{\left(f \ f\right)}_{t_4} \ 2\right)}_{t_3}$$

Equations	Substitution		
	$egin{array}{lll} t_f &=& orall t_x.\ t_x & ightarrow t_x \ t_2 &=& { m bool} \ t_3 &=& { m int} \ t_4 &=& { m int} ightarrow { m int} \end{array}$		
	$t_4 = int o int$		
$eta_1 o eta_1 \; = \; bool o bool$			
$egin{array}{lcl} t_f &=& t_f ightarrow t_4 \ t_0 &=& t_3 \end{array}$			



Equations	Substitution		
	t_f	=	$\forall t_x.\ t_x o t_x$
	t_2	=	bool
	t_3	=	int
	t_4	=	$int \to int$
	$oldsymbol{eta_1}$	=	$orall t_x. \ t_x ightarrow t_x$ bool int int $ ightarrow$ int bool
$t_f \; = \; t_f ightarrow t_4$			
$egin{array}{lll} t_f &=& t_f ightarrow t_4 \ t_0 &=& t_3 \end{array}$			

$$\underbrace{f}_{t_f} = \underbrace{\operatorname{proc}\left(\underbrace{x}_{t_x}\right) x}_{t_1} \text{ in if } \underbrace{\left(\underbrace{f\ true}_{t_2}\right)}_{t_2} \text{ then 1 else } \underbrace{\left(\underbrace{\left(f\ f\right)}_{t_4}\ 2\right)}_{t_3}$$

Equations	Substitution
	$egin{array}{lll} t_f &=& orall t_x.\ t_x ightarrow t_x \ t_2 &=& bool \ t_3 &=& int \ t_4 &=& int ightarrow int \ eta_1 &=& bool \ \end{array}$
	$t_2 = bool$
	$t_3 = int$
	$t_4 = int o int$
	$\beta_1 = bool$
$eta_2 ightarrow eta_2 \;\; = \;\; (eta_3 ightarrow eta_3) ightarrow t_4$	
$t_0 = t_3$	

$$\underbrace{f}_{t_f} = \underbrace{\operatorname{proc}\left(\underbrace{x}_{t_x}\right) x}_{t_1} \text{ in if } \underbrace{\left(\underbrace{f \ true}\right)}_{t_2} \text{ then 1 else } \underbrace{\left(\left(\underbrace{f \ f}\right)}_{t_4} 2\right)}_{t_0}$$

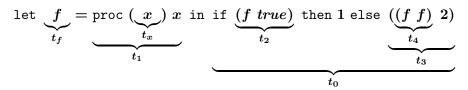
Equations	Substitution		
$eta_2 = eta_3 ightarrow eta_3 \ eta_2 = \inf ightarrow \inf$	$t_f = orall t_x. \ t_x ightarrow t_x$ $t_2 = ext{bool}$ $t_3 = ext{int}$ $t_4 = ext{int} ightarrow ext{int}$ $eta_1 = ext{bool}$		
$t_0 = t_3$			

$$\underbrace{f}_{t_f} = \underbrace{\operatorname{proc}\left(\underbrace{x}_{t_x}\right) x}_{t_1} \text{ in if } \underbrace{\left(\underbrace{f \ true}\right)}_{t_2} \text{ then 1 else } \underbrace{\left(\left(\underbrace{f \ f}\right)}_{t_4} 2\right)}_{t_0}$$

Equations		Su	bstitution
	t_f	=	$orall t_x.\ t_x o t_x$ bool int int $ o$ int bool $eta_3 o eta_3$
	t_2	=	bool
	t_3	=	int
	t_4	=	$int \to int$
	$oldsymbol{eta_1}$	=	bool
	$oldsymbol{eta_2}$	=	$eta_3 o eta_3$
$\rho_3 \rightarrow \rho_3 = \text{int} \rightarrow \text{int}$			
$t_0 = t_3$			

$$\underbrace{f}_{t_f} = \underbrace{\operatorname{proc}\left(\underbrace{x}_{t_x}\right) x}_{t_1} \text{ in if } \underbrace{\left(f \ true\right)}_{t_2} \text{ then 1 else } \underbrace{\left(\left(f \ f\right)}_{t_4} \ 2\right)}_{t_0}$$

Equations	Substitution			
	t_f	=	$orall transform egin{array}{c} orall t_x \cdot t_x ightarrow t_x \ ight$	
	t_2	=	bool	
	t_3	=	int	
	t_4	=	int o int	
	eta_1	=	bool	
	eta_2	=	$eta_3 o eta_3$	
eta_3 = int				
$egin{array}{lll} eta_3 &=& ext{int} \ t_0 &=& t_3 \end{array}$				



Equations	Substitution		
	t_f	=	$\forall t_x.\ t_x o t_x$
	t_2	=	bool
	t_3	=	int
	t_4	=	$int \to int$
	eta_1	=	bool
	eta_2	=	$int \to int$
	eta_3	=	int
	t_0	=	$orall t_x oldsymbol{t}_x oldsymbol{t}_x \to t_x$ bool int int $ o$ int bool int $ o$ int int int

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:

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

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