

Programming Languages

Lec 4: Design of PLs - Recursions, Scoping Rules (Advanced)

김재호

Proc Language

Where is the Recursion?

- Proc 언어에서 함수형 언어의 꽃(중 하나)인 recursion이 가능할까?

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

- 안 되는 듯하다..

Proc Language

Where is the Recursion?

- 사실, dynamic scoping이라면 recursion은 특별할 게 없다

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

- 그래도, static scoping을 포기할 수는 없다! — syntax, semantics를 조금 수정해 보자

RecProc: Proc + Recursions

RecProc Language

Syntax of RecProc

$$P \rightarrow E$$
$$E \rightarrow n$$
$$| x$$
$$| E + E$$
$$| E - E$$
$$| \text{iszero } E$$
$$| \text{if } E \text{ then } E \text{ else } E$$
$$| \text{let } x = E \text{ in } E$$
$$| \text{read}$$
$$| \text{letrec } f(x) = E \text{ in } E$$
$$| \text{proc } x \ E$$
$$| E \ E$$

RecProc Language

Example

```
letrec double(x) =  
  if iszero(x) then 0 else ((double (x-1)) + 2)  
in (double 1)
```

RecProc Language

Semantics of RecProc

- Domain:

$$\begin{aligned} Val &= \mathbb{Z} + Bool + Procedure + \textcolor{blue}{RecProcedure} \\ Procedure &= Var \times E \times Env \\ \textcolor{blue}{RecProcedure} &= \\ Env &= Var \rightarrow Val \end{aligned}$$

- Semantic rules:

RecProc Language

Semantics of RecProc

- Domain:

$$\begin{aligned} Val &= \mathbb{Z} + Bool + Procedure + \textcolor{blue}{RecProcedure} \\ Procedure &= Var \times E \times Env \\ \textcolor{blue}{RecProcedure} &= \textcolor{blue}{Var} \times \textcolor{blue}{Var} \times \textcolor{blue}{E} \times \textcolor{blue}{Env} \\ Env &= Var \rightarrow Val \end{aligned}$$

- Semantic rules:

$$\frac{[f \mapsto (f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{letrec } f(x) = E_1 \text{ in } E_2 \Rightarrow v}$$
$$\frac{\begin{array}{l} \rho \vdash E_1 \Rightarrow (f, x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \\ [x \mapsto v, f \mapsto (f, x, E, \rho')]\rho' \vdash E \Rightarrow v' \end{array}}{\rho \vdash E_1 E_2 \Rightarrow v'}$$

RecProc Language

Example

$$\frac{\begin{array}{c} [f \mapsto (f, x, f \ x, [])] \vdash f \Rightarrow (f, x, f \ x, []) \\ \frac{[x \mapsto 1, f \mapsto (f, x, f \ x, [])] \vdash f \ x \Rightarrow}{[x \mapsto 1, f \mapsto (f, x, f \ x, [])] \vdash f \ x \Rightarrow} \vdots \end{array}}{[f \mapsto (f, x, f \ x, [])] \vdash f \ 1 \Rightarrow} \frac{[f \mapsto (f, x, f \ x, [])] \vdash f \ 1 \Rightarrow}{[] \vdash \text{letrec } f(x) = f \ x \text{ in } f \ 1 \Rightarrow}$$

RecProc Language

Mutually Recursive Procedures

- 간혹 어떤 문제들은 서로를 참조하는 식으로 설계할 수 있기도 하다
- e.g., 홀짝 판별 (음이 아닌 정수 x 에 대해)
 - “ x 가 짝수입니까?”라고 물어보면
 - x 가 0이라면 맞습니다. 아니라면 “ $x - 1$ 은 홀수입니까?”의 대답과 같습니다.
 - “ x 가 홀수입니까?”라고 물어보면
 - x 가 0이라면 틀렸습니다. 아니라면 “ $x - 1$ 은 짝수입니까?”의 대답과 같습니다.

RecProc Language

Mutually Recursive Procedures

- 홀짝 판별을 OCaml에서 구현하면

```
let rec even x = if x = 0 then true else odd (x - 1)
```

```
and odd x = if x = 0 then false else even (x - 1)
```

- 이를 mutually recursive function(procedure)이라 부른다

RecProc Language

Mutually Recursive Procedures

- 우리의 RecProc에도 mutually recursive procedures를 추가해 보자!

$$\begin{array}{lcl} P & \rightarrow & E \\ E & \rightarrow & n \\ & | & x \\ & | & E + E \\ & | & E - E \\ & | & \text{iszero } E \\ & | & \text{if } E \text{ then } E \text{ else } E \\ & | & \text{let } x = E \text{ in } E \\ & | & \text{read} \\ & | & \text{letrec } f(x) = E \text{ in } E \\ & | & \text{letrec } f(x_1) = E_1 \text{ and } g(x_2) = E_2 \text{ in } E \\ & | & \text{proc } x \ E \\ & | & E \ E \end{array}$$

RecProc Language

Semantics of RecProc - Add Mutually Recursive Procedures

- Domain:

$$\begin{aligned} Val &= \dots + \textit{MRecProcedure} \\ \textit{MRecProcedure} &= ? \end{aligned}$$

- Semantic rules:

$$\frac{?}{\rho \vdash \text{letrec } f(x) = E_1 \text{ and } g(y) = E_2 \text{ in } E_3 \Rightarrow ?}$$

$$\frac{?}{\rho \vdash E_1 E_2 \Rightarrow ?}$$

Summary

RecProc = Basic Exp + Proc + Recursion + Muturally Recursion

- **Syntax**

$$\begin{array}{lcl} P & \rightarrow & E \\ E & \rightarrow & n \\ & | & x \\ & | & E + E \\ & | & E - E \\ & | & \text{iszero } E \\ & | & \text{if } E \text{ then } E \text{ else } E \\ & | & \text{let } x = E \text{ in } E \\ & | & \text{read} \\ & | & \text{letrec } f(x) = E \text{ in } E \\ & | & \text{proc } x \ E \\ & | & E \ E \end{array}$$

Summary

RecProc = Basic Exp + Proc + Recursion + Muturally Recursion

- Semantics

$$\begin{array}{c} \frac{}{\rho \vdash n \Rightarrow n} \quad \frac{}{\rho \vdash x \Rightarrow \rho(x)} \quad \frac{\rho \vdash E_1 \Rightarrow n_1 \quad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 + E_2 \Rightarrow n_1 + n_2} \\[10pt] \frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \text{iszero } E \Rightarrow \text{true}} \quad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \text{iszero } E \Rightarrow \text{false}} \quad n \neq 0 \quad \frac{}{\rho \vdash \text{read} \Rightarrow n} \\[10pt] \frac{\rho \vdash E_1 \Rightarrow \text{true} \quad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Rightarrow v} \quad \frac{\rho \vdash E_1 \Rightarrow \text{false} \quad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Rightarrow v} \\[10pt] \frac{\rho \vdash E_1 \Rightarrow v_1 \quad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{let } x = E_1 \text{ in } E_2 \Rightarrow v} \quad \frac{[f \mapsto (f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{letrec } f(x) = E_1 \text{ in } E_2 \Rightarrow v} \\[10pt] \frac{}{\rho \vdash \text{proc } x \ E \Rightarrow (x, E, \rho)} \\[10pt] \frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad [x \mapsto v]\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1 \ E_2 \Rightarrow v'} \\[10pt] \frac{\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad [x \mapsto v, f \mapsto (f, x, E, \rho')]\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1 \ E_2 \Rightarrow v'} \end{array}$$

Lexical Scoping of Variables

Declarations and References of Variables

변수의 생애

- Variable은 (당연히) 선언을 해야 사용할 수 있다
 - **Declaration of Variables:** 어떤 variable을 새롭게 어떤 값의 이름으로써 만드는 것
 - **Reference of Variables:** 특정 variable을 사용하는 것
-
- 우리는 declare된 variable을 어떤 값에 bound되었다 함

Scoping Rules

- 모든 variable들은 declare되어 reference할 수 있는 범위가 정해짐
- 이 범위에 대한 규칙을 **Scoping Rule**이라 함
- 대부분의 프로그래밍 언어는 **lexical scoping rule**을 사용. 왜? 실행 전에 알 수 있어서!

```
let x = 3
  in let y = 4
    in (let x = y + 5
        in x * y)
    + x
```

Static vs Dynamic Properties of Programs

- 프로그램에 존재하는 여러 요소들은 static하거나, dynamic하다
- **Static properties**
 - 프로그램이 실행되지 않아도 정해지는 요소들
 - Declaration, scope, etc.
- **Dynamic properties**
 - 프로그램이 실행되어야만 명확해지는 요소들
 - Values, types, the absence of bugs, etc.

Lexical Scopes of Variables

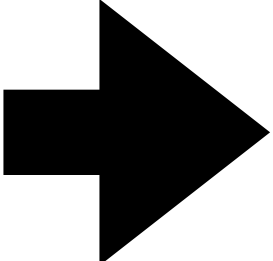
Example

```
proc (x)
  (proc (y)
    (let z = x + y
      in proc (x)
        (proc (z)
          (let x = (let x = x + y + z
                    in let y = 11
                      in x + y + z)
            in x + y + z)
          )
        )
      )
    )
  )
)
```

Lexical Address

Nameless / De Bruijn Representation

- 이런 식으로 lexical scope를 통해 variable의 scope를 알 수 있다면, 애초에 variable들의 이름을 없애버려도 되지 않을까?

<pre>let x = 1 in let y = 2 in x + y</pre>		<pre>let 1 in let 2 in #1 + #0</pre>
--	--	--

- 이런 표현법을 “**Nameless**” 또는 “**De Bruijn**” representation이라 한다
- 또, 이 때 각 variable에게 부여되는 **lexical address**는 **environment**에 추가되는 순서이기도 하다

Lexical Address

Example

```
(let a = 5 in proc (x) (x-a)) 7
```

```
(let x = 37  
  in proc (y)  
    let z = (y - x)  
    in (x - y)) 10
```

De Bruijn Proc

Syntax

$$\begin{array}{lcl} P & \rightarrow & E \\ E & \rightarrow & n \\ & | & \#n \\ & | & E + E \\ & | & E - E \\ & | & \text{iszero } E \\ & | & \text{if } E \text{ then } E \text{ else } E \\ & | & \text{let } E \text{ in } E \\ & | & \text{proc } E \\ & | & E \ E \end{array}$$

De Bruijn Proc

Semantics

$$\begin{aligned} \textit{Val} &= \mathbb{Z} + \textit{Bool} + \textit{Procedure} \\ \textit{Procedure} &= \textit{E} \times \textit{Env} \\ \textit{Env} &= \textit{Val}^* \end{aligned}$$

$$\frac{}{\rho \vdash n \Rightarrow n} \quad \frac{}{\rho \vdash \#n \Rightarrow \rho_n} \quad \frac{\rho \vdash E_1 \Rightarrow n_1 \quad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 + E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \text{iszero } E \Rightarrow \text{true}} \quad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \text{iszero } E \Rightarrow \text{false}} \quad n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow \text{true} \quad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Rightarrow v} \quad \frac{\rho \vdash E_1 \Rightarrow \text{false} \quad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad v_1 :: \rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{let } E_1 \text{ in } E_2 \Rightarrow v}$$

$$\frac{}{\rho \vdash \text{proc } E \Rightarrow (E, \rho)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad v :: \rho' \vdash E \Rightarrow v'}{\rho \vdash E_1 E_2 \Rightarrow v'}$$

De Bruijn Proc

더 자세한 이야기는 오프언 수업에서..

Summary

- **Static (lexical) Scope**는 프로그램의 scope를 실행 전에 확정하여 더 안전한 프로그램을 만들 수 있게 하기 때문에, 대부분의 언어가 차용중인 scoping rule
- **Static scope**에서 **recursion**을 사용하기 위해, Proc 언어에서 syntax와 semantics를 조금 손봐준 **RecProc**을 만들어 봄
- **Scoping rule**은 variables의 declaration과 reference를 명확히 하기 위한 규칙
- **Lexical scope**에서는, 프로그램을 실행하지 않고도 **nameless**로 바꿀 수 있음
- 사실 컴파일러가 내부적으로 **Nameless (De Bruijn) representation**으로 바꿔줌