Adding Procedures to LET (PROC)

CS496

PROC: A Language with Procedures

Extending our language LET with procedures:

1. Extending the concrete and abstract syntax.
2. Extending the set of Expressed Values.
3. Specification and Implementation of the interpreter.

The PROC-Language

An Interpreter for PROC

PROC: Concrete Syntax

Extending the concrete syntax of LET

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Number} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Identifier} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Expression} \rangle \text{ - } \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{zero? } (\langle \textit{Expression} \rangle) \\
\langle \textit{Expression} \rangle & ::= & \text{if } \langle \textit{Expression} \rangle \\
 & & \text{then } \langle \textit{Expression} \rangle \text{ else } \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{let } \langle \textit{Identifier} \rangle = \langle \textit{Expression} \rangle \text{ in } \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & (\langle \textit{Expression} \rangle)
\end{array}
$$

PROC: Concrete Syntax

Extending the concrete syntax of LET

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Number} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Identifier} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Expression} \rangle \text{ - } \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{zero? } (\langle \textit{Expression} \rangle) \\
\langle \textit{Expression} \rangle & ::= & \text{if } \langle \textit{Expression} \rangle \\
 & & \text{then } \langle \textit{Expression} \rangle \text{ else } \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{let } \langle \textit{Identifier} \rangle = \langle \textit{Expression} \rangle \text{ in } \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & (\langle \textit{Expression} \rangle) \\
 & & \\
\langle \textit{Expression} \rangle & ::= & \text{proc } (\langle \textit{Identifier} \rangle) \{ \langle \textit{Expression} \rangle \} \\
\langle \textit{Expression} \rangle & ::= & (\langle \textit{Expression} \rangle \langle \textit{Expression} \rangle)
\end{array}
$$

Examples of Expressions in PROC

```
let f = proc (x) { x-11 }
in (f (f 77))

(proc (f) { (f (f 77)) } proc (x) { x-11 })

let x = 200
in let f = proc (z) { z-x }
   in (f 1)

let x = 200
in let f = proc (z) { z-x }
   in let x = 100
      in let g = proc (z) { z-x }
         in (f 1) - (g 1)
```

PROC: Abstract Syntax

```
1  type prog =
2    AProg of expr
3
4  type expr =
5    | Var of string
6    | Int of int
7    | Sub of expr*expr
8    | Let of string*expr*expr
9    | IsZero of expr
10   | ITE of expr*expr*expr
11   | Proc of string*expr
12   | App of expr*expr
```

Concrete Syntax vs Abstract Syntax

- Concrete

```
let f = proc (x) { x-11 } in (f (f 77))
```

- Abstract

```
AProg
  (Let ("f",
    Proc ("x", Sub (Var "x", Int 11)),
    App (Var "f", App (Var "f", Int 77))))
```

An Interpreter for PROC

The Interpreter for PROC

- What is the result of evaluating the expression `2`?

The Interpreter for PROC

- What is the result of evaluating the expression `2`?

```
NumVal 2
```

- What is the result of evaluating the expression `zero?(2)`?

The Interpreter for PROC

- What is the result of evaluating the expression `2`?

```
NumVal 2
```

- What is the result of evaluating the expression `zero?(2)`?

```
BoolVal false
```

- What is the result of evaluating the expression `proc(x){ 7-x }`?

The Interpreter for PROC

- What is the result of evaluating the expression `2`?

  `NumVal 2`

- What is the result of evaluating the expression `zero?(2)`?

  `BoolVal false`

- What is the result of evaluating the expression `proc(x){ 7-x }`?
  - Its not a number
  - Its not a boolean
  - Its a closure
    - Record that stores the formal parameter and body
    - More details to follow

The Interpreter for PROC

- Before (for the LET-language)

```
eval_expr: env -> expr -> exp_val
```

- Now

```
eval_expr: env -> expr -> exp_val
```

- What's the difference?
  - The definition of expressed values
  - Our syntax now supports procedures
- Before

$$\text{ExpVal} \quad = \quad \text{Int} + \text{Bool}$$

- Now

$$\text{ExpVal} \quad = \quad \text{Int} + \text{Bool} + \quad \text{Clos}$$

The Interpreter for PROC

```
eval_expr: env -> expr -> exp_val
```

- Expressed values before (LET)

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
```

- Now (PROC)

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of ????
```

  - The ??? should be replaced by some representation of a closure

Defining Closures

What should the value of the following expression be?

```
proc (x){ x- 11 }
```

- A datatype that records the parameter and its body
- However, is that enough?
- What about the procedures `f` and `g` below?

```
1  let x = 200
2  in let f = proc (z) { z-x }
3     in let x = 100
4        in let g = proc (z) { z-x }
5           in (f 1) - (g 1)
```

- The value of `f` and `g` depends on the value of `x`.

Defining Closures

What should the value of the following expression be?

```
proc (x){ x- 11 }
```

- A datatype that records the parameter and its body
- However, is that enough?
- What about the procedures `f` and `g` below?

```
1  let x = 200
2  in let f = proc (z) { z-x }
3     in let x = 100
4        in let g = proc (z) { z-x }
5           in (f 1) - (g 1)
```

- The value of `f` and `g` depends on the value of `x`.

Defining Closures

```
let x = 200
in let f = proc (z) { z-x }
   in let x = 100
      in let g = proc (z) { z-x }
         in (f 1) - (g 1)
```

- The value of a procedure depends on the environment in which it is evaluated.
  - `f` and `g` only differ in the value of `x`.
- Summary of what a closure should contain:
  1. The formal parameter of the procedure
  2. The body of the procedure
  3. The environment extant at the point where the procedure was evaluated

Representing Closures

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*expr*env
and
  env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
```

- `exp_val` and `env` have to be defined together since they are mutually recursive

Specifying the Behavior of the Interpreter – `Proc`

```
1  eval_expr ρ Proc(var,body) = ProcVal (var,body,ρ)
```

- Recall from above: closure is a constructor with arguments:
  - a formal parameter var,
  - an expression body,
  - and an environment ρ.

Implementation

New clauses for `eval_expr`

```
1  eval_expr (en:env) (e:expr) :exp_val =
2    match e with
3    | Int n              -> NumVal n
4    ...
5    | Proc(x,e)          -> ProcVal (x,e,en)
6    | App(e1,e2)         -> ...
7    | _ -> failwith("Not implemented")
```

Specifying the Behavior of the Interpreter – `App`

- A procedure call is represented as

  `App(rator,rand)`

  in the abstract syntax
- We must therefore give meaning to

  `eval_expr` ρ`App(rator,rand)`

  1. Evaluate rator to a procedure (check it is `ProcVal`)
  2. Evaluate rand to an argument
  3. Pass argument to procedure
- Lets specify its behavior using equations

Evaluating Procedure Calls

- We must give meaning to `eval_expr` ρ`App(rator,rand)`
  1. Evaluate rator to a value
  2. Evaluate rand to an argument
  3. Make sure that value is a procedure (check it is `ProcVal`)
  4. Pass argument to procedure

```
1  eval_expr ρ App(rator,rand) =
2      let v1 = eval_expr en e1 in
3      let v2 = eval_expr en e2 in
4      apply_proc v1 v2
```

- apply_proc checks argument is actually a procedure (i.e. it was constructed with `ProcVal`)
- And then actually applies the closure. What does that mean?

Evaluating Procedure Calls

Consider the example:

```
apply_proc
  (ProcVal("x",
           Sub(Var("x",Int 11)),
           ρ)),
  (NumVal 20)
```

- The value of this expression is the value of the body `x-11` where `x`'s value is 20.

In general:

```
apply_proc ProcVal(var,body,ρ) val =
     eval_expr [var=val]ρ body
```

Evaluating Procedure Calls

```
1  apply_proc ProcVal(var,body,ρ) val =
2        eval_expr [var=val]ρ body
```

- The code corresponding to this specification is:

```
1  let rec apply_proc f a =
2    match f with
3    | ProcVal(x,b,env) -> eval_expr (extend_env env x a)
       ↪  b
4    | _ -> failwith "apply_proc: Not a procVal"
```

Implementation

New clauses for `eval_expr`

```
1  eval_expr (en:env) (e:expr) :exp_val =
2    match e with
3    | Int n             -> NumVal n
4    ...
5    | Proc(x,e)         -> ProcVal (x,e,en)
6    | App(e1,e2)        ->
7      let v1 = eval_expr en e1 in
8      let v2 = eval_expr en e2 in
9      apply_proc v1 v2
```

The Interpreter for PROC

- Code available in Canvas `Modules/Interpreters`
- Directory `proc-lang`
- Compile with `ocamlbuild -use-menhir interp.ml`
- Make sure the `.ocamlinit` file is in the folder of your sources
- Run `utop`