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

Specifying the Interpreter

Implementing the Interpreter

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 = 2
in let f = proc (z) { z-x }
in (f 1)

let x = 2
in let f = proc (z) { z-x }
in let x = 1
in let g = proc (z) { z-x }
in (f 1) - (g 1)
```

PROC: Abstract Syntax

```
type expr =
  | Var of string
  | Int of int
  | Sub of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
```

Concrete Syntax vs Abstract Syntax

- Concrete

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

- Abstract

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

Interpreter for Expressions

For each language construction we present two steps:

1. Specification of the interpreter (presented in a blue box in math-like language)

   Specification here!

2. Implementation of the interpreter (presented as a standard box in OCaml)

```
Code here!
```

Evaluation Judgements for PROC

$$\mathsf{e}, \rho \Downarrow r$$

- e is an expression in PROC
- ρ is an environment
- r is a result:

$$\mathbb{R} := \mathbb{EV} \cup \{\mathit{error}\}$$
$$\mathbb{EV} := \mathbb{Z} \cup \mathbb{B} \cup \mathbb{CL}$$
$$\mathbb{B} := \{\mathit{true}, \mathit{false}\}$$

Specifying the Interpreter

Consider successor applied to 2

```
let succ=proc (x) {x+1}
in (succ 2)
```

Or, in abstract syntax:

```
Let ("succ",Proc ("x", Add (Var "x", Int 1)),App (Var "succ", Int 2))
```

$$\frac{}{\texttt{Proc(id,e)}, \rho \Downarrow ???}\ \mathit{EProc}$$

Specifying the Interpreter

Consider successor applied to 2

```
let succ=proc (x) {x+1}
in (succ 2)
```

$$\frac{}{\texttt{Proc(id,e)}, \rho \Downarrow (id, e)}\ \textit{EProc}$$

Reasonable attempt but not quite right

Consider:

```
let x = 2
in let f = proc (z) { z-x }
in let x = 1
in let g = proc (z) { z-x }
in (f 1) - (g 1)
```

Specifying the Interpreter

$$\frac{}{\texttt{Proc(id,e)}, \rho \Downarrow (\texttt{id}, \texttt{e}, \rho)} \; \textit{EProc}$$

- A closure is a triple with arguments:
  - a formal parameter `id`,
  - an expression `e`,
  - and an environment ρ (the one extant when the procedure was first evaluated).

Specifying the Interpreter

$$\frac{}{\texttt{Proc(id,e)}, \rho \Downarrow (\texttt{id}, \texttt{e}, \rho)}\ \textit{EProc}$$

```
let x = 2
in let f = proc (z) { z-x }
in let x = 1
in let g = proc (z) { z-x }
in (f 1) - (g 1)
```

`f` and `g` will evaluate to different closures

- `f`: $("z", Sub(Var "z", Var "x"), [x := 2])$
- `g`: $("z", Sub(Var "z", Var "x"), [x := 1])$

Specifying the Interpreter

$$\frac{\texttt{e1}, \rho \Downarrow (\texttt{id}, \texttt{e}, \sigma) \quad \texttt{e2}, \rho \Downarrow w \quad \texttt{e}, \sigma \oplus \{\texttt{id} := w\} \Downarrow v}{\texttt{App(e1,e2)}, \rho \Downarrow v} \; \textit{EApp}$$

$$\frac{\texttt{e1}, \rho \Downarrow v \quad v \notin \mathbb{CL}}{\texttt{App(e1,e2)}, \rho \Downarrow \textit{error}} \; \textit{EAppErr}$$

The Interpreter for PROC

LET

```
eval_expr: expr -> exp_val ea_result
```

PROC

```
eval_expr: expr -> exp_val ea_result
```

- What's the difference?
  - The definition of expressed values
- Before

$$\mathbb{EV} := \mathbb{Z} \cup \mathbb{B}$$

- Now

$$\mathbb{EV} := \mathbb{Z} \cup \mathbb{B} \cup \mathbb{CL}$$

Expressed Values

```
eval_expr: expr -> exp_val ea_result
```

- 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 string*expr*env
```

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

Implementation – Proc Case

$$\frac{}{\texttt{Proc(id,e)}, \rho \Downarrow (\texttt{id}, \texttt{e}, \rho)}\ \mathit{EProc}$$

```
| Proc(id,e) ->
  lookup_env >>= fun en ->
  return @@ ProcVal(id,e,en)
```

where `lookup_env` is defined in file `ds.ml`:

```
let lookup_env : env ea_result =
  fun env ->
  Ok env
```

Implementation

$$\frac{\texttt{e1}, \rho \Downarrow (\texttt{id}, \texttt{e}, \sigma) \quad \texttt{e2}, \rho \Downarrow w \quad \texttt{e}, \sigma \oplus \{\texttt{id} := w\} \Downarrow v}{\texttt{App(e1,e2)}, \rho \Downarrow v}\ \mathit{EApp}$$

```
| App(e1,e2)  ->
  eval_expr e1 >>= fun v1 ->
  eval_expr e2 >>= fun v2 ->
  apply_proc v1 v2
```

where `apply_proc` is:

```
let rec apply_proc : exp_val -> exp_val -> exp_val ea_result =
  fun f a ->
  match f with
  | ProcVal (id,body,env) ->
    return env >>+
    extend_env id a >>+
    eval_expr body
  | _ -> error "apply_proc: Not a procVal"
```

The Interpreter for PROC

- Code available in Canvas `Modules/Interpreters`
- Directory `proc-lang`
- Compile from root dir with `dune utop` or `dune utop src`
- Type, for eg.,
  `interp "let f = proc (x) { x-11 } in (f (f 77)) ";;` in utop