simPL: Syntax & Dynamic Semantics

YSC3208: Programming Language Design & Implementation

Răzvan Voicu

Week 3, Jan 23-27, 2017

1. The Language simPL
   - The Syntax of simPL
   - Pre-Processing
   - Some simPL Programs
   - Abstract Syntax Tree in OCaml
2. Dynamic Semantics of simPL

Motivation for simPL

- built-in conditionals,
- function definition and application, and
- recursive function definitions.

simPL allows us to study typing, realistic interpretation and virtual machines in detail.

Types

$$\frac{}{\texttt{int}} \qquad \frac{}{\texttt{bool}} \qquad \frac{t_1 \qquad t_2}{t_1\ \texttt{->}\ t_2}$$

Types in OCaml

We can define types for sPL in OCaml as follows:

```
type sPL_type =
    | BoolType
    | IntType
    | Arrow of sPL_type * sPL_type
```

Expressions (from ePL)

$$\frac{}{x} \qquad \frac{}{n} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}}$$

$$\frac{E}{p_1[E]} \qquad \frac{E_1 \quad E_2}{p_2[E_1, E_2]}$$

Expressions (cont'd)

$$\frac{E \qquad E_1 \qquad E_2}{\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end}}$$

$$\frac{E \quad E_1 \quad \cdots \quad E_n}{(E\ E_1 \cdots E_n)}$$

Expressions (cont'd)

$$\frac{E}{\texttt{fun } \{t_1\texttt{->}\cdots\texttt{->}t_n \texttt{ -> } t\}\ x_1 \cdots x_n \texttt{ -> } E \texttt{ end}}$$

if $t_1, \ldots, t_n$ and t are types, $n \geq 1$. The variables $x_1, \ldots, x_n$ must be pairwise distinct.

$$\frac{E}{\texttt{recfun } f\ \{t_1\texttt{->}\cdots\texttt{->}t_n \texttt{ -> } t\}\ x_1 \cdots x_n \texttt{ -> } E \texttt{ end}}$$

if $t_1, \ldots, t_n$ and t are types, $n \geq 1$. The variables $f, x_1, \cdots, x_n$ must be pairwise distinct.

Syntactic Conventions

- Parentheses
- Infix and prefix notation for operators

  `x + x * y > 10 - x`

  stands for

  `>[+[x,*[x,y]],-[10,x]]`
- `->` is right-associative, so that the type

  `int -> int -> int`

  is equivalent to

  `int -> (int -> int)`

Example

```
fun {int -> int -> int} x ->
   fun {int -> int} y -> x + y end
end
```

takes an integer `x` as argument and returns a function, whereas the function

```
fun {(int -> int) -> int} f -> (f 2) end
```

takes a function `f` as argument and returns an integer.

Pre-Processing

- Certain constructs can be viewed as syntactic sugar to make it easier to write programs.
- We can define a core language containing only essential features.
- Syntactic sugar can be translated away (into equivalent constructs in the core language).
- This phase can be done after type-checking.

Let Expressions

Let can be viewed as a *syntactic sugar* for method application.

$$\texttt{let}\ \{t_1\}\ x_1 = E_1 \cdots \{t_n\}\ x_n = E_n\ \texttt{in}\ \{t\}\ E\ \texttt{end}$$

stands for

$$\texttt{(fun}\ \{t_1\texttt{->}\cdots\texttt{->}t_n\ \texttt{->}\ t\}\ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end}\ E_1 \cdots E_n\texttt{)}$$

Partial Application

Partially applied function is allowed. We can treat it as syntactic sugar for anonymous function.
If f has type `t1->t2->t3->t4`

$$(f \;\; e1 \;\; e2)$$

can be translated to:

$$\textbf{fun} \;\; v \texttt{->} f \;\; e1 \;\; e2 \;\; v \;\; \textbf{end}$$

Preprocessor for Syntactic Sugar

Syntactic sugar can be transformed away after type checking, but prior to compilation.

```
let trans_exp (e:sPL_expr) : sPL_expr  =
  let rec aux e =
    match e with
      | BoolConst _ | IntConst _ | Var _-> e
      | UnaryPrimApp (op,arg) ->
            let varg = aux arg in
            (UnaryPrimApp (op,varg))
      | Cond (e1,e2,e3) ->
            let v1 = aux e1 in
            let v2 = aux e2 in
            let v3 = aux e3 in
            Cond (v1,v2,v3)
      | Let(defs,t,e) -> ...
      | Appln(e1,t1,es) -> ...
```

Example

```
let {int} AboutPi = 3
    {int -> int} Square =
         fun {int -> int} x -> x * x end
in  {int} 4 * AboutPi * (Square 6371)
end
```

Example (continued)

This example would have been translated to the following method application without any let construct.

```
(fun {int -> (int -> int) -> int}
    AboutPi Square
    ->
    4 * AboutPi * (Square 6371)
 end
 3
 fun {int -> int} x -> x * x end)
```

Power Function

We can also define recursive methods. An example is:

```
recfun power {int -> int -> int}
   x y ->
   if y = 0
   then 1
   else x * (power x y - 1)
   end
end
```

Higher-Order Functions

We can also define higher-order functions. An example is:

```
recfun recurse
  {int->int->(int->int->int)->int->int}
          x y op iv
  -> if y = 0 then iv
     else (op x
        (recurse x (y - 1) op iv))
     end
end
```

Abstract Syntax Tree in OCaml

```
type sPL_expr =
   | BoolConst of bool
   | IntConst of int
   | Var of id
   | UnaryPrimApp of op_id * sPL_expr
   | BinaryPrimApp of op_id * sPL_expr * sPL_expr
   | Cond of sPL_expr * sPL_expr * sPL_expr
   | Func of sPL_type * (id list) * sPL_expr
   | RecFunc of sPL_type * id * (id list) * sPL_expr
   | Appln of sPL_expr * sPL_type option
              * (sPL_expr list)
   | Let of ((sPL_type * id * sPL_expr) list)
              * sPL_type * sPL_expr
```

Values

In simPL, functions are values, although their bodies may not be values.

```
fun {int -> int} x -> 3 * 4 end
```

Note that we typically do not reduce inside the body of a method until it has been applied.

Value

A simPL value is:

- an integer, or
- a boolean value, or
- a function definition `fun` $\cdots$ `->` $\cdots$ `end`, or
- a recursive function definition `recfun` f $\cdots$ `->` $\cdots$ `end`).

Contraction

$$\frac{}{p_1[v_1] >_{\mathrm{simPL}} v}\,[\mathrm{OpVals}] \qquad \frac{}{p_2[v_1, v_2] >_{\mathrm{simPL}} v}\,[\mathrm{OpVals}]$$

$$\frac{}{\texttt{if true then } E_1 \texttt{ else } E_2 \texttt{ end} \quad >_{\mathrm{simPL}} \quad E_1}\,[\mathrm{IfTrue}]$$

Contraction (cont'd)

$$\frac{}{\texttt{if false then } E_1 \texttt{ else } E_2 \texttt{ end} \quad >_{\text{simPL}} \quad E_2}\ [\text{IfFalse}]$$

Contraction of Function Application

- Free variables
- Substitution
- Contraction of function application

Free Variables

Let us introduce a relation to extract free variables:

$$\bowtie: \mathrm{simPL} \times 2^{V}$$

Example

$\texttt{4 * (square x)} \bowtie \{\texttt{square, x}\}$

Read: “the set of free variables of the expression `4 * (square x)` is {square,x}.

Another Example

```
(fun {int -> int} x -> 4 * (square x) end 3
```

$\bowtie \{$`square`$\}$

Read: "the set of free variables of the expression `(fun {int -> int} x -> 4 * (square x) end 3` is $\{$`square`$\}$.

Definition of $\bowtie$

$$\frac{}{x \bowtie \{x\}} \qquad \frac{}{n \bowtie \emptyset} \qquad \frac{}{\texttt{true} \bowtie \emptyset} \qquad \frac{}{\texttt{false} \bowtie \emptyset}$$

Definition of $\bowtie$ (cont'd)

$$\frac{E \bowtie X}{p_1[E] \bowtie X} \qquad \frac{E_1 \bowtie X_1 \qquad E_2 \bowtie X_2}{p_2[E_1, E_2] \bowtie X_1 \cup X_2}$$

Definition of $\bowtie$ (cont'd)

$$\frac{E_1 \bowtie X_1 \qquad E_2 \bowtie X_2 \qquad E_3 \bowtie X_3}{\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ end} \bowtie X_1 \cup X_2 \cup X_3}$$

Definition of $\bowtie$ (cont'd)

$$\frac{E \bowtie X}{\texttt{fun}\ \{\cdot\}\ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end} \bowtie X - \{x_1, \ldots, x_n\}}$$

Definition of $\bowtie$ (cont'd)

$$\frac{E \bowtie X}{\texttt{recfun}\ \{\cdot\}\ f\ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end} \bowtie X - \{f, x_1, \ldots, x_n\}}$$

Free Variables Implementation in OCaml

```
let rec fv (e:sPL_expr) : id list =
  match e with
    | BoolConst _  | IntConst _ -> []
    | Var i -> [i]
    | UnaryPrimApp (_,arg) -> fv arg
    | BinaryPrimApp (_,arg1,arg2)
        -> (fv arg1)@(fv arg2)
    | Cond (e1,e2,e3) -> (fv e1)@(fv e2)@(fv e3)
    | Func (_,vs,body) -> diff (fv body) vs
    | RecFunc (_,i,vs,body)
        -> diff (fv body) (i::vs)
```

Free Variables in OCaml (cont.)

```
| AppIn (e1,_,es)
    -> (fv e1)@(List.concat (List.map fv es))

| Let (lst,_,body) ->
    let bv = List.map (fun (_,i,_) -> i) lst in
    let vs = List.concat ((fv body)
        ::(List.map (fun (_,i,e) -> fv e) lst))
    in diff vs bv
```

Subtracting Variables

Simple implementation of `diff` method.

```
(* removing vars in ys that occur in xs *)
let rec diff xs ys =
  match ys with
    | [] -> xs
    | y::ys -> diff
        (List.filter (fun v -> not(v=y)) xs) ys
```

Substitution

Goal: For function application, replace all free occurrences of the formal parameters in the function body by the actual arguments.

```
(fun {int -> int} x -> x * x end 4)
```

Replace every free occurrence of `x` in `x * x` by the actual parameter 4, resulting in

```
4 * 4
```

Substitution

Define the substitution relation

$$\cdot[\cdot \leftarrow \cdot] \rightsquigarrow \cdot : \text{simPL} \times V \times \text{simPL} \times \text{simPL}$$

such that $\texttt{x * x}[\texttt{x} \leftarrow \texttt{4}] \rightsquigarrow \texttt{4 * 4}$ holds.

Definition of Substitution

$$\frac{}{v\,[v \leftarrow E_1] \rightsquigarrow E_1}\ \text{for any variable } v$$

$$\frac{}{x\,[v \leftarrow E_1] \rightsquigarrow x}\ \text{for any variable } x \neq v$$

Definition of Substitution (cont'd)

$$\frac{E_1\,[v \leftarrow E] \rightsquigarrow E_1' \qquad E_2\,[v \leftarrow E] \rightsquigarrow E_2'}{(E_1\ E_2)\,[v \leftarrow E] \rightsquigarrow (E_1'\ E_2')}$$

Definition of Substitution (cont'd)

$$\frac{}{\texttt{fun}\ \{\cdot\}\ v\texttt{->}E\ \texttt{end}\,[v\leftarrow E_1] \rightsquigarrow \texttt{fun}\ \{\cdot\}\ v\ \texttt{->}\ E\ \texttt{end}}$$

Note that the above rule help avoids name clashes.

$$\frac{E\,[v\leftarrow E_1] \rightsquigarrow E' \qquad x \neq v \qquad E_1 \bowtie X_1 \qquad x \notin X_1}{\texttt{fun}\ \{\cdot\}\ x\texttt{->}E\ \texttt{end}\,[v\leftarrow E_1] \rightsquigarrow \texttt{fun}\ \{\cdot\}\ x\ \texttt{->}\ E'\ \texttt{end}}$$

Definition of Substitution (cont'd)

$$\frac{E_1 \bowtie X_1 \qquad x \in X_1 \qquad E \bowtie X \qquad E\,[x \leftarrow z] \leadsto E' \qquad E'\,[v \leftarrow E_1] \leadsto E'' \qquad x \neq v}{\texttt{fun}\ \{\cdot\}\ x \texttt{->} E\ \texttt{end}\,[v \leftarrow E_1] \leadsto \texttt{fun}\ \{\cdot\}\ z\ \texttt{->}\ E''\ \texttt{end}}$$

where we choose z such that $z \notin X_1 \cup X$. The renaming of x to a fresh z is to avoid a free x variable in E_1 being accidentally captured as a bound variable.

Examples

- Avoiding name clash:

  `fun {int -> int} factor -> factor * 4 * y end`
  `[factor` $\leftarrow$ `x + 1]` $\rightsquigarrow$
  `fun {int -> int} factor -> factor * 4 * y end`

- No name clash below:

  `fun {int -> int} factor -> factor * 4 * y end`
  `[y` $\leftarrow$ `x + 1]` $\rightsquigarrow$
  `fun {int -> int} factor -> factor * 4 * (x + 1) end`

Examples

- Avoiding name capture with a fresh naming:
  `fun {int -> int} factor -> factor * 4 * y end`
  [y← `factor + 1`] $\leadsto$
  ```
  fun {int -> int} newfactor ->
     newfactor * 4 * (factor + 1) end
  end
  ```

Substitution in OCaml

Substitution need to avoid name clashes and also perform renaming to avoid name capture.

```
let apply_subs
  (fnc:id list->id list-> id list * (id * id) list)
  (rename_op:(id*id) list->sPL_expr->sPL_expr)
  (ss:(id*sPL_expr)list)
  (e:sPL_expr) : sPL_expr =
```

Substitution in OCaml (cont.)

Some OCaml code on substitution

```
let rec aux ss e =
  match e with
    | BoolConst _ | IntConst _ -> e
    | Var i -> subs_var i ss
    | UnaryPrimApp (op,arg)
       -> UnaryPrimApp (op,aux ss arg)
    | BinaryPrimApp (op,arg1,arg2)
      -> BinaryPrimApp (op,aux ss arg1,aux ss arg2)
    | Cond (e1,e2,e3)
      -> Cond (aux ss e1,aux ss e2,aux ss e3)
    ...
```

Contraction of Function Application

$$\frac{E\,[x{\leftarrow}v] \rightsquigarrow E'}{(\texttt{fun}\ \{\cdot\}\ x\ \texttt{->}\ E\ \texttt{end}\quad v)\ >_{\mathrm{simPL}}\ E'}\ [\mathrm{CallFun}]$$

Contraction of Recursive Function Application

$$\frac{E\,[f \leftarrow \texttt{recfun}\ \{\cdot\}\ f\ x\ \texttt{->}\ E\ \texttt{end}] \rightsquigarrow E' \qquad E'\,[x \leftarrow v] \rightsquigarrow E''}{(\texttt{recfun}\ f\ x\ \texttt{->}\ E\ \texttt{end} \quad v) >_{\mathrm{simPL}} E''}\,[\mathrm{RF}]$$

One-Step Evaluation

$$\frac{E >_{\text{simPL}} E'}{E \mapsto_{\text{simPL}} E'} [\text{Contraction}]$$

$$\frac{E \mapsto_{\text{simPL}} E'}{p_1[E] \mapsto_{\text{simPL}} p_1[E']} [\text{OpArg}_1]$$

One-Step Evaluation (cont'd)

$$\frac{E_1 \mapsto_{\text{simPL}} E_1'}{p_2[E_1, E_2] \mapsto_{\text{simPL}} p_2[E_1', E_2]}\,[\text{OpArg}_2]$$

$$\frac{E_2 \mapsto_{\text{simPL}} E_2'}{p_2[v_1, E_2] \mapsto_{\text{simPL}} p_2[v_1, E_2']}\,[\text{OpArg}_3]$$

One-Step Evaluation (cont'd)

$$\frac{E \mapsto_{\text{simPL}} E'}{\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end} \mapsto_{\text{simPL}} \texttt{if}\ E'\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end}}$$

One-Step Evaluation (cont'd)

$$\frac{E \mapsto_{\text{simPL}} E'}{(E\ E_1 \ldots E_n) \mapsto_{\text{simPL}} (E'\ E_1 \ldots E_n)}\ [\text{AppFun}]$$

One-Step Evaluation (cont'd)

$$\frac{E_i \mapsto_{\text{simPL}} E_i'}{(v\ v_1 \ldots v_{i-1}\ E_i \ldots E_n) \mapsto_{\text{simPL}} (v\ v_1 \ldots v_{i-1}\ E_i' \ldots E_n)}\ [\text{AppArg}]$$

Evaluation of simPL Programs

As for ePL, evaluation of simPL is defined by the evalution relation $\mapsto^*_{\mathrm{simPL}}$, the reflexive transitive closure of $\mapsto_{\mathrm{simPL}}$.