Operational Semantics

CS 131: Programming Languages
October 4, 2001

Q&A

- Q:
  - Where is all this weird stuff heading today?
- A:
  - A *formal* definition of a nontrivial language.
    - In this case, a large fragment of a functional language like Scheme or SML.

Q&A

- Q:
  - What makes a definition "formal" ?
- A:
  - A definition can be considered formal if it is precise and unambiguous.
    - Because it is extremely difficult to do this with prose, we will use tools from mathematics or logic.

Q&A

- Q:
  - Why are we looking at functional languages?
- A:
  - Functional languages are simpler to formally describe.
    - C++ and Java don't have a good tiny core to start with; certainly nothing as simple as the λ-calculus
  - Also, useful to see where functional languages come from

Q&A

- Q:
  - What are the topics for next week?
- A:
  - Tuesday: Static and Dynamic Scope
    - or, "Why you shouldn't make decisions about language design based on interpreter efficiency."
  - Thursday: Assigning to Variables

Elements of a Language Definition

1. Syntax
   - Concrete syntax
   - Abstract syntax
2. Other constraints on valid programs
   - Typically a type system
3. What does the code actually mean?
   - What should happen when you execute the program?

Operational Semantics

- Method for explaining the "meaning" of code
- Explain how to

Abstract Syntax: notation

M,N ::=	x	*variables*
|	$\lambda x.M$	*functions*
|	M N	*applications*
|	0, 1, -1, ...	*integer constants*
|	M + N	*additions*
|	tt | ff	*booleans*
|	M == N	*equality-test*
|	if M then N_1 else N_2	*conditionals*
|	<M,N>	*pairs*
|	fst M	*first projection*
|	snd M	*second projection*
|	nil	*empty list*
|	let x=M in N	*local definitions*

Vs. Rex Concrete Syntax

M,N ::=	x	x
|	λx.M	(x) => M
|	M N	M(N)
|	0, 1, -1, ...	0, 1, -1, ...
|	M + N	M + N
|	tt | ff	0, 1
|	M == N	M == N
|	if M then N_1 else N_2	M ? N_1 : N_2
|	<M,N>	[M | N]
|	fst M	first(M)
|	snd M	rest(M)
|	nil	[]
|	let x=M in N	x=M, N

SML Implementation

```
datatype absyn =
   Var   of string
 | Lam   of string*absyn
 | App   of absyn*absyn
 | Num   of int
 | Plus  of absyn*absyn
 | Equal of absyn*absyn
 | If    of absyn*absyn*absyn
 | Bool  of bool
 | Pair  of absyn*absyn
 | Fst   of absyn
 | Snd   of absyn
 | Nil
 | Let   of string*absyn*absyn
```

Abstract Syntax Example

λg.(λx.if (x=nil) then 0 else 1+g(snd(x)))

```
Lam("g",
    Lam("x",
        If(Equal(Var "x", Nil),
           Num 0,
           Plus(Num 1, App(Var "g", Snd(Var "x"))))))
```

The Values

V ::=

A Big-Step Semantics

- Also known as natural semantics.
- Instead of specifying how to take one step of computation, directly define the "evaluates to" relation.

$$M \Downarrow v$$

CBV Big-Step Semantics

$$\frac{}{v \Downarrow v} \qquad \frac{M \Downarrow m \quad N \Downarrow n}{M+N \Downarrow m \oplus n} \qquad \frac{M \Downarrow v_1 \quad N \Downarrow v_2}{M == N \Downarrow v_1 \equiv v_2}$$

$$\frac{M \Downarrow tt \quad N_1 \Downarrow v}{\text{if } M \text{ then } N_1 \text{ else } N_2 \Downarrow v}$$

$$\frac{M \Downarrow ff \quad N_2 \Downarrow v}{\text{if } M \text{ then } N_1 \text{ else } N_2 \Downarrow v}$$

$$\frac{M \Downarrow v_1 \quad N[x \to v_1] \Downarrow v_2}{\text{let } x=M \text{ in } N \Downarrow v_2}$$

$$\frac{M \Downarrow \lambda x.M' \quad N \Downarrow v_2 \quad M'[x \to v_1] \Downarrow v}{M(N) \Downarrow v}$$

$$\frac{M_1 \Downarrow v_1 \quad M_2 \Downarrow v_2}{\langle M_1, M_2 \rangle \Downarrow \langle v_1, v_2 \rangle}$$

$$\frac{M \Downarrow \langle v_1, v_2 \rangle}{fst(M) \Downarrow v_1} \qquad \frac{M \Downarrow \langle v_1, v_2 \rangle}{snd(M) \Downarrow v_2}$$

A Big-Step Interpreter: Values

```
exception Error

fun eval expr =
   (case expr of
       Num n => Num n
     | Bool b => Bool b
     | Nil => e
     | Lam _ => e


     | Var _ => raise Error
```

$$\frac{}{v \Downarrow v}$$

Addition

```
| Plus(e1,e2) =>
   let
     val v1 = eval e1
     val v2 = eval e2
   in
     (case (v1,v2) of
        (Num m1, Num m2) => Num (m1+m2)
      | _                => raise Error)
   end
```

$$\frac{M \Downarrow m \quad N \Downarrow n}{M+N \Downarrow m \oplus n}$$

Equality Test

```
| Equal(e1,e2) =>
    let
      val v1 = eval e1
      val v2 = eval e2
    in
      (case (v1,v2) of
         (Num m1, Num m2) => Bool(m1=m2)
       | _                => raise Error)
    end
```

$$\frac{M \Downarrow v_1 \quad N \Downarrow v_2}{M == N \Downarrow v_1 \equiv v_2}$$

Conditional

$$\frac{M \Downarrow tt \quad N_1 \Downarrow v}{\text{if } M \text{ then } N_1 \text{ else } N_2 \Downarrow v}$$

$$\frac{M \Downarrow ff \quad N_2 \Downarrow v}{\text{if } M \text{ then } N_1 \text{ else } N_2 \Downarrow v}$$

```
| If(M,N1,N2) =>
   (case eval M of
      Bool true  => eval N1
    | Bool false => eval N2
    | _          => raise Error)
```

Application

$$\frac{M \Downarrow \lambda x.M' \quad N \Downarrow v_2 \quad M'[x \to v_1] \Downarrow v}{M(N) \Downarrow v}$$

```
| Apply(M,N) =>
     let
        val v1 = eval e1
        val v2 = eval e2
     in
        (case v1 of
           Lam(x,M') => eval(subst(M',x,v2))
         | _         => raise Error)
     end
```

Local Definitions

$$\frac{M \Downarrow v_1 \quad N[x \to v_1] \Downarrow v_2}{\texttt{let } x\texttt{=}M \texttt{ in } N \Downarrow v_2}$$

```
| Let(x,M,N) =>
   let
     val v1 = eval M
     val v2 = eval(subst(N,x,v1))
   in
     v2
   end
```

Pairs and Projections

$$\frac{M_1 \Downarrow v_1 \quad M_2 \Downarrow v_2}{\langle M_1, M_2 \rangle \Downarrow \langle v_1, v_2 \rangle}$$

```
| Pair(e1,e2) => Pair(eval e1, eval e2)
```

```
| Fst M => (case (eval M) of
              Pair(v1,_) => v1
            | _ => raise Error)
```

$$\frac{M \Downarrow \langle v_1, v_2 \rangle}{\texttt{fst}(M) \Downarrow v_1}$$

```
| Snd M => (case (eval M) of
              Pair(_,v2) => v2
            | _ => raise Error))
```

$$\frac{M \Downarrow \langle v_1, v_2 \rangle}{\texttt{snd}(M) \Downarrow v_2}$$

How Could We Be Lazier?