Parameter Passing and Delayed Computations

April 16, 2001

CS 131: Programming Languages

Review: Reduction Strategies

- Depending on choice of reductions, may or may not reach a normal form.

  $(\lambda x.\ulcorner 0\urcorner)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta \ulcorner 0\urcorner$

  $(\lambda x.\ulcorner 0\urcorner)((\lambda x.xx)(\lambda x.xx))$
  $\rightarrow_\beta (\lambda x.\ulcorner 0\urcorner)((\lambda x.xx)(\lambda x.xx))$
  $\rightarrow_\beta (\lambda x.\ulcorner 0\urcorner)((\lambda x.xx)(\lambda x.xx))$
  $\rightarrow_\beta \ldots$

Review: Reduction Strategies

- Two strategies that do yield a normal form may take a different number of steps

  $(\lambda y.yyy)((\lambda x.x)(\lambda x.x))$
  $\rightarrow_\beta ((\lambda x.x)(\lambda x.x))((\lambda x.x)(\lambda x.x))((\lambda x.x)(\lambda x.x))$
  $\rightarrow_\beta \ldots \rightarrow_\beta (\lambda x.x)$ [6 steps total]

  $(\lambda y.yyy)((\lambda x.x)(\lambda x.x))$
  $\rightarrow_\beta (\lambda y.yyy)(\lambda x.x)$
  $\rightarrow_\beta (\lambda x.x)(\lambda x.x)(\lambda x.x)$
  $\rightarrow_\beta (\lambda x.x)(\lambda x.x) \rightarrow_\beta (\lambda x.x)$ [4 steps total]

Review: Reduction Strategies

- Call-by-Name
  - Substitutes in function arguments immediately
  - Never reduces a function argument until it is needed.
  - Argument evaluated repeatedly if it is used repeatedly.
- Call-by-Value
  - Apply beta-reduction only when the argument is a value (e.g., normal form)
  - May spend time evaluating an argument that never gets used.

Evaluation in PLs

- Lambda calculus forms a basis for all functional programming languages
  - And languages with a procedure-call mechanism
- Can classify languages according to their evaluation strategy
  - Algol 60: Call-by-name
  - SML, C: Call-by-value

Evaluation in NQSML

- As defined, NQSML was call-by-value.

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}$$

$$\frac{}{(\textbf{fix}\ \mathrm{f(x)}\ \textbf{is}\ \mathrm{e})\ v \rightarrow e[x \rightarrow v][\mathrm{f} \rightarrow \textbf{fix}\ \mathrm{f(x)}\ \textbf{is}\ \mathrm{e}]}$$

Evaluation in NQSML

- But we could change it to call-by-name:

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2}$$

$$\frac{}{(\textbf{fix}\ \mathrm{f(x)}\ \textbf{is}\ \mathrm{e})\ e_2 \rightarrow e[x \rightarrow e_2][\mathrm{f} \rightarrow \textbf{fix}\ \mathrm{f(x)}\ \textbf{is}\ \mathrm{e}]}$$

Evaluation in SML

- What is the result of the following SML code?

```
fun loop() = loop()

fun f _ = 3

val x = f (loop())
```

Evaluation in Haskell

- The Haskell language is very like SML, but is call-by-name.
  - Officially, no side-effects
  - Slightly different concrete syntax

```
loop() = loop()

f _ = 3

x = f (loop())
```

Evaluation in ALGOL 60

- Assume we call id in a context where `i` is an integer variable and A is an array of reals.
  - Compare with macro expansion.

```
real procedure id(r);
  real r;
begin
  integer i;
  i := 7;
  id := r
end id;
...
m := id(3);
n := id(A[i]);
```

Evaluation in ALGOL 60

- What does the following code do?

```
real procedure sum(expr,i,low,high);
  real expr
  integer i, low, high;
begin
  real rtn
  rtn := 0;
  for i := low step 1 until high do
     r := rtn + expr;
  sum := rtn;
end sum
```

Evaluation in ALGOL 60

- What does the following procedure calls compute?

```
y := sum(x*x,x,1,10)

w := sum(sum(B[j,k],j,1,20),k,1,20)
```

- This trick is known as "Jensen's device"

Evaluation in ALGOL 60

- The following code looks ok.

```
procedure swap(a,b);
  real a, b;
begin
  real temp;
  temp := a;
  a := b;
  b := temp;
end swap;
...
swap(n,m);
swap(A[1],A[2]);
```

Evaluation in ALGOL 60

- But the code does not work as one would expect for all inputs
  - One *cannot* write swap to work for all inputs!
- What's the problem?

- Common (but not universal) conclusion:
  - Having call-by-name in languages with assignment and other side-effects is too confusing.
  - Also carries implementation overhead.

Digression

- Other parameter passing conventions
  - All have call-by-value like evaluation order
  - Only difference is exactly what value is passed

```
SUBROUTINE F(X,Y)
X = X+1
Y = Y+1
RETURN
END
...
Z = 3
W = 4
CALL F(Z,W)
```

Call-By-Reference

- Used by FORTRAN, optional in Pascal and C++
  - Implicitly passing pointers to the arguments
  - Formal parameters are aliases of the actual parameters.

```
SUBROUTINE F(X,Y)
X = X+1
Y = Y+1
RETURN
END
...
Z = 3
W = 4
CALL F(Z,W)
```

Call-By-Value/Result

- Optional in Algol-W and Ada (`IN OUT`)
  - Like call-by-value except that actual parameters are updated once when subroutine ends.

```
SUBROUTINE F(X,Y)
X = X+1
Y = Y+1
RETURN
END
...
Z = 3
W = 4
CALL F(Z,W)
```

Exercise

- What is the value of Z after the function call?
  - Consider each mode of parameter passing

```
SUBROUTINE F(X,Y)
X = X+1
Y = Y+1
RETURN
END
...
Z = 3
W = 4
CALL F(Z,Z)
```

Simulating Call-by-Name

- The argument to a call-by-name procedure is not a value, but a *computation* that may result in a value.
- We can simulate this in SML when desired by using a type of "suspended computation"

```
type 'a susp

val force : 'a susp -> 'a

val delay :
```

Implementation 1

```
type 'a susp = unit -> 'a

fun force (s : 'a susp) = s()
```

Optimization: Call-by-Need

- Idea:
  - Remember the result the first time a delayed computation is forced
  - Return this result each for each successive `force`
- Note
  - If the delayed computation has no side-effects, (other than potentially nontermination) then cannot distinguish call-by-name from call-by-need!
  - Haskell actually implemented with call-by-need.

By-Need Implementation 1

```
type 'a susp =
```

By-Need Implementation 2

```
type 'a susp =
```