Function Abstraction

Parameter Passing

Lecture #11, Spring 2015
Sara Porat
What is an Abstraction?

- **A mode of thought:** concentrate on general ideas rather than on the specific *manifestation* of these ideas
  - Philosophy *ideas*
  - Mathematics

- **Abstraction in systems analysis and design**
  - Focus on essential aspects of the problem on hand, and ignore extraneous aspects
  - Example: air traffic control
    - Essential: aircraft’s type, destination, location in space, velocity, …
    - Irrelevant: color, names of passengers

- **Abstraction in programming**
  - Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time
  - A program can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer
  - For example, low-level abstraction layers expose details of the hardware where the program runs, while high-level layers deal with the business logic of the program
Procedure and Function Abstraction

- **Procedures and functions**: entities which embody computation
  - Function Abstraction: embodies an *expression* to be evaluated
  - Procedure Abstraction: embodies a *command* to be executed

- The embodied computation is performed whenever the abstraction is **called**

- **Separation of concerns**:
  - Implementer is concerned with *how* the computation is to be performed
  - Caller (user) is concerned with *what* the computation does

- Effectiveness is enhanced by *parameterization*
Parameters

Parameterization: generalizing the abstraction

val pi = 3.14159;
val r = 1.0;
fun circum1() = 2 * pi * r;
  The function call circum1() always performs the same computation
fun circum (r: real) = 2 * pi * r;
circum (1.0);
circum (a+b);
Arguments

- An **argument** is a value that may be passed to an abstraction
- **Pascal**
  - primitive values
  - composite values (except files)
  - pointers
  - references to variables
  - procedure and function abstractions
  - i.e. all Pascal values except files
- **ML**
  - primitive values
  - composite values
  - references to variables
  - function abstractions
  - i.e. all ML values
Parameter Passing

- The associations between formal and actual parameters
- A variety of mechanisms
  - by value
  - by result
  - by value-result
  - by reference
  - constant parameters
  - procedural/functional parameters
  - ...

- Can all be described in terms of two concepts:
  - Copy, e.g. by-value
  - Definitional, e.g. by-reference
Copy Mechanisms

- Allows for values to be copied into and/or out of an abstraction when it is called

**Formal parameter:** a *local* variable
- A value is copied into the variable on entry to the abstraction, and/or is copied out of the variable (to a nonlocal variable) on exit from the abstraction
- Any updating of the local variable has no effect on any nonlocal variable

**Actual parameter:**
- *value* parameter
  - legality: any first-class value
  - entry effect: evaluated, value assigned to formal parameter
  - exit effect: none
- *result* parameter
  - legality: a variable
  - entry effect: none (initial value is undefined)
  - exit effect: final value of the local variable assigned to argument variable (actual parameter)
- *value-result* parameter
  - legality: a variable
  - entry effect: value assigned to formal parameter
  - exit effect: returned value assigned to actual parameter
Pseudo Pascal Example

```pascal
type Vector = array[1..n] of Real;

procedure add(value v, w: Vector, result sum: Vector);
var i: 1..n;
begin
  for i := 1 to n do
    sum[i] := v[i] + w[i];
end;

procedure normalize(value result u: Vector)
var
  i: 1..n;
  s: Real;
begin
  s := 0.0;
  for i:=1 to n do
    s := s+sqr(u[i]);
  s := sqrt(s);
  for i:=1 to n do
    u[i] := u[i]/s;
end;
```

What’s the effect of
```pascal
add(a,b,c)
normalize(c)
```
Definitional Mechanisms

- A formal parameter (X) is bound directly to an actual parameter
  - Suitable to all values in the programming language (not just first class values)
  - *constant* parameter
    - argument is (first-class) value
    - X is bound to the *value* during activation
  - *variable (reference)* parameter
    - argument is a reference to a variable (Y)
    - X is bound to the argument variable (*reference* of Y) during activation
    - any inspection/update of X actually inspects/updates Y
  - *procedural/functional* parameter
    - argument is a procedure/function abstraction (Y)
    - X is bound to Y during activation
    - any call to X is an indirect call to Y

- The effect is as if the abstraction body were surrounded by a block in which there is a definition that binds X to the argument – hence the terminology *definitional mechanism*
Pseudo-Pascal Example

**type** Vector = array[1..n] of Real;

**procedure** add(const v, w: Vector, var sum: Vector);
var i: 1..n;
begin
  for i := 1 to n do
    sum[i] := v[i] + w[i];
end;

**type** Vector = array[1..n] of Real;

**procedure** normalize(var u: Vector);
var
  i: 1..n;
  s: Real;
begin
  s := 0.0;
  for i:=1 to n do
    s := s+sqr(u[i]);
  s := sqrt(s);
  for i:=1 to n do
    u[i] := u[i]/s;
end;

What’s the effect of
add(a,b,c)
normalize(c)
now?
procedure confuse1(var a,b: Integer)
var
begin
  a := 1;
  a := b+a;
end;

Procedure confuse2(var a,b: Integer)
Begin
  a := b+1;
End;

var x,y: Integer;
begin
  ... 
  confuse1(x,y);  // like confuse2(x,y)
  confuse1(x,x);  // unlike confuse2(x,x)
end;
procedure swap1(var a,b: Integer)
var
tmp: Integer;
begin
tmp := a;
a := b;
b := tmp;
end;

procedure swap2(var a,b: Integer)
begin
    a := a + b;
b := a - b;
a := a - b;
end;

var x,y: Integer;
begin
    ...
    swap1(x,y);
    swap2(x,y);
    swap1(x,x);
    swap2(y,y);
end;
Why Is Aliasing a Problem?

- “I will never write something as silly as `swap(x, x)`, so why worry?”

- Did you ever write a line like `swap(x[i], x[j])`?
  - Can you tell in advance that `i` will never equal `j` in this line?

- What about `swap(*pa, *pb)`?
  - Can `pa` and `pb` be pointers to the same object?

- No compiler can accurately detect aliasing in general
More Aliasing Problems...

What’s wrong with the following definition of operator= in a C++ class?

```cpp
class Person {
    Address* addr;
    Person& operator=(const Person& rhs) {
        delete addr; // release original object
        addr = new Address(*(rhs.addr));

        return *this;
    }
};
```

(See: Scott Meyers, Effective C++, 3rd ed., item 11, p.53)
Call-by-name (Algol 60)

- Textually substitute the actual argument expressions for the corresponding formal parameters in the body of the procedure

```plaintext
procedure double(x);
    real x;
begin
    x := x * 2
end;
```

double(C[j]) is interpreted as C[j] := C[j] * 2
Call-by-name (cont’)

- Implications of the pass-by-name mechanism:
  - Normal-order evaluation
  - The procedure can change the values of variables used in the argument expression and hence change the expression's value

```plaintext
procedure swap (a, b);
integer a, b, temp;
begin
  temp := a;
a := b;
b := temp
end;
```

- Effect of the call `swap(x, y):`
  - `temp := x; x := y; y := temp`

- Effect of the call `swap(i, x[i]):`
  - `temp := i; i := x[i]; x[i] := temp`

- It doesn't work!
Passing expressions into a routine so they can be repeatedly evaluated has some valuable applications.

Consider calculations of the form:
- "sum $x[i] \times i$ for all $i$ from 1 to $n$"
- $\text{Sum}(i, 1, n, x[i] \times i)$?
- Using pass-by-reference or pass-by-value, we cannot do this because we would be passing in only a single value of $x[i] \times i$, not an expression which can be repeatedly evaluated as "$i$" changes.

Using pass-by-name, the expression $x[i] \times i$ is passed in without evaluation.

```pascal
real procedure Sum(j, lo, hi, Ej);
value lo, hi;
integer j, lo, hi; real Ej;
begin
real S;
S := 0;
for j := lo step 1 until hi do
  S := S + Ej;
Sum := S
end;
```

For $\text{Sum}(i, 1, n, x[i] \times i)$
Each time through the loop, evaluation of $Ej$ is actually the evaluation of the expression $x[i] \times i$.
Strict vs. Non-Strict Functions

- **Strict function in the** \( n^{\text{th}} \) **argument**: a function \( f \) is said to be **strict** in its \( n^{\text{th}} \) argument if it cannot be evaluated unless this argument is evaluated.

- **Non-Strict in the** \( n^{\text{th}} \) **argument**: a function \( f \) is said to be **non-strict** in its \( n^{\text{th}} \) argument if there are cases in which this argument cannot be evaluated but \( f \) can.

```ml
fun cand (b1: bool, b2: bool) = 
  if b1 then b2 else false
```

cand(n > 0, t/n > 0.5)

<table>
<thead>
<tr>
<th>n</th>
<th>t</th>
<th>eager</th>
<th>normal-order</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\star)</td>
<td>2</td>
<td>0.8</td>
<td>false</td>
</tr>
<tr>
<td>(\star)</td>
<td>0</td>
<td>0.8</td>
<td><strong>FAIL!</strong></td>
</tr>
</tbody>
</table>

- The function **cand** is
  - **strict** in its first argument
  - **non-strict** in its second argument