Programming Languages: Abstraction

Onur Tolga Şehitoğlu

Computer Engineering

18 March 2008
Iceberg: Details at the bottom, useful part at the top of the ocean. Animals do not care about the bottom.

User: “how do I use it?”, Developer: “How do I make it work?”

User: “what does it do?”, Developer: “How does it do that?”

Abstraction: Make a program or design reusable by enclosing it in a body, hiding the details, and defining a mechanism to access it.

Separating the usage and implementation of program segments.

Vital large scale programming.
Abstraction is possible in any discipline involving design:

- radio tuner. Adjustment knob on a radio is an abstraction over the tuner element, frequency selection.

- An ATM is an abstraction over complicated set of bank transaction operations.

- Programming languages can be considered as abstraction over machine language.

- ...
Purpose

- Details are confusing
- Details may contain more error
- Repeating same details increase complexity and errors
- Abstraction philosophy: Declare once, use many times!
- Code reusability is the ultimate goal.
- Parameterization improves power of abstraction
Function and procedure abstractions

- The computation of an expression is the detail (algorithm, variables, etc.)
- Function call is the usage of the detail
- Functions are abstractions over expressions
- void functions of C or procedure declarations of some languages
- No value but contains executable statements as detail.
- Procedures are abstractions over commands
- Other type of abstractions possible?
Selector abstraction

- Arrays: `int a[10][20]; a[i]=a[i]+1;`
- `[..]` operator selects elements of an array.
- User defined selectors on user defined structures?
- Example: Selector on a linked list:

```c
int & get(List *p, int el) { /* linked list */
    int i;
    for (i=1; i<el; i++) {
        p = p->next; /* take the next element */
    }
    return p->data;
}
get(head, i) = get(head, 2) + 1; ...
```

- C++ allows overloading of `[]` operator for classes.
Generic abstraction

- Same declaration pattern applied to different data types.
- Abstraction over declaration. A function or class declaration can be adapted to different types or values by using type or value parameters.

```cpp
template <class T>
class List {
    T content;
    List *next;
public: List() { next=NULL; }
    void add(T el) { ... };
    T get(int n) { ...};
};

template <class U>
void swap(U &a, U &b) { U tmp; tmp=a; a=b; b=tmp; }
...
List<int> a; List<double> b; List<Person> c;
int t,x; double v,y; Person z,w;
swap(t,x); swap(v,y); swap(z,w);
```
### Iterator abstraction

- Iteration over a user defined data structure. Ruby example:

```ruby
class Tree
  def initialize(v)
    @value = v; @left = nil; @right = nil
  end
  def traverse
    @left.traverse {|v| yield v} if @left != nil
    yield @value # block argument replaces
    @right.traverse {|v| yield v} if @right != nil
  end
end

a = Tree.new(3); l = []
a.traverse { |node| # yield param
  print node # yield body
  l << node # yield body
}
```
If any programming language entity involves computation, it is possible to define an abstraction over it.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Function</td>
</tr>
<tr>
<td>Command</td>
<td>Procedure</td>
</tr>
<tr>
<td>Selector</td>
<td>Selector function</td>
</tr>
<tr>
<td>Declaration</td>
<td>Generic</td>
</tr>
<tr>
<td>Command Block</td>
<td>Iterator</td>
</tr>
</tbody>
</table>
Many purpose and behaviors in order to take advantage of “declare once use many times”.

**Declaration part:** abstractionname(Fp_1, Fp_2, ..., Fp_n)

**Use part:** abstractionname(Ap_1, Ap_2, ..., Ap_n)

- **Formal parameters:** identifiers or constructors of identifiers (patterns in functional languages)
- **Actual parameters:** expression or identifier based on the type of the abstraction and parameter

**Question:** How actual and formal parameters relate/communicate?

**Programming language design should answer**

**Parameter passing mechanisms**
Parameter Passing Mechanisms

Programming language may support one or more mechanisms. 3 basic methods:

1. Copy mechanisms (assignment based)
2. Binding mechanisms
3. Pass by name (substitution based)
Copy Mechanisms

- Function and procedure abstractions, assignment between actual and formal parameter:

  1. **Copy In:**
     On function call: \( Fp_i \leftarrow Ap_i \)

  2. **Copy Out:**
     On function return: \( Ap_i \leftarrow Fp_i \)

  3. **Copy In-Out:**
     On function call: \( Fp_i \leftarrow Ap_i \), and
     On function return: \( Ap_i \leftarrow Fp_i \)

- C only allows copy-in mechanism. This mechanism is also called as **Pass by value.**
int x=1, y=2;
void f(int a, int b) {
    x += a+b;
    a++;
    b=a/2;
}
int main() {
    f(x,y);
    printf("x:%d, y:%d\n", x, y);
    return 0;
}
- Based on binding of the formal parameter variable/identifier to actual parameter value/identifier.
- Only one entity (value, variable, type) exists with more than one names.
  1. **Constant binding**: Formal parameter is constant during the function. The value is bound to actual parameter expression value. Functional languages including Haskell uses this mechanism.
  2. **Variable binding**: Formal parameter variable is bound to the actual parameter variable. Same memory area is shared by two variable references. Also known as *pass by reference*
- The other type and entities (function, type, etc) are passed with similar mechanisms.
int x=1, y=2;
void f(int a, int b) {
    x += a+b;
    a++;  
b = a/2;
}
int main() {
    f(x, y);
    printf("x:%d, y:%d\n", x, y);
    return 0;
}

Variable binding:

<table>
<thead>
<tr>
<th></th>
<th>f():a</th>
<th>f():b</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>x: 5</td>
<td>y: 2</td>
<td></td>
</tr>
</tbody>
</table>
Pass by name

- Actual parameter syntax replaces each occurrence of the formal parameter in the function body, then the function body evaluated.
- C macros work with a similar mechanism (by pre-processor)
- Mostly useful in theoretical analysis of PL's. Also known as Normal order evaluation
- Example (Haskell-like)

\[
f \times y = \begin{cases} 
\text{if } (x < 12) & \text{then } x^2 + y^2 + x \\
\text{else } x + x^2 
\end{cases}
\]

Evaluation: \( f \ (3 \times 12 + 7) \ (24 + 16 \times 3) \mapsto \begin{cases} 
\text{if } ((3 \times 12 + 7) < 12) & \text{then} \\
(3 \times 12 + 7) \times (3 \times 12 + 7) + (24 + 16 \times 3) \times (24 + 16 \times 3) + (3 \times 12 + 7) \\
\text{else} & (3 \times 12 + 7) + (3 \times 12 + 7) \times (3 \times 12 + 7) 
\end{cases}
\mapsto \begin{cases} 
\text{if } (43 < 12) & \text{then} \\
(3 \times 12 + 7) + (3 \times 12 + 7) \times (3 \times 12 + 7) \mapsto (3 \times 12 + 7) + 43 \times (3 \times 12 + 7) \mapsto \\
\text{else} & \\
\text{if } \text{false} \mapsto (3 \times 12 + 7) + 43 \times (3 \times 12 + 7) \mapsto \\
\text{then} & \text{false} \mapsto (3 \times 12 + 7) + 43 \times (3 \times 12 + 7) \mapsto \\
\text{false} & \\
\text{false} \mapsto 1892
\)
Evaluation Order

- **Normal order evaluation** is mathematically natural order of evaluation.

- Most of the PL’s apply **eager evaluation**: Actual parameters are evaluated first, then passed.

  \[
f (3*12+7)(24+16*3) \mapsto f (36+7)(24+16*3) \mapsto f 43 72 \mapsto \text{if (43<12)}\]

  \[
  \text{then } 43*43+72*72+43 \text{ else } 43+43*43 \mapsto \text{if (false) then ... } \mapsto 43+43*43 \mapsto 1892
  \]

- Consider “\(g x y = \text{if } x>10 \text{ then } y \text{ else } x\)” for \(g \ 2 \ (4/0)\)

- **Side effects are repeated in NOE.**

- **Church–Rosser Property**: If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing eager and normal-order evaluation), then all of these evaluation orders yield the same result.
Haskell implements **Lazy Evaluation** order.

Eager evaluation is faster than normal order evaluation but violates Church-Rosser Property. Lazy evaluation is as fast as eager evaluation but computes same results with normal order evaluation (unless there is a side effect)

Lazy evaluation expands the expression as normal order evaluation however once it evaluates the formal parameter value other evaluations use previously found value:

\[
f (3 \times 12 + 7) (24 + 16 \times 3) \mapsto \text{if } (x : (3 \times 12 + 7) < 12) \text{ then } \]
\[
x : (3 \times 12 + 7) \times x : (3 \times 12 + 7) + y : (24 + 16 \times 3) \times y : (24 + 16 \times 3) + x : (3 \times 12 + 7) \text{ else } \]
\[
x : (3 \times 12 + 7) + x : (3 \times 12 + 7) \times x : (3 \times 12 + 7) \mapsto \text{if } (x : 43 < 12) \text{ then } \]
\[
x : 43 \times x : 43 + y : (24 + 16 \times 3) \times y : (24 + 16 \times 3) + x : 43 \text{ else } x : 43 + x : 43 \times x : 43 \mapsto \text{if } \]
\[
\text{(false) then } \ldots \mapsto x : 43 + x : 43 \times x : 43 \mapsto x : 43 + 1849 \mapsto 1892 \quad (7 \text{ steps})
Correspondence Principle

- **Correspondence Principle:**
  For each form of declaration there exists a corresponding parameter mechanism.

- **C:**
  - `int a=p;` ↔ `void f(int a) {`
  - `const int a=p;` ↔ `void f(const int a) {`

- **Pascal:**
  - `var a: integer;` ↔ `procedure f(a:integer) begin`
  - `const a:5;` ↔ `??? {`
  - `???` ↔ `procedure f(var a:integer) begin`

- **C++:**
  - `int a=p;` ↔ `void f(int a) {`
  - `const int a=p;` ↔ `void f(const int a) {`
  - `int &a=p;` ↔ `void f(int &a) {`