On Functions and their Evaluation

Luca Abeni
`luca.abeni@santannapisa.it`

Function Application, Again

- Application of function "`f`" to actual parameter "`x`"
  - Notice: 1 single argument... And this is not a restriction! Why?
- In C-like languages, we are used to "`f(x);`", but...
  - Are the parentheses really needed, here?
  - In case of "`f(x + y)`", they are needed to make a distinction with "`f(x) + y`", but for "`f(x)`"...
- Some languages do not require these "useless" parentheses: `f(x)` $\rightarrow$ `f x`
- In some other languages, the parentheses go around the application: `f(x)` $\rightarrow$ `(f x)`
  - Can you see where LISPs are coming from, now?

More Complex Expressions

- The C-style syntax for function application makes it simple to understand function composition
    - If $h = f \circ g$, then $h(x)$ is coded as `f(g(x))`!
- If parentheses are removed, then some associativity rules are needed
    - Does “`f g h`” mean “`f(g(h))`”, or “`(f(g))(h)`”?
    - If left associativity is used, then currying has a natural syntax: “`sum_c a b`” means “`(sum_c a) b`”, making the usage of curried functions pretty simple!
- With parentheses around function application, we have things like “`((sum_c a) b)`”

Example

- Assume that `K x y = x` and `S p q r = p r (q r)` ...
- What is the value of `S K K a`?

`S K K a` →

`(p r (q r))` with “`p`” replaced by “`K`”, “`q`” replaced by “`K`” and “`r`” replaced by “`a` →

`(K r (q r))` with “`q`” replaced by “`K`” and “`r`” replaced by “`a` →

`(K r (K r))` with “`r`” replaced by “`a` →

`K a (K a)` →

`x` with “`x`” replaced by “`a`” and “`(K a)`” discarded →

`a`

Formal Arguments and Actual Arguments

- If $f(x) = x + 1$, applying “`f`” to “`2`” requires to:
  - Replace “`f`” (function name) with “`x + 1`” (function body)
  - Replace “`x`” (formal parameter) with “`2`” (actual parameter)
  - Compute the result $2 + 1 = 3$
- In C-like languages, we are used to look at function invocation in a different way:
  - Push “`2`” (actual parameter) on the stack
  - Call the function body (which pulls the parameter’s from the stack)
  - Different argument-passing methods

Passing Parameters by Value

- Only possible method in C
- One local variable is allocated (on the stack) when the function is called
- The local environment contains a binding between the formal parameter’s name and this local variable
- The variable is automagically initialized with the value of the actual parameter

```
int f(int n)
{
  n = n + 1;

  return n * 2;
}
```

Passing Parameters by Reference

- Possible in C++
- No local variable for the formal parameter
- The local environment contains a binding between the formal parameter's name and the actual parameter
    - The actual parameter must be an L-Value
    - The formal parameter is an alias for the actual parameter

```
int f(int &n)
{
  n = n + 1;

  return n * 2;
}
```

Passing Pointers by Value

- “Emulation” of reference passing in C
- A pointer to the “real” actual parameter is passed by value
- First difference with parameter passing by reference: syntax
    - But there are other notable differences... For example, in this case the formal parameter is still a local variable!
    - Think about “`n = n + 1`” in the example below

```
int f(int *n)
{
  *n = *n + 1;

  return *n * 2;
}
```

Passing Parameters by Name

- Seen for functional programs evaluation
  - Function name replaced by function body
  - Formal parameter replaced by actual parameter
- Not very useful for imperative languages...
  - Parameters can be evaluated every time they are used... Think about “`x + x`” with actual parameter “`i++`”!
- ...But good model for how FP reduction works!

Issues with Parameters by Name

```
int a = 1;

int f(int v)
{
  int a = 666;

  return a + v;
}
```

- What is `f(a)` if the parameter is passed by name?
- `{ int a = 666; return a + a;}`... Returns `1332`!
- If the name of the local variable is changed to “`b`”, we get `{ int b = 666; return b + a;}` and the return value is `667`!
- The return value depends on the name of a local variable???

Call by Name, Again

- consider this code:

```
int infinite_recursion(int z)
{
  return infinite_recursion(z);
}

int select(int n, int x, int y)
{
  return n == 0 ? x : y;
}
```

- What happens in C++ (parameters passed by value) when calling `select(0, 1, infinite_recursion(1))`?
- What would happen if parameters were passed by name?
  - Can you emulate pass-by-name, in this case?