On the representation of functions in different programming languages

Sven Moritz Hallberg
pesco @ khjk.org

2013 – 09 – 03

Abstract

The specific realization of functions and subroutines as provided by six popular programming languages are surveyed from the standpoint of language design. The languages discussed are C, Scheme, (Common) Lisp, Smalltalk, Python, and Scala. Questions of interest include the sharing of namespaces between functions and other objects, closures and scoping rules, support for higher-order functions, lambda expressions, etc. An additional section notes interesting points about some languages that are not included in the main presentation.

1 Introduction

The concept of the *function* is omnipresent in programming languages. The specific realizations of the abstract concept, however, differ. The purpose of this article is to explore the facilities provided by some popular programming languages that the author holds to be representative of a wide range of designs.

Questions of interest will concern the sharing of namespaces between functions and other objects, closures and scoping rules, support of higher-order functions, lambda expressions, etc. Note that these are aspects of the *use* and creation of functions. Deliberately left out of the scope of this article are variations on their form such as specific support for multiple return values, keyword arguments, optional arguments, and arguments of variable number.

Even more primitive than the concept of the function, though obviously related, is that of the variable. Roughly speaking, the things that can be assigned to variables and passed as function arguments are referred to as first-class objects or "first-class citizens" of a language. When functions fall into this category, the language is commonly said to "have first-class functions".

Since whether an object is first-class hinges upon its being able to be used as the value of a variable, comparing the form of function and variable definitions in a given language will be interesting.

2 On metalanguage

When discussing languages, it is in order to recapitulate the framework of *metalanguage* to be employed.

Most importantly, the things any language allows discussing are called its *objects*. The objects of a metalanguage are other languages. Consequently, the languages under study are referred to as *object languages*. Our object languages, obviously, will be programming languages. The objects of programming languages, in turn, are functions, values, classes, etc.

The reader should be sure to remain conscious of the level, metalanguage or concrete (object) language, which is the context of any statement. Particularly, the term "object" as it pertains to (meta-) language is not to be confused with the concept of the same name from an object-oriented object language (see?).

For example, a function in C++ does not itself constitute a first-class object. Nevertheless, it is an object of the language insofar as the language allows "talking" about it, in the form of describing its definition and referencing it in the definition of other functions.

3 C (1972)

```
unsigned int hash(const char *buf, size_t len)
{
    unsigned int v = 5381;
    while(len--) {
        v = v*33 + *buf++;
    }
    return v;
}
```

C[6] is a *procedural* programming language. That is, its central unit of abstraction is the procedure, a (named) sequence of instructions. Procedures may accept some parameters and return a result. In fact, C procedures are termed *functions* and are obviously meant to represent (computed) value mappings as well as instruction sequences.

The set of values in C includes *function pointers*: indirect references to a procedure that can be called just like a direct reference to that function's name.

Function pointer types are odd-looking but their syntax is consistent with the rules followed by other types. It is common to hide the unwieldy type behind an alias.

```
typedef unsigned int (*HashFunPtr)(const char *, size_t);
```

As first-class values, function pointers can be passed as arguments, assigned to variables, and returned as results. The following example shows a function pointer variable being declared, assigned, and called:

```
HashFunPtr hp;
```

```
hp = &hash;
unsigned int x = (*hp)("text", 4);
```

Functions have access to the environment, that is variables and other functions, available at the place of their definition – functions in C have *lexical scope*. Unfortunately, they can only be created by definition at the top level of the program. There is no support for local definitions, lambda expressions, or partial application. However, these shortcomings can be worked around to a greater extent than is often realized:

- *Lambda lifting* is the standard technique by which the required context of a lambda expression is moved into the arguments of a top-level combinator. Since there is no alternative, C programmers will do this instinctively when defining "helper functions".
- Partial application can be emulated to a degree by combining raw function pointers with dynamic "environment" objects. There is no satisfactory general solution, so instances of this workaround will be tailored to specific situations. A generic example could look like this:

  ```
  struct dfun { // function with dynamic environment
    void (*f)(void *env, ...);
    void *env;
  };

  void f_foo(void *env, ...);
  struct dfun foo = {.f = &f_foo, .env = NULL};
  ```

Though restrictive, these techniques can go a surprisingly long way to support "quasi-functional" combinator libraries [9].

4 Scheme (1974)

```
(define (hash str)
  (let ((len (string-length str))
        (elt (lambda (i) (char->integer (string-ref str i)))))
    (do ((i 0 (+ i 1))
         (v 5381 (+ (* v 33) (elt i))))
        ((= i len) v))))
```

Scheme is a distinct Lisp dialect designed from the ground up to be lean and to support a number of "modern" features not present in early Lisps. Predating the venerable Common Lisp by a decade, the requirement of, e.g., first-class continuations and tail-call elimination by the Scheme standard was notable. As such, Scheme had a strong influence on later Lisp dialects, not least Common Lisp itself.

Function calls in Scheme take the familiar form of *S-Expressions*:

```
(hash "Hello!")     ; call 'hash' on a string
```

The only exception to this format are so-called *special forms*, indicated by certain keywords in the place of the function name. The important examples are `define` which binds a top-level variable and `lambda` which returns a function value.

```
(define tthree 23)  ; bind symbol 'tthree to value 23
(lambda (x) ...)    ; create a function of one argument
```

Functions are first-class values and the variant of `define` shown in the opening definition of the `hash` function binds the variable `hash` to a function value. Using `define` in this way is indeed equivalent[10] to binding `hash` to the result of a lambda expression.

```
(define (hash str) ...)     ; just a shortcut for...
(define hash (lambda (str) ...))
```

Matching the fact that function names are simply variables bound to function values, the first element in a function call S-expression may generally be any other expression that evaluates to a function; for instance:

```
((lambda (x) (x x)) (lambda (x) (x x)))
```

Functions in Scheme form *lexical closures*, i.e. they always execute in the environment that was in effect, syntactically, at their place of creation. This allows, for instance, a partial application operator — in this case only for two-argument functions — to be defined:

```
(define (papp f x) (lambda (y) (f x y)))
```

The outer argument `x` is "enclosed", or *captured*, by and remains available to the returned function until needed.

5 LISP (1958 — Common Lisp 1984)

```
(defun hash (str)
  (reduce #'(lambda (v x) (+ (* v 33) (char-int x))) str
          :initial-value 5381))
```

The great mother of functional programming languages, LISP[1] turned into a practical general purpose language almost by accident when Steve Russell implemented its first metacircular[2] interpreter in 1960 [7]. The language subsequently saw busy development into a number of dialects that eventually congealed in the specification of Common Lisp[11].

Lisp is obviously similar to its descendant Scheme in many respects, but the treatment of functions specifically is one area of significant difference. For example, early Lisps did not support lexical closures. While that feature, pioneered by Scheme, has since become standard in Common Lisp, another even more obvious distinction remains.

[1] now usually spelled Lisp
[2] i.e. written in Lisp itself, cf. [1]

Functions in Common Lisp are conceptually first-class values that can be passed around, etc., but by default they occupy a namespace separate from variables.

```
(defun  foo () 'fun)            ; define function
(defvar foo (lambda () 'var))   ; define variable
```

Executing the above lines does not present a name clash. It associates the *function name* `foo` with one function and the *variable name* `foo` with another.

The split of namespaces makes it necessary to distinguish which one is to be referenced by symbols in any given context. For instance, the first place in a function call form naturally accesses the function namespace while symbols in other places usually refer to variables.

```
(foo)                       ; => 'fun
(funcall foo)               ; => 'var
```

The `funcall` function is used to call a function when it is given by a variable or other expression. Conversely, to dereference a function name like a variable, Lisp provides the `function` special form. The character sequence `#'` is a syntactic abbreviation for it.

```
(funcall (function foo))    ; => 'fun
(funcall #'foo)             ; => 'fun
```

At the same time, `function` provides the interpretation of `lambda` expressions as shown in the opening example. On top of that, the "bare" form used as `(lambda () 'var)` earlier is implemented by a syntactic *macro*. Thus the two forms are ususally interchangeable. Yet, the following is an error; recall that function lookup occurs not via evaluation:

```
(#'(lambda (x) x) 'ok)      ; "invalid function"!
```

Perhaps surprisingly, the bare form works. Function lookup explicitly recognizes and interprets it in the same way the `function` form does.

```
((lambda (x) x) 'ok)        ; => 'ok
```

In summary, it should be clear why Scheme opted to avoid this confusing state of affairs.

6 Smalltalk (1980)

```
hash
    | v |
    v := 5381.
    1 to: (self size) do:
        [ :i | v := v*33 + ((self at: i) asciiValue) ].
    ^v
```

What Lisp is to functional languages, Smalltalk[4] is to object-oriented programming. Its inventor Alan Kay coined the notion when he developed Smalltalk. At the heart of the language, however, as much as "everything is an object", lies the notion of method invocation as *message passing*.

To pass the message `even` to the object `1`, for instance, it is simply written after it as `1 even`. Every message receives an answer, so expressions like that yield a functional result, in this case `false`. A method argument is indicated in the message by a colon. Binary operators are a syntactic special case that elides the colon.

```
'hello' at: 2               " => $e "
3 + 2                       " => 5  "
```

More than one argument can be passed in a uniform way by attaching additional *keywords* to the message.

```
'ask' swap: 2 with: 3       " => 'aks' "
```

Thus messages and methods form the primary representation of functions in Smalltalk.

While neither methods (nor messages) are a priori cast as first-class objects, Smalltalk provides and makes extensive use of a first-class representation for blocks of code. For example, the familiar *if* construct is provided as a method of the `Boolean` class:

```
(x > 0) ifTrue: [x "ok"] ifFalse: [0 "clamp"]
```

The bracket expressions can encapsulate any sequence of instructions and yield code block (`BlockClosure`) objects A `BlockClosure` is called by sending the `value` message to it.

As seen in the opening example, code blocks can take arguments; they also form lexical closures. Thus they effectively model lambda expressions, albeit in a fashion that is slightly inconvenient to call compared to regular functions (methods).

7 Python (1991)

```
def hash(str):
    v = 5381
    for x in str:
        v = v*33 + ord(x)
    return v
```

Python is designed to be very much a prototypical[3] object-oriented language. All first-class objects are represented as instances of some class. Functions may be defined in any scope. Functions as well as methods in Python are themselves objects. In general, a method call

```
obj.foo(x)
```

occurs in two conceptually distinct steps[2]:

1. *Lookup* of the method `obj.foo` akin to an attribute. This yields an object of class `MethodType`. This object, called a *bound method*, wraps the function implementing `foo` and the instance `obj`.

[3]in the ordinary sense of the word; not to be confused with *prototype-based*

2. *Invocation* of the bound method's special `__call__` method. This will in turn call the definition of `foo` on `obj` (as its `self` parameter).

Free functions as well as *unbound* method definitions are represented as objects of class `FunctionType`. An unbound function call such as `hash('hi')` results, as above, in the invocation of the function object's special method `__call__`.[4]

Unbound methods are accessible as attributes of their class. Being ordinary functions, they can even be called directly, without connection to an instance. Consider:

```
class C:
    def foo(self, x):
        return self

c = C()                         # instantiate
c.foo(x)                        # => c
C.foo(z,x)                      # => z
```

It is worth noting that bound method objects save the method definition as it was at the time of their creation. Later modification to the class do not affect them.

```
f = c.foo                       # get bound method
C.foo = None                    # modify class
f(x)                            # => c
```

This fits with the fact that Python's functions form lexical closures; extracting a bound method is equivalent to the partial application of its `self` argument.[5]

Even though idiomatic Python focuses on imperative and object-oriented designs, functional programming styles are supported by built-in higher-order functions and the availability of lambda expressions.

```
map(lambda x: x+2, [1,3,5])     # => [3,5,7]
```

The utility of lambda expressions is somewhat constrained by the fact that their body must consist of a single expression; no compound statements are allowed. This restriction can be worked around with local function definitions.

8 Scala (2003)

```
def hash(str : String) : Int = {
    str.foldLeft(5381)((v,x) => v*33 + x.toInt)
}
```

Scala[8] is an object-oriented programming language that combines Java with a number of features from strongly-typed functional languages like Haskell. Its syntax, while still reminiscent of Java, has undergone significant changes aimed at making it more concise and convenient.

[4] Calling that method itself is possible and short-circuits to its own invocation.

[5] Actually, proper lexical scoping was not introduced until Python 2.2. [5]

The stain on Java's "purely" object-oriented design is gone in Scala: every value is an object. The primitive types of the JVM are automatically and transparently dressed in Scala classes. Accordingly, functions to be passed around are represented by objects as well. Unlike Python, however, Scala does not apply this representation to methods. Methods and the mechanism for calling them are at first outside the realm of values. Syntax is then provided to construct function objects from arbitrary code. In this respect, Scala is similar to Smalltalk.

Lambda expressions (not including an actual lambda) are formed with a double arrow:

```
() => println("Hello!")          // 0-argument procedure
(x:Int) => x+1                   // unary function
(x:Int, y:Int) => x+y            // multiple arguments
```

The type annotations on the arguments may or may not be required, depending on the system's ability to infer them in a given context. A useful shorthand is available when only few arguments are used and their types are known. Using an underscore as a placeholder for subexpressions forms a "headless" lambda expression that accepts the required arguments in the order in which they appear.

```
(_ + 1)                          // increment argument by 1
(_ + _)                          // addition function
```

Variables, as mutable or immutable references, are declared by the `var` and `val` keywords, respectively.

```
var step = 1
val inc  = (x:Int) => x+step     // lexical closure
```

Note that the closure captures the variable *reference*, not its value at the time; changing `step` in the above example will modify the behaviour of `inc` accordingly.

To make function objects as convenient as methods, the function call syntax is overloaded. When used on a function object, it is translated to a special method call `apply` on that object. This is similar to the situation with `__call__` in Python.

```
inc.apply(1)                     // => 2
inc(1)                           // equivalent to the above
```

Since the self-reference `this` and all instance variables are in scope during object construction, it is interesting to note that it would be feasable to represent an object's methods as variables of function type alone:

```
class Counter {
    var value = 0                // attribute
    val count = () => value+=1   // method
}
```

Attribute initializations as seen above are placed automatically into the class's constructor. Unfortunately, the "methods as variables" style results in an extra indirection of every call and bloats objects with the extra attributes. The `def` keyword as seen in the opening example defines "real" methods with the added

benefit of allowing a one-to-one interface with classes written in Java. Local definitions via `def` are also allowed inside any code block.

Support is provided to make regular methods usable as function objects. Firstly, a method name can be turned into a function object by use of an underscore in the place of the argument list.

```
val foo = obj.foo _
```

This underscore may be omitted where the type system knows that a function object is required. One must be aware that this shortcut is part of some syntactic overlap. Combined with other features, the possible meanings of a simple symbol include:

- reference to a local variable
- reference to an attribute
- reference to a method to be wrapped in a function object
- call to a method where an empty argument list may be omitted
- call to an "accessor" method that has no argument list

In summary, a remarkable amount of clever syntactic and semantic effort is included in Scala's design to make the transition between methods and function objects appear as seamless as possible. It is on the programmer, however, to know where the shortcuts are applicable and where they are not.

9 Notable Absences

Given the proliferation of programming languages, many have to be left out to limit the length of this presentation. The omissions do include some interesting cases to be mentioned — at least in passing — here:

FORTRAN (1957) One of the oldest programming languages, FORTRAN is remarkable in that it explicitly distinguishes between `FUNCTION`s and `SUBROUTINE`s. Only the former evaluate to a result for use in arithmetic expressions while the latter support alternative exits/continuations. This distinction between functions in the mathematical sense and structural subprograms fits with the fact that the language — named for "formula translating" — was developed with a focus on numerical calculation.

C++ (1983) Functions in C++ are much like their counterparts in C, gaining only the generic programming capabilities offered by the template system. In addition, though, classes and templates allow the implementation of function objects. These are much comparable to those of Scala, albeit having to obey much more cumbersome syntax. C++11 introduces lambda expressions.

Haskell (1990) As a functional programming language through and through, Haskell matches Scheme in making the definition of a function exactly equivalent to the assignment of a lambda expression to a variable. Its very

concise syntax is in part supported by "Curry-style by default" where partial and full function application are syntactically the same.

Java (1995) With the introduction of generics, it would be possible to officially include function objects in Java, but no standard has emerged. Otherwise, methods work essentially as they do in C++.

Ruby (1995) While drawing inspiration from a number of languages, Ruby models its object system closely after Smalltalk. Code block objects carry over in a slightly modified fashion, where they always form a special kind of function argument. Lambda "expressions" are constructed as a regular function in this framework.

JavaScript (1995) Under a veneer made to look like "classical" object-oriented languages, JavaScript is actually prototype-based with a surprisingly clean functional language at its core. Function definitions are for the most part equivalent to lambda expressions assigned to variables, functions form lexical closures, and methods are implemented as function objects in attribute slots.

10 Conclusion

While the basic concept is universal, the details of how programming languages realize functions show significant differences. The design space appears not very large but with interesting subtleties.

It seems noteworthy that there is no clear progression from immature to optimal designs over time. Very early, Scheme offers arguably the cleanest, most unobstructed representation, surpassing and subsequently re-inspiring its own origin. Other, more prevalent languages have found different variations necessary to accommodate their chosen design goals.

Future programming language design will surely keep drawing inspiration from a pool of ideas, not to mention in areas outside the representation of functions.

References

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., USA, 2 edition, 1996.

[2] D.M. Beazley. *Python essential reference*. New Riders professional library. New Riders, 2000.

[3] A. Goldberg and D. Robson. *Smalltalk-80: the language*. Addison-Wesley series in computer science. Addison-Wesley, 1989.

[4] J. Hylton. Statically nested scopes. PEP 227, `http://www.python.org/dev/peps/pep-0227/`.

[5] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Pearson Education, 2005.

[6] J. McCarthy. History of lisp. *History of Programming Languages*, 1981.

[7] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.

[8] M.L. Patterson, D. Hirsch, et al. The hammer parsing combinator library. `http://github.com/UpstandingHackers/hammer`.

[9] M. Sperber, R.K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19:1–301, 8 2009.

[10] G.L. Steele. *Common Lisp: The Language*. HP Technologies Series. Digital Press, 1990.