NOVEMBER 2021 ISSUE 01

HATS & CLOAKS

THE GUIDEBOOK

Fashion never grows old

&

A Quick Look At Function Parameters

Charles Lobo*

charles.lobo@gmail.com

October 20, 2021

Abstract

A short note on the use of function/method parameters in modern programming, their ideal position and number along with techniques for designing good parameters.

I. Introduction

The problem of function parameters has been around for a long time - as soon as we moved beyond programming with subroutines and into "procedures" and "functions", we have had to pass parameters to them. However, the idea of how many are needed and in what order - while trivial in a sense - has never been clearly laid out.

The reason for this is we need to draw from multiple fields in order to get this clarity. Otherwise, as we will see, it is easy to look at existing trends, misunderstand them and derive wrong conclusions.

This short note attempts to definitively answer some of the questions around function parameters - starting with what they are, why they are needed, how many are ideal for good programming, and what order they should be in.

II. Definitions

We start by defining the model of function and parameters and introducing the concept of a *global hidden parameter* (γ) as well as side effect returns (v).

i. Functions & Parameters

A simple way to think of a function is a process or relation that associates each element of an input set P, the domain of the function, to a single element r, the codomain (or range) of a function.

$$r = f(p) \tag{1}$$

It has been said that functions are "the central objects of investigation" and they have a variety of expressions. But for our purposes this is a good model of a function.

In the expression $r = f(p)$, the parameter is p and the function *returns* the value r. We can now build on this to come to conclusions on function parameters.

ii. Arity

In this case of $r = f(p)$ the function takes only one parameter. The number of parameters is known as the *arity* of the function and this function would be have *unary* arity.

A two-parameter function $r = f(p_1, p_2)$ has a *binary* arity, while a three parameter function $r = f(p_1, p_2, p_3)$ has a *ternary* arity. Table 1 lists examples of functions with their arity.

iii. Global Inputs

In programming, functions can be affected by state outside their definition. While this state could be encapsulated in a higher module (object or function), we can treat it as "global" without loss of fidelity.

For the purpose of this model we can simply treat global variables as another, hidden,

*

Table 1: *Function Arity*

x-ary	Arity	Sample	Example
0-ary	*Nullary*	$r = f()$	'True', 'False'
1-ary	*Unary*	$r = f(p)$	Logical 'NOT'
2-ary	*Binary*	$r = f(p_1, p_2)$	'OR', 'XOR', 'AND'
3-ary	*Ternary*	$r = f(p_1, p_2, p_3)$	Conditional '?:'
4-ary	*Quaternary*	$r = f(p_1, p_2, p_3, p_4)$	
n-ary	*n-ary*	$r = f(p_1, p_2, \ldots, p_n)$	

parameter (γ) passed into the function.

$$r = f(\gamma, p) \tag{2}$$

In this model, global inputs simply change the *arity* of the function (*arity* is discussed in the next sections). In some programming languages a first argument of this sort is sometimes reified and called 'this' or 'self'.

iv. Side Effects Returns

In programming, besides returning values, functions can also "do things". These are often called "side-effects".

As with the case of the hidden input (γ) discussed above, we can model an expanded output set to hold all the side effects of the function:

$$\mathrm{Y} = f(p) \tag{3}$$

where $f(p)$ has side effects.

For the purpose of this discussion we can now see that functions that are "pure" (no side effects) and return r or function that have side effects and return Y both can be treated the same in our analysis of the their input parameters.

III. Significant Concepts

In this section we look at a some situations commonly encountered in programming and how the model works in those contexts.

i. Variadic Arguments

Given a definition of a function as a mapping from input to output, *variadic* functions are simply syntactic sugar over an 'n-ary' function with multiple arguments being set to nullified/zero points ($\varnothing$)

$$r = f(p_1, p_2, \varnothing, \varnothing, \varnothing, ...) \tag{4}$$

ii. Named Arguments

While it is tempting to think of named arguments as another layer of syntactic sugar we will see as, we discuss "Parameter Objects" later, that the addition of the naming makes this technique far more significant.

iii. Objects

If we have followed the arguments above, we can now make our first interesting observation - an Object in this model - is simply an 'n-ary' function with n named slots.

$$O_n(m) = v \tag{5}$$

where v is the value at the slot named m.

The methods of the object all take the hidden parameter γ , that references $O_n(m)$ itself. This explains the common naming convention of 'self' and 'this' for the hidden parameter.

$$\gamma \equiv O_n(m) \tag{6}$$

iv. Callbacks

Callbacks are an interesting case as well. While on the surface they look like parameters (p), a bit of thought will show that in Equation (1) $r = f(p)$ callbacks are actually *return values* (r).

In our model therefore, callbacks are **not** counted in the *arity* of a function.

IV. Discussion

We are going to solve two problems related to function parameters. They are:

1. What is the ideal arity (number of parameters)?
2. How should the parameters be ordered?

Let's look at each in turn.

i. The ideal number of parameters of a function

Let's start by asking a simple question - what is the minimum number of parameters of a useful function (the lower bound)?

i.1 The Lower Bound

A *nullary* (0-ary) is called a `Constant` in programming and is therefore not a useful "function". Therefore the first useful function we have in programming is the unary function (1-ary). Remember this includes the hidden variable γ [2]:

$$r = f(p) \tag{7}$$

So the minimum number of arguments of an ideal function n_{args} is:

$$1 \leq n_{args} \tag{8}$$

i.2 The Upper Bound

Given this lower bound, what is the upper bound for a useful function? Is there even any upper bound? Can't we just have functions with dozens of parameters ($f(p_1, p_2, p_3, p_4, p_5, p_6, p_7, \dots)$)?

I argue that there is indeed an upper bound for the arity of programming functions. Most programmers instinctively have a feel for this bound but would be hard pressed to articulate why (hence the reason for this note).

Defining a function with any number of parameters is a simple enough matter and presents no problem to the programmer. The issue comes when *calling* that function.

When *calling* a function the programmer has to remember not just the parameters, but their *position* in the parameter list. This is a tall order for every function the programmer must call. It is well known that people remember 7 ± 2 items most easily. Given this limitation, and the fact that programmers are not expecting to memorise how a function should be called, it is ideal to keep the number of arguments to this limit.

So now we have an upper bound:

$$1 \leq n_{args} \leq 7 \pm 2 \tag{9}$$

This is a more reasonable upper bound but we can take it a step further. The function arguments we have been discussing thus far are mathematically derived but there is a corresponding *linguistic* concept of "*argument*".

Definition: *In linguistics, an **argument** is an expression that helps complete the meaning of a predicate (a main verb and its auxiliaries).*

This helps us a lot because linguistics provides us with both empirical and historically *evolved* information on the number of argument humans feel comfortable dealing with naturally. This number, in linguistics, is called "*valency*" (corresponding to the *arity* we have been discussing so far).

0. An *impersonal* valency has no subject. e.g. *It rains*
1. An *intransitive* valency has one argument. e.g. *He sleeps*
2. A *transitive* valency has two. e.g. *He kicks the ball*
3. A *ditransitive* valency has three. e.g. *He gave her a flower*

There are few sentences that take four arguments (4-valency). The most well known of these in English is *bet*.

4. *Tritransitive* : e.g. *I bet him a dollar on a horse*

More than this and the sentence becomes harder to understand. From this we can infer that:

- Most functions should take a up to 3 parameters
- If pushed, the maximum a function should take is 4 parameters

Leading nicely to the bounds for the number of parameters:

$$1 \leq n_{args} \leq 4 \tag{10}$$

ii. How to reduce the number of parameters

Given $1 \leq n_{args} \leq 4$, we are now faced with a problem. Refactor all we like, some functions may simply *need* a certain number of parameters. We cannot dictate the requirements so we need design tools to bring the number of parameters to within our limit.

In such a case, there are well-known solutions that work:

1. Given the hidden parameter γ , we can hold many of the parameters in the scope captured by γ. This is what Objects Oriented Programming does: by holding parameters as members variables we can design the objects to have "tiny" methods that only require one or two parameters (in addition to the hidden parameter).

$$f(p_1, p_2, p_3, p_4, ..., p_n) \rightarrow f'(\gamma, p) \tag{11}$$

2. The hidden parameter γ also holds values when using higher order functions and closures which perform the same transformation.

$$f(p_1, p_2, p_3, p_4, ..., p_n) \rightarrow f'(\gamma, p) \tag{12}$$

3. Finally, we can use **Parameter Objects** - objects or structures created specifically to wrap a large number of parameters and pass that in as a single parameter.

$$f(p_1, p_2, p_3, p_4, ..., p_n) \rightarrow f'(o, p) \tag{13}$$

where $o = O_{n-1}(m)$

While these may looks like "tricks" to reduce the number of parameters we have actually introduced three significant changes:

1. We have introduced *names* for the parameters. This seems like a trivial change but it is actually very significant. As soon as we have names, the cognitive overload of remembering the position of parameters goes away.

2. Unless we create anonymous objects, we have introduced a new *type* that holds a group of parameters. This also aids greatly in comprehension of the function itself (which now operates on the new type abstraction).

3. Because the object ($O_n(m)$ see Equation 5) being used is effectively a function, it introduces *another level of redirection*. This layer of abstraction allows our parameters to be more resilient to future changes; for example: new parameters can be introduced without needing to change all older call references.

iii. How function parameters should be ordered

Finally, we turn our attention to how the parameters of a function should be ordered. We start with the ordering of two parameters and then extend our results.

$$y = f(a, b)$$

vs

$$y = f(b, a)$$

First we need to distinguish a from b in some way. To do this, we define a measure $\Phi(a)$ as the *likelihood that a will change* and a partial order relation:

$$a \leq b, iff$$

$$\Phi(a) \leq \Phi(b) \tag{14}$$

That is $a \leq b$ if the likelihood of a changing is lower than the likelihood of b changing (a is more *stable*).


THE
BASICS

iii.1 Currying

Currying is the technique of converting a function that takes multiple arguments into a sequence of functions that each take a single argument.

$$f(a,b,c) = h(a)(b)(c) \tag{15}$$

Currying is useful both in theoretical models and in practice as some programming languages support curried functions. We can use this theoretical model to decide on the correct ordering of function parameters by arguing as follows.

Given $f(a,b)$, we can $curry(f) = h$ and memoize the value of $h(a) = X1$. Now if $\phi(b) \leq \phi(a)$ we often have to discard the value of $X1$ *even if the value of b has not changed* (because a is less stable than b). However if $\phi(a) \leq \phi(b)$ we are less likely to discard the value of $X1$ when b changes because a is more stable than b.

With this argument, we can formulate a **Principle of Decreasing Stability**:

Definition: *Parameters should be ordered from left to right in order of increasing likelyhood of change ($\phi(x)$).*

That is, the most stable parameter should be first, followed by the next most likely to change, and so on.

$$\overrightarrow{f(p_1, p_2, p_3, p_4, ...)} where p_n \leq p_{n+1} \tag{16}$$

iii.2 Example: Updating a value

As a common example, for a function that "updates" one of it's parameters, we *know* with certainty that the parameter most likely to change is the one being updated. By the *Principle of Decreasing Stability*, this should then be the *last* parameter. For example:

```
    setcolor(red, circle) is correct,
while
    setcolor(circle, red) is not
```

V. Results

From the above short discussion we conclude that

Programming functions should have between 1 *and* 4 *parameters, (with additional callbacks), ordered from most stable to least. Parameters can be grouped into types/objects to reduce the arity if needed.*

This advice aligns nicely with most best-practices followed in the industry today. It now has a stronger theoretical foundation to stand upon.

References

[Encyclopedia Britannica] Function definition. *https://www.britannica.com/science/function-mathematics*.

[Graham Hutton] comp.lang.functional FAQ. *http://www.cs.nott.ac.uk/ pszgmh/faq.htmlcurrying*

[Schechter, Eric (1997)] Handbook of Analysis and Its Foundations. Academic Press. p. 356. ISBN 978-0-12-622760-4.

[Allerton, D. J.(1982)] Valency and the English verb. London: Academic Press

[Herbst, Thomas] English Valency Structures - A first sketch *http://webdoc.gwdg.de/edoc/ia/eese/artic99/herbst/main1.html*

ISSUE 01

HATS & CLOAKS
THE GUIDEBOOK

NOVEMBER 2021

Fashion never grows old

&