|  |
| --- |
| Circle Language Spec: Commands |

## Comparison to CPU-Like Call Instantiation

There was a lot of brainstorming about how the new computer language’s call instantiation compares to call instatiation in other programming languages, which are totally compliant to the CPU’s way of handling command calls, but is not in the spirit of *commands as an object*. But that brainstorming was postponed, because it did not have anything to do with a goal of a workable type of call instatiation, but more with optimization of the new computer language. The contemplations are still there in this article, as an unfinished brainstorm, that may be later worked out, to better take advantage of CPU power.

Perhaps creation behavior is just the most important issue of Command is an Object. It really belongs to the introduction of commands, that explains how a command is an executable object. Closely related to it, is the special creation behavior of executable objects. You have to well understand the creation behavior to implement a command as an object.

There was a brainstorm about the problems, that come with creating a sub-call object just before it executes. This brainstorm in further down below in the section *Former Text*.

But the idea will now be further worked out by making a comparison between other programming languages and how it will be solved in the new programming language.

### What happens in other programming languages?

In other programming languages, when a call is executed, calls to sub-commands are not copied to the call instance.

In other programming languages, it is always the command definition itself that is running. But because each time it is running it is operating on different data, you can consider it a separate command object every times it runs.

From a single-threaded point of view, the same command does not run at the same time, so if a command runs multiple times, it is always first running one one piece of data and only after that running on another piece of data.

In a multi-threaded situation a command could run twice at the same time.

A thread, however, has its own stack and also its own instruction pointer, as well as its own registers.

In a time-sliced threading situation, whenever the CPU switches from one thread to another, the data the command operates on is changed and the command will pick it up where it left off in that thread’s previous time slice.

In multi-threaded situations with two CPU’s or two CPU cores things still go ok, because each thread has its own instruction pointer and a different call stack. It is no problem to reuse the same code. The code does not specify which specific data it is operating on, so the code is just instruction data, that is reused.

The only thing a call-instance consists of in other programming languages is either different data on the call stack, or a different call stack altoghether (different thread).

The parameters and local variables *do* need to be instantiated on the stack in other programming languages. But sub-command calls, parameter assignments, and all that, do not need to be copied to the call-instance, because it is always just the definition that is running.

What the CPU does, is get new instructions from the spot pointed out by the instruction pointer. The instruction may refer to data with a pointer address. This pointer address is always fixed in the code, but it is related to a base address. This base address is stored as the *data segment* in the DS register of the CPU, to make hard-coded pointer addresses work by always relating it to this base address. A thread has an instruction pointer location, a stack, and its own CPU register values. Therefore each thread at one fiven time operates on its own piece of data. There are CPU registers that store an instruction pointer (IP CPU-register), a memory base address (DS CPU-register), a code base address (CS CPU-register) and a stack base address (SS CPU-register).

The same code, with fixed pointers has to be executed on different data all the time, so that must mean a data switch. Each object has its own data segment. To operate on that data, the DS register needs to be set. That means, that on each call to an object’s method, the original DS register is put onto the stack to remember it, and the DS register is replaced to point to the object from which to call a method. Inside the method of the object called, other objects may be created. This changes nothing about the DS register. The object pointers are put onto the stack, instructions can operate on a fixed spot in the stack, instead of a fixed spot in the data segment. But inside the method of the object called, objects are created, but also objects are called. As soon as another object’s method is called, the original value in the DS register is put on the stack, and the DS register is set to point to the object to call. After the object call is done, the DS register is restored to its original value, by popping the original value off the stack again. That’s probably the way it works in object oriented systems.

### Why does that not work for my own system?

The CPU is dumb. It executes and executes doing exactly what it is lead to do by its stupid instruction pointer that only goes foreward, and it only skips to another spot, when it encounters a jump instruction and the data then determines whether it will jump at all. A CPU is really dumb and just does what we tell it to do. As a normal program runs, you do not *see* debug information. When a program runs normally, you do not see what method it is in and of what object, and even the CPU does not see what method it is in and of what object. When you debug, though, you want to see the exact code line of what exact method and of what object, you can even see (from an alternated call stack) what commands were called before and of what object.

In my own computer language, when you pause on a running command, you see the procedure call stack as the hierarchy upward. This call stack was not built up as the command was running. You can zoom in on any piece ‘call stack’, that is not even running, but might be running in the future.

Now, the data, assigned to a call’s parameters… where does it come from in CPU-like, procedural and object oriented programming languages? Where does it come from in my own system?

There is:

- data

- pointers to this data

- sub-commands

- parameters of these sub-commands

- and the assignment of data to the sub-commands’ parameters

funny enough, that is all a command call consists of. Do remember that machine instructions, arithmetic operators, comparative and boolean algebra and execution control statements such as If and For, are all just command calls.

### Data

In a CPU-like programming language, the data is already there as created objects somewhere in memory. That’s the same in my own language.

### Pointers to this data

Pointers to the data, that was already there, are on the call stack in a CPU-like language. In my own language the pointers to this other data are inside the call object. So from that perspective, the call object constitutes, what is otherwise a chunk of the stack, that belongs to a particular procedure call.

### Sub-commands

Now, the sub-commands in a CPU-like language are there as CALL instructions. CALL instructions are in the command definition. In my own language sub-commands are there as sub-command references, with a *definition* assigned to them, but no object yet. The sub-command *object* is only created just before a sub-command will be run.

I still do not have a clear view on why in CPU-like languages it is CALL instructions in the definition, and in my own language it is sub-command references inside a call object.

<…>

< It’s just that, in CPU-like languages the instructions run and the data to operate are such separated things. The registers and the stack hold the data, and the instructions hold the code. In my own language, the data to operate on is not on *the stack*, but *inside* the procedure to execute. So the procedure call (instruction) *contains* the data to operate on. No. I still don’t get it>

### Parameters of these sub-commands

In CPU-like languages this means, that space is created on the stack for these parameters just before the sub-command is executed. Actually this happens in one blow with assigning data to the parameters.

In my own language, the pointer parameters are created as pointers, the value parameters are created as pointers and simple created objects (data). This all done, just before the sub-command is executed.

### Assignment of data to the sub-commands’ parameters

In CPU-like languages this means, that parameter space on the stack is assigned values. Actually, this happens in one blow ith creating the space on the stack. This all just before the sub-command executes. In my own language, this means, that the pointer parameters are set to point to objects and the value parameters, that are created simple objects, are now assigned values. This also happens, just before the sub-command executes.

< There is a difference between when the assignment is executed, and where the assignment call is. The assignment call in CPU-like languages are PUSH instructions in the command definition. Only when executing the parameter assignment / push instruction, the parameters are actually created. The assignment call in my own language is a special assignment command call, displayed as a parameter passing. The assignment command call must already link to parameter references of sub-commands, therefore the sub-commands must already be created, but not yet initialized. Only when the sub-command is about to be run, the parameter assignment is actually executed. >

### Reason to always create and assign sub-commands…

For visibility...

When navigating through the call symbols in a diagram, you want to see call objects as created objects anyway, but this is just visualization of what *might* happen. Huh?

When you debug calls, then you want to see all the sub-command objects created, even though nothing needs to be assigned to the parameters. The parameter passings are visible anyway, because those are actually assignment commands *displayed as* parameter passing.

You have:

- data

- pointers to this data

- sub-commands

- parameters of these sub-commands

- and the assignment of data to the sub-commands’ parameters

The sub-commands have to be created and their parameter sub-objects need to be in there, but can be Nothing. The parameter assignments are (really) special sub-commands, that are displayed with a parameter passing notation, but that does not require the parameters to be created.

### How can a parameter passing assignment be stored

A parameter passing asignment could only be stored by the parent command-call if the sub-commands are created (but not initialized; parameter references are there, but not filled in yet).

### More Brainstorming

< A problem is, that in my own computer language, the command definition is not assembly

code. It is objects of the new computer language. >

In CPU-like languages the sub-command-calls are in the machine instructions.

In my own computer language sub-commands-calls are in an objects somewhere.

The command definition object has the sub-commands, but they are not lined up

like machine instructions. Each command call is tied to the next with a pointer

to another command object.

It is very hard to make the command definition use the data in the command CALL,

because that is what has to be done, to do it in the more CPU-like fashion.

In CPU-like fashion you have a split up between the data context, and the code context.

In my own language you really see both data and code together.

To let a definition run multiple times at the same time, with a different data context,

just looks wrong in a diagram.

But can you let a definition run multiple times at the same time, with a different data context,

and in the diagram let it SEEM the CALL is running?

The definition is an object now. Let's not change THAT.

I'm having trouble in my head now, because up until now, the diagram was a clear

representation of what was actually going on.

-----

Isn't it jut solved if you just accept, to create a call object as a replica of a definition,

even though, then you are copying 'code' as well as data.

And just when running the parent command, you create all the sub-commands,

and their parameters, but leave them unfilled in and create the parent command's

parameter assignments. Then you have everything instantiated, that you need,

and you can just start at sub-command 1, run the input parameters assignments, and run it,

run the output parameter assignments (even when they immediately assign something to

parameters of another command), and then continue with sub-command 2.

That would just solve the problem.

Just leave that CPU comparison stuff in another article.

Just confess in that article, that it is just a brainstorm,

the goal of the brainstorm and that it is still not solved,

and then leave it unfinished indefinitely.

### Former Text

*{ Most of this is brainstorming. The more true conclusions are there under ‘What happens in current programming languages?’. }*

The problem is:

Parameter assignment calls have to point to

parameters. How can you point to a parameter,

if the call is not created yet?

You have to add to the command call article, that in a definition, the calls have to be created, because a definition has to specify what will become

input and output for the command call. Or perhaps add that to sub-commands articles.

I am still totally confused about the creation behavior of calls.

It looks like a call object has to be created as soon as its parent call is created, because input and output references have to be defined?

No. Input and output references are not defined. Input and output references are actually assignments, that only look like references. Yeah, but the assignments reference the parameters.

Perhaps you have to separate the creation behavior from the explanations of the general terms.

-----

Eventually it’s all call structure, for which definitions are just blue prints.

It looks like after a call’s created, based on the definition, input is set, it starts executing. First the first sub-command is created, based on its definition, input is set, based on the parent command’s definition, and it starts executing.

The assignment commands of parameter assignments are special. They do not count as the sub-commands of a command.

So what’s complicated about this? Well… the fact that when you process a sub-command, to set its input, the parent command’s definition must be consulted.

So what if after you created the parent command, based on the definition, you set input and then, before executing, the sub-commands are all created, and their input set, but not their sub-commands.

Only when you are about to *execute* a sub-command, its own sub-commands are created. But before that the parameters had already been assigned…

But crap. That works for reference parameters, but not for value parameters. You have to set value input right before you execute.

Create the parent command, based on the definition

Set input of parent command

Create all sub-commands,

input, output assignments taken over from parent command

set input of first sub-command

execute first sub-command

get output from first sub-command

So just before executing a parent command, you don’t only see the parent command’s input, but you also create all sub-commands, but leave their sub-objects empty. And then you copy the parameter assignments from the parent command definition.

Then you have the material ready to start executing the sub-commands one by one. The parameter assignments are executed appropriately before and after a sub-command executes.

So the connection between a sub-command parameter and something out of the parent command is not part of the parent command’s definition, but the assignment call that makes the connection between a sub-command parameter and something of the parent command is part of the parent command’s definition. The assignment call is only executed as the parent command executes.

So it’s all about getting the material ready for a parent command to execute. The creation behavior of a call is indeed still to be worked out.

> I do not think it is mandatory to immediately create sub-command-calls and their parameters for a call-instance