A Brief introduction to the Falcon Organic VM

Giancarlo Niccolai
The Falcon Programming Language

1. Background

The Falcon Programming Language has the characteristic of not concentrating on a single programming paradigm, but to actively search for integration across more paradigms. While it's relatively common for modern "scripting" languages to integrate elements from various "programming styles", presenting elements of OOP as classes and objects, concepts from functional programming as first-class function items, and a touch of aspect-oriented programming via control of object property access, a the programmatic exploration of different paradigms in the search of a possible integration is something relatively new.

What we're trying to achieve with Falcon is a meta-paradigm that puts the mind of the programmer and the solution of the task at hand into a direct relationships by giving him all the tools he needs, or might need, to model the most adequate solution under any circumstance. This aim requires more than just putting in cool functionalities taken from various pre-existing approaches. For instance, having functional sequences as object properties, or sending messages through look-up table callbacks opens whole new dimensions to approach coding.

Another primary goal of Falcon is that of achieving this goal at unprecedented performance. While dynamic and untyped language by choice, and so, falling in the category of scripting languages, we always advocated the need for raw performance in the inner part of the sytem; performance and simplicity of data transfers between modules (or host applications) and the script code was always on top of our concerns, with a particular care for garbage collecting strategies and integration of native code.

While having reached significant goals, the previous virtual machine model reached its limits when we tried to push our search for new paradigms in the realm of rule-based programming.

Also, the complexity of the integration of the scripting language with third party applications advocated for a revision of the integration API. While our integration model provides better performance and unprecedented level of integration between the scripting environment and the native-code sides, it required a more careful development of the glue code. As projects are often short of time before deadlines, while performances are a secondary concerns in many fields, this could have had a negative impact for the wide adoption of our language.

Finally, among the various projects and paradigms that we decided to integrate, evolutionary programming begin to take a relevant place. The experiment of TEG (The Evolution Game) showed the significative ability of some part of our programming model (mostly the functional programming) to evolve dynamically. Seen the potential of this elements, the rigidity of our originary OOP and procedural models begun to seem inadequate, also because we spotted the possibility to make them more fluid and more efficient at the same time.

Theese are the reasons that leaded to the massive effort of redesigning the Falcon scripting engine we're undertaking. In the rest of the article I am going to introduce our findings and to report about the status of this effort.

2. Engine outlook

A scripting language needs many different elements to be integrated into a single entity that is then proposed to user applications as an "engine". The number of elements and the complexity of the engine may vary depending on the goals of the scripting language. If the goal is that of just provide flexible and configurable code into a host application, it may just provide a way to run source scripts. If it aims to provide a way to build complex applications and to solve complex problems on its own, it should also provide an environment that can be used by the foreign code to interface with the system. The elements that we're offering in the new engine are the followings:

- Compilation of source code in finite units (modules).
- Compilation of source code into discardable code snippets.
- Sharing of pre-compiled modules, generated at remote location.
- Security control and jail-setting for running untrusted script.
- Comprehensive support for debugging facilities.
- Transparent access to an abstract, virtual I/O system.
- Transparent support for international text I/O and handling.
- Data sharing mechanism, with granular selection of garbage-collection sensible data.

And of course, a mean to run code. That's the central part of the engine, and possibly one of the most interesting ones: the organic virtual machine.

3. The organic virtual machine

Organic things feed, grow, reproduce and eventually die. Most notably, they do that in an environment, and if they do this well enough, they spread and survive.

The idea behind the new virtual machine we designed for Falcon was that of an object feeding on code incoming from the outer world. The code itself should have not been known in advance, but could come from different sources so that the VM might respond to new inputs maintaining its internal coherence. Possibly, the code could have come from the code itself, so that evaluating a certain code would have resulted in more code to evaluate.

On this basic mechanism, more complex code constructs could have been then used to do more complex things. For instance, an "if" construct would have selected one of the possible outcomes by checking the environmental circumstances and then selecting one of the pre-recorded code paths to be executed next by the VM by directly feeding the code in the VM.

Another aspect we wanted to introduce in the structure of the new virtual machine was the meaningfulness of the executed code. The code that the VM executed should not have been just random bits of code; it should have had self-consistent meaning. This meant that the code executable directly by the virtual machine should have had the ability to map itself to the original source code that generated it. Normally, the "compilation" process destroys the source code by creating machine-executable code; in fact, the compilation process consists in creating instructions for a certain physical or virtual machine to perform what is required to be performed by the source code.

We wanted to surpass this model and try to have the VM execute directly the instructions that were stated in the source code. The reason for this has something to do with symbolic evolutionary calculus, but I won’t digress about this topic in this article. Let me just point out the fact that in this way it is always possible to inspect the code that is being executed, and modify it by navigating through it.

Here is how we solved this problem.

3.1. Organic virtual machine structure.

The Falcon organic virtual machine uses a support structure that contains the current status of an execution path, called *context*. More or less, you could consider the context as the data needed by an execution thread; by swapping a context, the VM could safely execute another code line. Copying and restoring a context would allow to reset the execution of a process from that exact point.

The context is considered opaque and unclean during the execution of code, and becomes coherent each time the VM main loop is re-entered. As such, it is always totally safe to swap, inspect, suspend and resume whole contexts while the processor is in the VM main loop or if the main loop has exited. Also, it is possible to single-step or enter-exit from the main loop always leaving the context in a coherent state.

Notice that the virtual machine supports global variables that can be shared across contexts. However, using that space is at discretion of the executed programs.

Each context presents three stacks to the virtual machine:

- The *code stack* is the sequence of code that must be executed.
- The *data stack* is a place where temporary values are stored.
- The *call stack* is the sequence of major code units (called *functions*) that have been encountered during the processing of the code, and determines what portions of the code stack and data stack are currently visible to the virtual machine.

The activity of the virtual machine is that of checking the item on top of the code stack, called *code frame*, and execute it. This goes on until the code frames are exhausted or until an uncaught event reaches the main loop.

3.1.1. The code frame

The code frame is composed of an elementary execution unit, called *PStep*, and of an integer value called *sequence id*. The PStep is an object instance, providing callback a virtual function which is executed by the virtual machine[1]. The PStep has also support for textual rendering (i.e. to display a text representation of itself), and pointers to a source file that generated it, if there was any. However, a PStep is a totally arbitrary execution unit, whose origin is unknown. For instance, it might be explicitly created by extensions (modules or user embedding applications) for their own purposes. The sequence id in the code frame is an integer number that is at disposal of the PStep to record its status. It's called sequence-id because its main purpose is that of recording a linear phase or status in a PStep that should be increased at each visit, but it's actually at PStep disposal for any purpose it might wish to use it.

At VM level, the execution of a PStep is totally opaque, so each PStep may be considered the correspondent of an OP-Code in a traditional virtual machine. The execution of a PStep doesn't exhaust it; in fact, the PStep may decide to inject new PSteps after itself in the execution context, or to return the control to the calling VM without removing itself; this will cause the same PStep to be repeatedly called until it decides to remove itself explicitly from the code stack, or until an external event causes it to be modified.

3.1.2. The data frame

The data stack is actually a sequence of Falcon *items*. The item is the atomic data entity at script and engine level; native C++ data can be accessed through the *Item API* which we'll describe later on. The data frame is the topmost item in the data stack, and it is considered the "value" of the currently ongoing code evaluation.

3.1.3. The call frame

The call stack holds a list of call frames; each frame represents a discrete step in a symbolic source code execution unit. For simplicity, we call this execution unit *function*, even if the concept here is wider than in the classical formulation.

The call frame holds information about:

- the position of the code (the source were it was created), the amount of parameters that were passed to the function
- the *self* entity, or in other words, the object that is currently subject of *method* application (in the OOP meaning), which is stored as a Falcon *item*.
- The status of the data and code stacks when this execution unit was invoked.
- The status of the stack when the last *rule* was confirmed.
- The status of the nearest *catch point*[2].

Other than providing vital information about the significance and environmental information of the currently executed code, the call frame is used to perform *return*, that is, to discard PStep units and the data they generated and reset the status of the context to the moment in which the call stack was generated.

3.1.4. Notable types of PStep subclasses

The PStep, that is, the minimal Organic VM execution unit is an abstract class. As such, it might be provided by foreign code, or the engine itself may provide special PStep to perform special tasks. For instance, the initialization of an *hyperclass* requires several dedicated PSteps to be iteratively fed into the virtual machine.

However, there are several prototyped PStep subclasses that have a special significance to the Engine, and while their usage is not mandatory, the engine itself uses them regularly. As they might be valuable resource for third party code, they are part of the Falcon Engine Specifications:

- *Expression*: A special PStep meant to represent an unary, binary or ternary infix expression. Expressions are limited in number and enumerated through an *expression type identifier*. They have also a standardized behavior: take one, two or three items from the top of the data stack (depending on their arity) and convert them into one single item.
- *PCode*: A stack of PSteps that are executed one at a time. It's mainly meant to be used by an expression operation called precompile; to reduce the number of operations on the VM data stack, expressions precompiles themselves into a root PCode (which is generally held in the statement hosting the expression) and that PCode is pushed instead of the single expressions. However, PCodes may be used to store any sequence of PStep that is known in advance and have to be executed always in the same order[3].
- *Statement*: A PStep specialized as a top-level language statement. They are limited in number and enumerated through a *statement type identifier*, even if an "unknown" ID allows to create extended statements. Similarly to the Expression case, their behavior is structured: they must abandon the virtual machine removing any extra item left in the stack by elements they control. In other words, the size of the data stack before and after their execution must be the same. They are said to be *stack-neutral*, while expressions are *stack-active*.

- *SynTree*: A linear sequence of Statements, to be executed one after another. Many statements hold a SynTree themselves, and then the combination of SynTree and statements forms a complete source program definition.
- *RuleTree*: A special SynTree representing a *rule context*. The *rule* is a new construct introduced in the current version of Falcon. Each statement is catalogued as deterministic or non-deterministic depending on some predefined rule; when a statement is encountered, if it is formed by an expression evaluating to true the evaluation continues, otherwise it is interrupted and the nearest non-deterministic statement is tried again, until all the possible alternatives are exhausted. The RuleTree implements the vast majority of the operations needed to support the rule construct.

4. The item model

The Falcon Organic Virtual Machine knows just one type of data: the Item. Each item is set of five elements:

- The *Class*, also known as the item handler.
- An opaque *user data* (void* in terms of C++ code).
- A optional function that can be applied to the Class-data pair (called *method*).
- An optional integer *type identifier*, or simply type.
- A set of options called *flags*. Most of them are for VM internal use, but some, as the *out-of-band* bit, is available to the user.

Theoretically, the virtual machine uses the Class entity to manipulate the opaque data, and relies on flags for internal handling (copy-on-write support, garbage collecting are the most important ones). The type ID is at disposal of the user to check for item compatibility at script level, or to simplify the writing of extension modules.

Actually, there are several optimization strategies that rely on well-known type IDs for common item types (numbers, strings, boolean values, nil special value and so on) and define how the items are internally represented. However, this can be now seen as an implementation detail which isn’t affecting the theoretical model. Even "flat types", as 64-bit integers, can be handled as a pair of Integer class and of a pointer to a 64 bit integer, and in various parts of the engine, they are actually seen in this way. An engine-wide map of well-known type IDs and handler classes allows a transparent transformation of the "compressed" item representation into the expanded standard model.

4.1. Item Classes

The Class of an item identifies what operations can be performed on it. A set of operand overrides, called *class ops*, are serving for the purpose of applying expression results on items. For instance, the *op_add* member of the Class indicates how the VM should handle addition operations[4].

Other important features that Class handlers offer to the VM are the following:

- Instantiation (creation) of new items, and initialization of new instances.
- Serialization and de-serialization of items
- Cloning of items
- Item marking and garbage collection control
- Access to properties and methods offered by the items (dot accessor)
- Access to elements offered by the items via (square parenthesis accessor)
- Destruction of unneeded items

It is important to note that the Class interface itself doesn't present any pre-determined OOP model; the only concession to OOP built in the class interface is an abstract method that can be used by some special classes to determine direct parents of a class, but the method can be usually ignored by extension code. In other words, the basic Class interface acts as a simple *data handler* that exposes an object to the Virtual Machine and to foreign code.

In fact, the Class doesn't present methods meant to publish the contents of an object towards third party, abstract code; mostly, a Class must expose code gluing an opaque object into the Virtual Machine. If abstracting an object at user code level is desired, the user must implement its own solution to publish arbitrary data, or rely on some specialization of the base Class interface offered by the engine.

4.2. Special classes

While the engine and the virtual machine don't are neutral with respect to the class type, and a user can create new types by simply providing a new class to handle them, the engine offers a set of classes that have a special meaning at language level and that can be used also outside the engine context in case they are found useful. They are namely:

- *FalconClass*: This represents a class respecting the Falcon "class" statement declaration OOP model. FalconClass model supports multiple inheritance, with the ability to access the parents, and expose set of property and methods (but where properties can be overridden by callable items, or re-defined as methods in subclasses). FalconClass can have an unlimited amount of parents as long as they are all FalconClass. Thanks to this limitation, it is possible to create optimized instances as flat structures of Falcon Items, where access to each property is extremely simple and fast.
- *HyperClass*: This classes implement the Falcon "class" statement OOP model, but remove the limitation of parent classes being just other FalconClasses. HyperClasses support any kind of Class as parent (even third party, foreign classes), as long as the properties it offers stay constant[5]. At creation, the HyperClass queries all its parents for offered properties and creates a property map that is later used to handle property access requests to the right class handler, which will receive the correct opaque data it expects to handle. The engine automatically creates an HyperClass out of a source script "class" declaration, when it detects that at least one of the parents declared in the *from* clause is not a FalconClass.

- *Prototype*: A prototype is a class whose structure or inheritance (both in terms of set of base classes or in terms of composition of those base classes) may change during its lifetime. Any Change of the structure of a prototype is immediately reflected on all the instances of that prototype, and on all the instances of prototypes derived from the changed one.
- FlexyClass: It's a typeless class that whose instances might have different structure. In short, a FlexyClass instance is just a mutable dictionary of properties.

Other than this there is a very important class called *MetaClass*, which is the class handling instances of items derived from Class.

Notes

1. Actually, the structure of the PStep class and of the code frame are slightly different, but we're not discussing about implementation details and performance optimizations here. We're just presenting the logic structure of the abstract organic VM.
2. Actually we're studying the possibility to introduce a fourth stack that can be used to unroll the other three stacks when an exception is caught.
3. Actually, some expressions (namely, shortcut binary AND/OR, and short-if expression) require to be executed under a PCode, as they suppose that their current execution frame is a that of an host PCode. Theoretically, it is possible to create a different set of expressions not assuming this fact so that they can be used in other contexts, but at the moment we didn't find any practical usage for that. Just, we rarely use expressions outside PCode frames, and when we push them directly in the VM we know what we're doing. On the other hand, a PCode makes no assumption on the PStep it holds.
4. The most common expressions perform some simple operations on well-known types internally, without using the support provided by their class, but that is to be considered a simple optimization. Even in those cases, the class still provides the ability to be used in place of the built-in code.
5. Actually, an HyperClass can host even parent classes whose structure may change. Just, they won't be notified about new properties exposed at a later time by a parent class, while they will still route not existing properties even if they are removed.