Assess Applicability of the Functional Programming Paradigms in Embedded Hardware

Danny Benlin Oswan
Petra Christian University
Siwalankerto 121-131
Surabaya 60236
(+62) 312983455
overbored_dundee@yahoo.com

Michiel W. Koehorst
Fontys Hogeschool ICT
Rachelsmolen 1
5612 MA Eindhoven
(+31) 885080000
m.koehorst@fontys.nl

Marcin Gramza
Philips Lighting
High Tech Campus 45
5656 AE Eindhoven
(+31) 402791111
marcin.gramza@philips.com

ABSTRACT

Programming in embedded lighting domain is commonly done using the C language with the Object Oriented programming paradigm at Philips Lighting. However applying that paradigm in combination with the low-level language like C creates a conceptual gap between the requirements and design and actual implementation. This results in reduced source code readability and maintainability. Functional programming paradigm was expected to alleviate this problem by reducing the gap and enhancing readability. A proof of concept was built on an advanced, IP-connected, digital LED driver (Power over Ethernet) device. The actual code was inspired by the rule-based decision engine concept developed by EnLight.

Based on the hardware specifications of the device, the existing code to communicate with, and adherence to the functional paradigm, Lua was chosen to build the proof with. The implementation of the decision engine was altered to exploit characteristics of functional programming, such as representing actions as functions rather than as an enumeration value, using the common filter function to replace loops, and many more.

The proof of concept was able to run in the device. It was also relatively more readable and maintainable. However, it was slightly slower, less memory efficient, and less capable in dealing with low-level problems such as garbage compared to the engine in C language.

Keywords: Functional programming paradigm, Decision engine, Embedded, Lighting

1. INTRODUCTION

In developing embedded programs to operate lighting components, Philips Lighting uses C language. Meanwhile, requirements in program designs are defined as a set of functions, and translation to the software implementation creates some understanding gap. Philips attempted to close this gap using the object-oriented paradigm. However, compounded with fact that C is not designed for object-oriented paradigm, implementation results in boiler plates and verbose code, reducing readability. As a program gets more complex, maintainability rapidly drops.

The functional programming paradigm then draws the interest of Philips Lighting. It is expected that using said paradigm the gap between the requirements and implementation can be reduced. Boilerplate codes can be removed, along with overall reduction of code length. This could lead to fewer points of failure, less code to test, and overall increased maintainability.

However, there is also a possibility that the functional paradigm is inapplicable in the embedded domain. All of these aspects needs some dedicated research to uncover, which was the reason for this work. A proof of concept was built on an embedded hardware, starting with Power over Ethernet (PoE) device from Philips. It was based on a part from the EnLight Project, a rule-based decision engine.

2. COMPONENTS

Important components of this work are the hardware device on which the proof of concept is built, the functional programming paradigm itself and the language used, and the decision engine as the proof of concept.

2.1 Embedded Hardware Device

An embedded device used to control luminaire wirelessly was provided to build the proof of concept on. It uses the STM32F427VG microcontroller. Highlights are 1024kB flash size and 256kB RAM. Software code are in C, using a simple operating system OpenRTOS. A part of C standard libraries is available in the device. Several mandatory modules of the device are modules for memory management and circuit board control. Other optional modules are event handler, IP stack, ZigBee wireless protocol, LED driver, TFTP server and file I/O utilizing the flash memory.

2.2 Functional Programming Paradigm

Functional Programming Paradigm was based on a formal system called lambda calculus, devised by Alonzo Church in the 1930's as a model for computability, along with Turing Machine and recursive function theory. It is mainly a system for manipulating lambda expressions, which could be a name, a function, or a function application [5].

The first actual functional programming language, was LISP, which is also the second oldest programming language still in use from

1958. Despite not actually being originated from it, all recent versions of LISP are lambda calculus based. Multiple other functional languages follows the development of functional programming, such as ISWIM, ML, Miranda, and Haskell, which is the latest well known purely functional programming a language. Certain characteristics of functional programming took form between them, including lazy, higher order, polymorphically typed, pattern matching, and list comprehensions. [6].

Due to its stateless nature of the paradigm, input and output are handled in a more complicated way and may vary between functional languages. Current research are trying to utilize the stateless nature to better handle parallel programming. The potential of using the paradigm for certification and proving correctness of a program is also being observed [1].

Ten programming languages capable for functional programming were examined at the beginning: F#, Haskell, Scala, Scheme, Clojure, OCaml, Erlang, Groovy, Lua, and MATLAB. Based on the hardware specifications of the embedded device, the chosen language must be able to run with very low memory and disk space and to communicate with C code. It is also preferable for the chosen language to be able to handle input and output easily, as well as having low building complexity.

Among the ten languages, Lua was chosen to build the proof of concept with. It has the smallest size, and being embeddable, communicates easily with C language [4]. Being basically a set of C libraries, it is also built by simply including these libraries along with the actual hardware device's source code.

2.3 EnLight Project Rule-Based Engine

The EnLight Project was executed by a consortium of 27 companies from 2011 to 2014. The main goal was to develop a next generation energy efficient and intelligent lighting systems. Said system is based on networked luminaires, each are capable to adjust to their own sensory inputs. Each module is specifically programmed to achieve high efficiency and decoupling, as well as providing special functionalities required by the project. With this distributed control, the pilots achieved 40-80% energy savings, with the same or better user comfort [7].

One vital building block for the luminaire's intelligence is a programmable local rule-based decision engine inside the embedded controller, alongside a rule storage. Each rule consists of a triggering event, an optional condition, and one or more actions.

The engine works by receiving events, consisting of an enumerated event type and source MAC address, as well as arbitrary amount of integer parameters. Each rule inside the storage is then checked one by one, whether they have a matching address and event type. If they have a special condition, it will also be evaluated. Condition can check whether a luminaire level is currently in/active, compare two integer arguments, or check multiple sub-conditions using logic gates. If a rule is matched and its condition evaluated into true, all actions inside it will be executed. The engine will then move on to match and evaluate the next rule.

There are also a large variety of possible actions, such as configuring the luminaire, setting up an internal variable or time, generate a new event, or de/activate luminaire levels. Levels themselves are virtual settings of luminaire. The logical luminaire component can have multiple levels of luminaire configuration, and the highest level active will affect the actual physical luminaire settings.

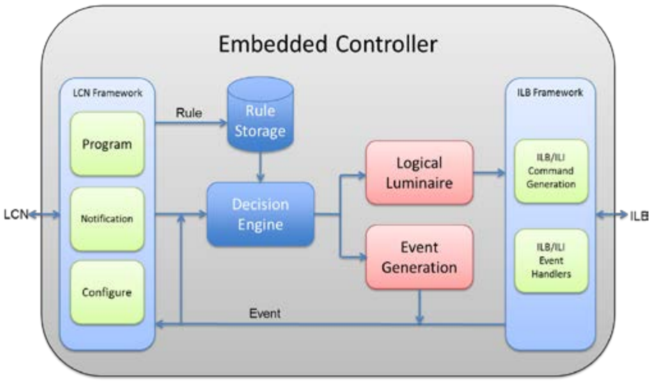

Figure 1 EnLight Embedded Controller Block Diagram

As the block diagram in Figure 1 could show, the rule storage is separated from the engine. To reflect this, the program built in Lua also have two separate scripts, one for the engine and one for the rules.

3. IMPLEMENTATION

Before being able to integrate Lua into the embedded device to start the implementation, several modifications must be done on both Lua and the device. Most optional modules of the device was turned off to free up memory, and allocated heap and stack were increased to use the entire 256 kB RAM. Inside the Lua C libraries, all references to FILE stream were deleted, as did all functions that rely on it. The same holds true for several other streams and functions, such as fwrite, setlocale, stdin, stdout, and stderr. As a result, three optional Lua libraries, I/O, Debug, and OS were deleted as whole, along with several functions dealing with printing, error logging, and files. Additionally, memory functions inside Lua function l_alloc were replaced with equivalent functions available for the hardware device. Meanwhile, all Lua optional libraries, excluding table library, were not loaded to reduce the interpreter memory, leaving only the base and table library.

All this modifications allows Lua to run inside the hardware device. As a consequence, however, the ZigBee module needed to wirelessly control luminaires was turned off. The device was also only able to process IP packets for TFTP server and not for the event handler. To compensate, onboard LEDs were used as actuators, and an internal task was set to periodically fire events into the device itself as input. Also, since FILE stream are removed, the scripts are loaded into Lua as strings instead. The rule scripts, which is relatively small, can be loaded through the TFTP server and stored in flash memory, while the larger engine script are included along with the C source code as C-string object.

3.1 Functional Practices

There were several notable common functional programming paradigm practices frequently used in building the new decision engine. The basic one was immutable variables and absence of side effects. Other than returning the requested value, any statement and function must not change any variable. A variable, once given value, never change. Whenever a new data is needed from an old one, a new instance variable is created with the value of modified old data. The old data itself is unchanged. This allows functions to become referentially transparent: being provided the same input, a function will always provide the same output. While this practice could not be enforced entirely, as the luminaire output and rule storage were both global states, it was applied to as many variables inside functions as possible.

The next one was first-class functions, where functions are treated as normal variables: functions can be passed as arguments, returned, assigned, stored in data structures, and created at runtime, like other primitive values such as integer or character. This allows for higher-order function: a function that receives another function as parameter or returns one. This was extensively used in the implementation. Instead of using type enumeration and branching construct to decide which function to use for a certain type of condition and action, references to the function itself are stored to identify the condition and action types. The engine then send the stored parameters inside the rules to these function references when evaluating conditions or executing actions.

After that was array functions, some common functions provided specifically to manipulate arrays. One aspect that functional paradigm avoid is iteration. In addition to what the iteration is trying to do, the presence of either loop counter or variable used as control statement already uses mutable data. Since iterations mostly deal with the manipulation of data arrays, these functions are used to replace them. They generally receive an array and another function as parameter, along with other needed data. Such functions do not exist in Lua, but they can be manually created [2]. During the implementation, four kinds of array functions were created: each (applying a function to each array element), reduced (reducing an array to a single value using provided function), filter (return another array containing only elements that is filtered by the provided function), and map (return another array with each elements transformed by the provided function). These four functions were used to replace most loop constructs of the engine. The filter function in particular is used to evaluate condition of rules. Due to how it works, rules are all evaluated at once instead of one by one.

For array manipulation iterations that are not solved immediately with array functions, recursions are used. Tail call optimization is also important to prevent stack overflow from happening, especially in a resource constrained device such as the one being used. Since most loops were already replaced with array functions, only one was left and was replaced by a recursive function.

There were also other practices examined but then discarded because its usage was inappropriate in the context, or the chosen language does not support them. Currying transforms a function that takes multiple arguments into a function that takes just a single argument and returns another function if any arguments are still needed. Composition pipelines the result of one function to the input of another, creating a single function composed from two or more. Benefits of both of these practices are rather insignificant compared to the amount of memory needed in the resource constrained device. Lazy evaluation, delaying the evaluation of an expression until its value is needed, is not supported by Lua. So does pattern matching.

3.2 Other Conversions

There were several other conversions from the original EnLight engine to the new one that were unrelated to functional programming paradigm but were necessary due to limitations in Lua or to take advantage of several features. Rules, which are a set of data, were loaded using a callback technique in which all data are passed as a parameter to a loading function [3].

Hash tables were used to represent both objects and arrays. The rules in particular were no longer stored sequentially by their triggering event. Instead the new engine stores the rules in hash table based on trigger. When matching rules with events, instead of iterating through the array and matching the content inside the rule, the engine will use the trigger event on the hash table as key to obtain a list of rules associated with that event. Event parameters were stored in a global table instead of passed between all functions in the original engine. The new functions could access them, but they were more readable with one less parameter. All integer parameters and arguments were signed 32 bits integer. Lua only has one type of integer, thus unable to differentiate integer types as in the original engine. To ensure integer division also returns an integer, the "//" operator was used. The error handling was extended to also remove rules which causes error during runtime, instead of just logging them.

Due to how Lua deals with variable naming and strings, temporary variables across functions were given same names to save memory. The names were also relatively short in length. This was later toned down due to oversimplification making the script more difficult to read.

3.3 Special Features Used From Device

Due to the differences between the hardware device used by EnLight and the one used in this work, the engine had to be modified to compensate for missing features from the original device. The engine was made running alongside the event handler module to be able to receive events that were fired internally. The device used in this work seemed incapable of detecting MAC address, so IPv6 address were used to determine source of event. It was also difficult to implement a class with array of bytes in Lua, so they were represented as strings. To save memory usage, the new engine did not involve logging process. There were also less event types and luminaire configuration action types as they were not supported by the device used. The new device also did not have an internal clock, so set time action was not supported.

One major modification was the usage of timer for event generation action and level timeout. The timer in the used device was much more limited compared to the one in the original engine. First, each timer can only fire an event once before having to be reactivated manually. Second, timer must be created on startup, and the fired event cannot be changed. Third, event fired by timer cannot carry any data. In the original engine, whenever a generate event and timeout was instructed, all data was passed to the timer module to be managed. This is not possible due to the limitations above. The new decision engine was made responsible for internally storing data for event generation and level timeout inside a hash table. The timer only needed to fire its own unique event, an enumerated number. When this number was sent to the engine, it was used as a key to find the appropriate data from the hash table to generate the event or trigger the timeout.

4. EVALUATION

Two aspects were evaluated from the decision engine: performance, to measure the capability of the new engine, and readability, which in turn could reflect maintainability.

4.1 Performance Evaluation

Using between one to three actions for each event, the new engine takes between 9-14 milliseconds for each processed event. By default, event handlers in the embedded device are restrained to take at most 10 milliseconds to process an event.

For memory requirements, creation of a Lua state in C takes 3480 bytes, the base and table library take 3288 bytes, and the engine needs 40664 bytes. The memory needed for rule varies depending on the amount of rules loaded. Eight rules, for example, requires 9160 bytes. Meanwhile, 24 rules require 27984 bytes and 25 rules require 29064 bytes. Extrapolating this calculation into a chart will yield a linear chart for rule memory usage, shown in Figure 2. One rule would need 1200 bytes on average.

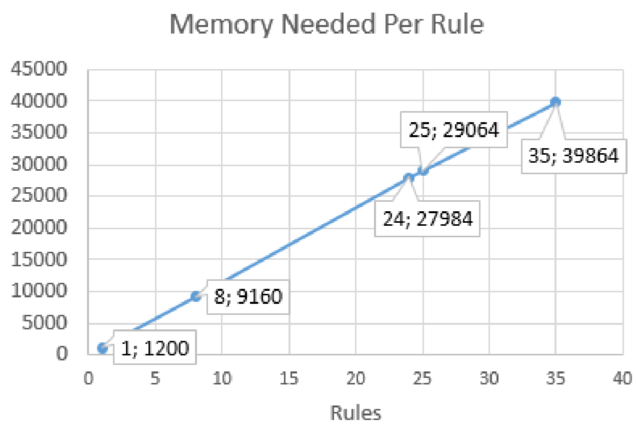

Figure 2 Calculation Chart of Rule Memory Usage

There was a problem regarding garbage that prevents testing of higher number of rules. The automatic garbage collection of Lua was too slow, it did not run on certain number of events and run after several, causing processing time to rise to up to 20 milliseconds. The garbage collector had to be run manually to achieve consistent processing time. Also, right after the loading of the rules, the engine will accumulate a large amount of garbage, up to 22 kilobytes when using 25 rules. While this garbage can be collected manually to allow runtime memory needs, it completely exhausted the memory, preventing more rule from being loaded.

4.2 Readability Evaluation

A small survey was held with 40 people by showing them two rule scripts. The first script was an original EnLight rule script in XML format from EnLight project. The second was a rule script in Lua made to be similar in functionality with that original script. These people were then asked which format was more readable, understandable, and preferable to write with.

A majority, 33 of them, said that the new script is more readable, but only 19 agreed it is more understandable. Since the new rule script involves much less characters, they are faster to read. However, this also leads to lack of overview and sense of completeness. Participants who prefer reading XML were used to deal with XML. They criticized the amount of curly brackets involved in the new script.

Lastly, 24 preferred to write in the new script's format. Some participants pointed out that less characters means less room for error and less parsing power needed for the program. New script's simplicity might help new users in learning the format. Meanwhile, participants who preferred to use XML mentioned that, since XML is a standardized format, it is easier to find people who are already familiar with XML and need no training. Presence of standard also allows for "safer" feeling when using XML format.

The most common suggestion for the new rule script was standardization, particularly to JSON. The new format is close to JSON format. Standardization allows for syntax checker and finding experienced people. If JSON schema is used, then clear layout will also be achieved. Other common suggestion was providing comments to better describe the rule-script.

5. CONCLUSION

It is possible to use the functional programming paradigm in the embedded lighting domain. However, some practices are less exploitable in this domain compared to other possible domains, which ideally involves more data arrays and less states. If communication with existing C-code is mandatory, Lua should be used. Said existing code slightly limits functional programming. The functional paradigm creates a more readable new decision engine. However, it is slower and consumes more memory for equivalent functionality in original C language. Due to high-level characteristic of the functional language, memory management such as garbage collection and loading from flash is difficult. Using a less popular standard in the new program could also lower interest and understandability.

6. REFERENCES

[1] Barendregt, H. P., Manzonetto, G., and Plasmeijer, M. J. The imperative and functional programming paradigm. in Cooper, B. and van Leeuwen, J. ed. *Alan Turing — His Work and Impact*, Elsevier, Boston, 2013, 121-126.

[2] Chisholm, J. A Functional Introduction to Lua. *Pragpub: The First Iteration, 47*. 2013. 11-17.

[3] Ierusalimschy, R. *Programming in Lua*. Lua.Org, Rio de Janeiro, 2013.

[4] Lua. 2012. *About*. URI= http:// www.lua.org/about.html.

[5] Michaelson, G. *An introduction to functional programming through Lambda calculus*. Dover Publications, Mineola, N.Y., 2011.

[6] Turner, D. A. Some History of Functional Programming Languages. in *13th International Symposium, TFP 2012*, (St. Andrews, United Kingdom, 2012), Springer Berlin Heidelberg, 1-20.

[7] van Tujil, F., James, L., Creusen, M., and Stalpers, M. EnLight Project Outcomes. *LED Professional Review, 48*. 2015. 66-79.