Implementation and Evaluation of an Interpreter for Functional Reactive Programming on Small Embedded Devices

Go Suzuki
gosuzuki@psg.c.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Takuo Watanabe
takuo@acm.org
Tokyo Institute of Technology
Tokyo, Japan

Sosuke Moriguchi
chiguri@acm.org
Tokyo Institute of Technology
Tokyo, Japan

ABSTRACT

This paper reports the implementation of Emfrp-REPL, an interactive interpreter (REPL) of a functional reactive programming (FRP) language for resource-constrained embedded systems. Its goal is to accelerate the prototyping and development of microcontroller-based embedded systems. The interpreter runs on small-scale embedded devices based on 32-bit microcontrollers, such as ESP32 with 520KiB size data RAM. The evaluation shows that the memory usage of Emfrp-REPL is comparable to MicroPython, and the range of its latency is narrower than MicroPython, according to microbenchmarks.

CCS CONCEPTS

• **Computer systems organization** → *Embedded software*; • **Software and its engineering** → **Functional languages**.

KEYWORDS

functional reactive programming, embedded systems, Read-Eval-Print Loop

ACM Reference Format:
Go Suzuki, Takuo Watanabe, and Sosuke Moriguchi. 2023. Implementation and Evaluation of an Interpreter for Functional Reactive Programming on Small Embedded Devices. In *Proceedings of Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (MoreVMs '23).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A *reactive system* is a computational system that continuously reacts to external inputs. *Functional Reactive Programming* (FRP) is a programming paradigm that facilitates the programming of reactive systems. Its central idea is the use of *time-varying values* (aka *signals*), which abstract values that change over time (*e.g.*, sensor readings). By describing a system in terms of time-varying values, it is possible to avoid using techniques such as polling and callbacks, which are often used in microcontroller-based embedded systems and recognized as obstacles to readability, maintainability, and evolution.

Emfrp [2] is an FRP language designed for small-scale embedded systems. Its compiler translates source programs into standard C, which can be compiled into native binaries using platform-dependent C compilers. In many cases, microcontroller-based devices do not have enough resources to execute compilers and other tools, so we run them on the host PC and load the resulting binaries onto the target. Thus, the compile/load process is repeated for each program change, which hinders rapid development.

To accelerate the prototyping and development of embedded systems based on microcontrollers, we propose Emfrp-REPL[1], an interpreter for an FRP language that runs on various microcontrollers (*e.g.*, ESP32, ARM Cortex-M). As the name indicates, it provides a REPL (Read-Eval-Print Loop) where we can interactively develop programs using a terminal emulator connected to a target device via a UART or USB serial port. The language of Emfrp-REPL is a dynamically typed variant of Emfrp with some extensions. Unlike Emfrp, it can change time-varying values and their dependencies during execution, facilitating incremental development.

Emfrp-REPL interprets ASTs instead of byte codes. This is to reduce the code size of the interpreter itself. In addition, the interpreter constructs the environments for local variables lazily to minimize memory usage on updating time-varying values.

After a brief description of the Emfrp-REPL language, this paper presents the current results of the implementation and evaluation of the interpreter. We evaluated the memory usage and the end-to-end latency. The former is essential for resource-constrained systems, and the latter is also important for the responsiveness and real-time performance. Through comparisons with Emfrp and MicroPython [1], we show that Emfrp-REPL has adequate performance for some embedded systems.

2 LANGUAGE OVERVIEW

2.1 Node Definitions and Redefinitions

An FRP program is often expressed as a directed acyclic graph (DAG) with time-varying values as nodes and their dependencies as edges. For this reason, Emfrp refers to time-varying values as *nodes* and defines them with the keyword `node`. Emfrp-REPL also follows this tradition.

Listing 1 defines a node named `sensor`, `temp`, and `gpio0`. Note that '`#`' starts a line comment. Some nodes are classified as *input* and *output*, corresponding to the input and output values. In this example, `analog0` and `gpio0` are input and output nodes, respectively. The correspondence between these and the actual input/output (GPIO ports) is specified separately. We say that node X depends on Y if X references Y. Here, the `sensor` depends on `analog0`, `temp` depends on `sensor`, and `gpio0` depends on `temp`. The dependency should form a DAG. The interpreter repeatedly updates node values along the DAG. A single update is called an *iteration*. The expression X`@last` refers to the *previous value* of node X, *i.e.*, the value at the last iteration. The initial value of a node whose previous value is referenced is specified with `init[...]`.

MoreVMs '23, March 13, 2023, Tokyo,Japan
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]Available at: https://github.com/psg-titech/emfrp-repl

Listing 1: Node Definitions in Emfrp-REPL

```
1 # Value of the sensor.
2 node sensor = (500 * analog0) / 1024
3 # Temperature (Moving Average Filter)
4 node temp init[sensor] = (temp@last*8 + sensor*2)/10
5 # gpio0 connects to the motor.
6 node gpio0 = if temp > 30 then True else False
```

Emfrp-REPL allows redefining nodes at runtime. Listing 2 shows examples of node redefinitions. After defining nodes `A` and `B` in lines 1 and 2, `B` and `A` are redefined in lines 3 and 4 in this order. The interpreter correctly maintains the dependency inversion caused by this change. `B` is redefined again in line 5, which causes a cycle in the dependency graph, so the interpreter reports an error.

Listing 2: Node Redefinitions in Emfrp-REPL

```
1 node A = gpio0
2 node B = A * 2
3 node B = gpio0 # redefining the node B
4 node A = B * 2 # redefining the node A
5 node B = A     # Error, due to a cyclic dependency
```

2.2 Objects

Emfrp-REPL supports seven kinds of objects: integers, floating-point numbers, tuples, user-defined records (tagged tuples), closures, Boolean values, and nil. The sizes of integers and floating-point numbers are platform-dependent.

Tuples and records are immutable values. So the element of a tuple (a record) cannot be replaced. A record can be regarded as a tagged tuple. Listing 3 shows examples of using tuples and records.

Unlike Emfrp, closures in Emfrp-REPL are first-class objects. Expressions representing anonymous functions are also available. Listing 4 shows some usages of functions.

Listing 3: Tuples and Records

```
1 node someTuple = (1, 2)
2 node (fst, snd) = someTuple         # fst -> 1, snd -> 2
3 record someRecord(srItem1, srItem2)
4 node recordVal = someRecord(1, 2)
5 node itemVal = srItem1(recordVal)   # itemVal -> 1
```

Listing 4: Functions

```
1 node anonFun = func(a) -> (func(b) -> a + b)
2 func namedFun(a) = if a > 0 then 100 else 50
3 node funRes = namedFun(anonFun(analog0)(analog1))
4 func deconstFun(someRecord(i1, i2)) = i1 + i2
5 deconstFun(recordVal)  # -> 3
```

3 IMPLEMENTATION

3.1 Execution

Program execution in Emfrp-REPL is done by repeatedly updating nodes. As explained in Section 2.1, the dependencies of the nodes constitute a DAG, so a linear list of nodes is obtained by topological sorting. Each iteration (see Section 2.1) updates the nodes along the list. For example, in the case of Listing 1, each iteration updates `analog0`, `sensor`, `temp`, and `gpio0`, in that order.

For each node update, the interpreter traverses the abstract syntax tree (AST) of the expression that defines the node's value. The current implementation does not provide bytecode VM or JIT compilation to reduce the size of the interpreter. The interpreter uses two tables for name resolution. One is for local variables and function arguments, and the other is for nodes. The former corresponds to the environment commonly used by many interpreters. In our interpreter, table allocations occur only when needed to reduce memory size. For example, no table allocations are necessary in the case of Listing 1.

3.2 Object Management

The structure for Emfrp-REPL's object representation is just four words (1 word = 32 bits). The first word represents the kind of the object. The least significant bit represents whether the object is marked during garbage collection (GC). There are seven kinds: free cell, one-element tuple, two-elements tuple, many-elements tuple, symbol, closure, and variable table.

Integers and floating-point numbers are inlined. The low-order two bits of an object's pointer are used to distinguish between integers, floating-point numbers and pointers.

Tuples represent tuples and records. The remaining space in the structure is three words. One is a pointer to a symbol object to distinguish records. To reduce the amount of memory used, one-element and two-elements tuples are dedicated and there are no additional allocations. A many-elements tuple contains its length and a pointer to an additional buffer allocated in the heap unmanaged by GC to retain its elements.

The second word of closures represents a kind of the closure object. Four possible kinds: AST, callback, record constructor/accessor. An AST contains an AST written in Emfrp-REPL and a pointer to its environment. A callback contains a function pointer defined in the host C code.

We use Snapshot GC [3]. Snapshot GC is proposed for real-time systems, and is simple. GC is the one of barriers to real-time systems because of its pause time. Marking and sweeping all available objects is time-consuming and unpredictable. Snapshot GC proceeds incrementally with each allocation.

3.3 Defining Nodes

When defining a new node, there may be problems with its dependency, *e.g.* cyclic reference, missing identifier[2], and so on. To recover after failing to add a node, our interpreter records modifications to the node definitions. A simple modification does not require a complete topological sort algorithm. In such cases, copying the whole of the dependency graph is wasteful. Instead of copying, modifying the present node definitions and recording the modifications results in slightly better performance and less amount of memory used when checking dependencies.

3.4 I/O and Customizability

Input and output nodes are predefined by the host code because it is difficult to write a program for I/O in the Emfrp-REPL language. Our interpreter provides some APIs in Listing 5. The function `emfrp_repl` interprets from the argument `str` and returns into the argument `output`. This accepts not only node definitions, but also evaluates an expression such as `if a > 0 then 12 else 0`. The functions in lines 8 through 11 define input or output nodes. They

[2]It is not allowed to refer missing identifiers in expressions of the nodes because the nodes are evaluated soon after defining.

can take callbacks called when a value of the node is needed or the value of the node is changed. The function `emfrp_set_node_value` puts a value into a node. `emfrp_update` executes one iteration.

Listing 5: APIs of Emfrp-REPL

```
typedef struct emfrp_t emfrp_t;
typedef struct em_object_t em_object_t;
typedef em_object_t * (*em_input_callback)(void);
typedef void (*em_output_callback)(em_object_t *);
emfrp_t * emfrp_create(void);
enum em_result emfrp_repl(emfrp_t * self,
                          char * str, em_object_t ** output);
enum em_result emfrp_add_input_node(emfrp_t * self,
          char * node_name, em_input_callback callback);
enum em_result emfrp_add_output_node(emfrp_t * self,
         char * node_name, em_output_callback callback);
enum em_result emfrp_set_node_value(emfrp_t * self,
                 char * node_name, em_object_t * value);
enum em_result emfrp_update(emfrp_t * self);
```

4 EVALUATION

4.1 Evaluation Detail

We evaluated the memory usage on Emfrp, MicroPython [1], and Emfrp-REPL. We executed the iterations in Listings 6 through 10. The iterations are repeated 10 and 100 times by the host code (C) in Emfrp-REPL and by the guest code (Python) in MicroPython. Unmanaged heap means the value of `esp_get_free_heap_size()`. In the case of MicroPython, we measured `gc.mem_free()` which does not include the unmanaged heap. MicroPython does not allocate Python objects on the unmanaged heap.

We also evaluated the end-to-end latency. Each iteration is executed on rising and falling edges of the GPIO16 using interruptions as shown in Listings 11 and 12. We connected the GPIO16 to a function generator and input 100Hz square waves. We observed the latencies between the input (GPIO16) and the output (GPIO17) by probing them with an FPGA and an oscilloscope (Figure 1). We collected 120000 samples of each program. Table 1 shows the evaluation environment. We used the oscilloscope to verify the results of the FPGA. All resources and results (including snapshots of the oscilloscope, and the design of the FPGA) are available: https://github.com/psg-titech/morevms-2023-experiments.

In the tables in section 4.2, "(Direct)" means programs in Listings 6 and 9, "(Tuple)" is in Listings 7 and 10, and "(10 nodes)" is in Listing 8.

Listing 6: Assign directly in Emfrp and Emfrp-REPL

```
node gpio17 = gpio16
```

Listing 7: Assign via tuple in Emfrp and Emfrp-REPL

```
node (v, a, b) = (gpio16, 1, 2)
node gpio17 = v
```

Listing 8: Assign with many nodes in Emfrp and Emfrp-REPL

```
node (node0, a0, b0) = (gpio16, 1, 2)
node (node1, a1, b1) = (node0, 1, 2)
...
node (node9, a9, b9) = (node8, 1, 2)
node gpio17 = node9
```

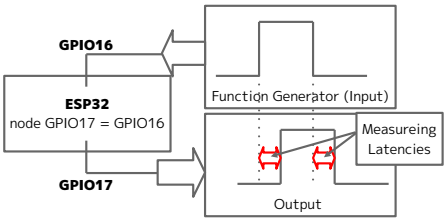


Figure 1: Circuit diagram

Table 1: The evaluated environment.

Microcontroller	ESP32-WROOM-32D (espressif)
SDK(Emfrp, Emfrp-REPL)	ESP-IDF v5.0
MicroPython	v1.19.1 from Official
Function Generator	KENWOOD FG-272
Oscilloscope (To verify)	Tektronix TDS 220 (The bandwidth is 100[MHz])
FPGA (To measure)	Terasic DE10-Nano (The sample rate is 50[MHz])

Listing 9: Assign directly in MicroPython

```
gpio17.value(gpio16.value())
```

Listing 10: Assign via tuple in MicroPython

```
(v, a, b) = (gpio16.value(), 1, 2)
gpio17.value(v)
```

Listing 11: Interruption in Emfrp-REPL and Emfrp

```
emfrp_t * em;
void irq_handler(void * p) { emfrp_update(em); }
void app_main(void) {
  gpio_install_isr_service(0);
  gpio_isr_handler_add(16, irq_handler, nullptr);
// ...
}
```

Listing 12: Interruption in MicroPython

```
def irq_received(un_used):
  # The program in Listing 9 or 10
gpio16.irq(irq_received)
```

4.2 Result

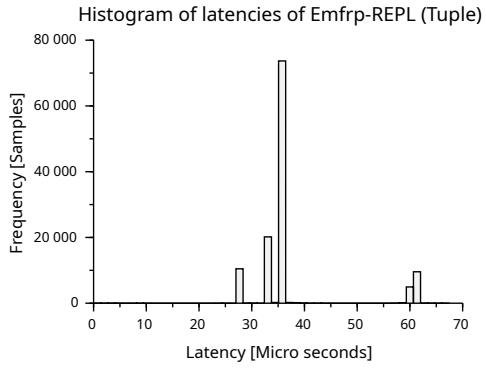
Table 2 shows the results of the available heap size. In Emfrp, a space enough to work will be allocated at the startup time because the Emfrp compiler calculates the size of the required space at compile time. Thus, `malloc` is never called during iterations. In Emfrp-REPL and MicroPython, an iteration without tuples does not consume memory because Boolean values are not newly allocated. Since the loop in MicroPython is written in Python, the memory consumption in MicroPython is due to a `list` object created by the function `range` and the byte code of the loop. Using tuples, Emfrp-REPL shows comparable result to MicroPython.

Table 2: The results of available heap size

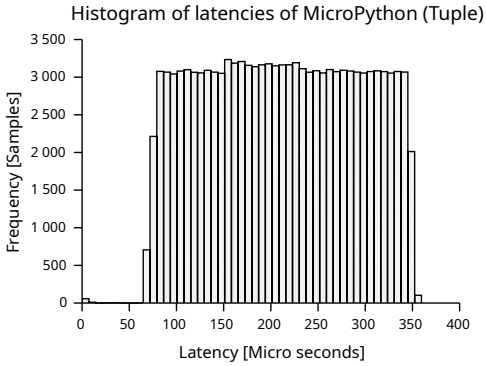
	Executed iterations	Unmanaged [Bytes]	Managed [Bytes]
Emfrp (Direct)	0, 10, 100	294544	N/A
Emfrp (Tuple)	0, 10, 100	294504	N/A
Emfrp (10 nodes)	0, 10, 100	294360	N/A
Emfrp-REPL (Direct)	0, 10, 100	139828	131040
Emfrp-REPL (Tuple)	0	139420	131040
	10	139260 (-160)	130880 (-160)
	100	137812 (-1608)	129440 (-1600)
Emfrp-REPL (10 nodes)	0	135720	131040
	10	134104 (-1616)	129440 (-1600)
	100	119688 (-16032)	115040 (-16000)
MicroPython (Direct)	0	-	108720
	10, 100	-	108272 (-448)
MicroPython (Tuple)	0	-	108720
	10	-	107808 (-912)
	100	-	104928 (-3792)

Table 3: The results of latency

	Mean
Emfrp (Direct)	3.39 µs
Emfrp (Tuple)	3.64 µs
Emfrp (10 nodes)	7.31 µs
Emfrp-REPL (Direct)	9.98 µs
MicroPython (Direct)	38.86 µs



(a) Emfrp-REPL (Tuple)



(b) MicroPython (Tuple)

Figure 2: The results of latency (scattered results)

The end-to-end latency results are shown in Table 3 and Figure 2.[3] 98% of samples of the results shown in Table 3 fall within ±0.05 µs of the mean. This is because Emfrp works without GC and GC does not wake up in Emfrp-REPL and MicroPython.

In Figure 2, there are variations due to GC. In Emfrp-REPL, there are two major intervals of approximately 35 and 65. The frequency of the GC phases is approximately Idle:Mark:Sweep = 3:0:1 according to the logs. (The log was taken at the beginning of each iteration.) The GC starts when the available heap space is half of the total heap space. Thus, the number of iterations sweeping unmarked objects is one-eighth of the total number of iterations. The frequency of the largest latency is also one-eighth. It means that sweeping tuples occurs the largest latency. By changing how many GC marks or sweeps at one time, we can narrow the range of the latency. In MicroPython, the result shows that the latency ranges from 75 µs to 350 µs. Since MicroPython uses naive Mark&Sweep GC, it takes more than four times longer in the worst case than in the best case. To make it smaller, the heap managed by GC must be reduced.

5 CONCLUSION

We designed Emfrp-REPL, a language for functional reactive programming, and implemented an Emfrp-REPL interpreter. We also evaluated its memory usage and its end-to-end latency, and compared with Emfrp and MicroPython. The results show that the memory usage is comparable to MicroPython and the range of the end-to-end latency is narrower than MicroPython.

[3]The result of Emfrp-REPL (10 nodes) is not included, see our repository. (https://github.com/psg-titech/morevms-2023-experiments)

REFERENCES

[1] George Robotics. Accessed Jan. 2023. MicroPython - Python for microcontrollers. https://micropython.org/

[2] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems. In *Companion Proceedings of the 15th International Conference on Modularity* (Málaga, Spain) *(MODULARITY Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 36–44. https://doi.org/10.1145/2892664.2892670

[3] Taiichi Yuasa. 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software* 11, 3 (1990), 181–198. https://doi.org/10.1016/0164-1212(90)90084-Y