Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems

Kensuke Sawada[1] Takuo Watanabe[2]

Department of Computer Science, Tokyo Institute of Technology
W8-75, 2-12-1 Ookayama, Meguroku, Tokyo 152-8552, Japan
[1]sawada@psg.cs.titech.ac.jp, [2]takuo@acm.org

Abstract

In this paper, we introduce a new functional reactive programming (FRP) language Emfrp designed to support small-scale embedded systems. An Emfrp program defines a system as a fixed directed graph whose nodes correspond to the time-varying values in the system. The language equips a simple mechanism that enables each node to refer the past values of arbitrary nodes. Using this mechanism, Emfrp provides simplicity and flexibility for describing complex time-dependent reactive behaviors without space and time leaks. Our Emfrp compiler produces platform-independent ANSI-C code that can run on multiple processors including resource constrained microcontrollers. To demonstrate the capabilities of the language, we show a simple but non-trivial example application.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Control structures; D.3.2 [*Language Classifications*]: Applicative (functional) languages

General Terms Languages, Design

Keywords Functional Reactive Programming, Embedded Systems

1. Introduction

Reactive programming[1, 13] is a complicated and stressful task for developers using traditional programming paradigms. Handling of asynchronous events and/or time-varying values is essential for reactive programming. However, conventional methods, such as callbacks, polling and state-machines are obstacles to modularity because they tend to cut a control flow into small pieces. Also, this makes testing, maintenance, and extension of a program using these methods laborious tasks.

Functional Reactive Programming (FRP) is a programming paradigm that supports reactive programming using (purely) functional building blocks. Since its invention[5], FRP has been actively studied and applied in many areas such as animation[5], robotics[10, 6], web applications[9] and GUIs[3]. Recently, several FRP languages and frameworks that can bear to practical use have been developed (e.g., Elm[3, 4] and FRPNow[16]).

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-4033-5/16/03...
http://dx.doi.org/10.1145/2892664.2892670

With a few exceptions, majority of the FRP systems proposed so far are based on Haskell thanks to the powerful language mechanisms that can, for example, realize arrows to abstract time-varying values or signals. However, this fact has constrained the application areas of FRP. Because currently available Haskell processors require relatively large runtimes, it is hard to use a Haskell-based FRP system on resource constrained platforms. Of course, the advance of programming language studies and microprocessor technologies might resolve such situation in the future. However, the need for small-scale and cost-effective devices will remain.

We designed and developed a new FRP language named Emfrp that mainly targets small-scale embedded systems. The term *small-scale* here indicates that the target platform is not powerful enough to run full-fledged operating systems such as Linux. As in the existing FRP systems, Emfrp is based on the notion of behaviors and signals to represent time-varying values. However, we adopt different abstraction mechanisms to evade dynamic memory reclamation (garbage collection) and time-/space-leaks.

To demonstrate our approach, we implemented a compiler and an interpreter of Emfrp[1]. The compiler generates platform-independent ANSI-C code that can be deployed on multiple processors including microcontrollers. The interpreter can be used as a practical testing tool for reactive behaviors in both manual and automated way. We show, in this paper, a simple but non-trivial application to illustrate the power of the language.

The rest of this pater is structured as follows. In the next section, we overview the language with some examples. Section 3 presents a nontrivial application as a case study. In Section 4, we discuss how to test the Emfrp code using the interpreter. Section 5 discusses the FRP style realized by the language and Section 6 briefly explains the implementation of the language. Finally, we show some related works in Section 7 and concludes the paper in Section 8.

2. Emfrp Overview

Emfrp is a purely functional language with features for reactive programming. This section overviews Emfrp's abstraction mechanisms for FRP and other language features.

2.1 Design Considerations

Expressing time-varying values and events is the central topic of FRP language design. Existing FRP languages and frameworks, such as Elm[4] and Yampa[10, 6, 2], treat time-varying values as first-class data using types that encapsulate time dependencies. Types (or type constructors) for the purpose are either built-in (e.g., `Signal` in Elm) or user-defined using sophisticated type constructors such as arrows[7].

[1] Available at: https://github.com/sawaken/emfrp

```
1  module Thermostat
2  in
3    temperature : Int,   # temp sensor
4    pulse1min : Bool     # periodical pulse
5  out
6    switching : Bool     # heater state
7  use
8    Std                  # standard library
9
10 node init[False] switching =
11   if pulse1min then
12     temperature < 50
13   else
14     switching@last     # previous value
```

Code 1. Simple Thermostat Controller in Emfrp

We adopt a different approach for Emfrp. A program in (functional) reactive style is often expressed as a directed graph whose nodes and edges represent time-varying values and their dependencies respectively. The design of Emfrp follows this tradition but in a simple and direct manner. An Emfrp program contains a fixed number of named *nodes* that represent time-varying values. A node may refer to the names of other nodes that it depends on. If node x refers to y (namely, x depends on y), we say that there is an edge from y to x. Emfrp's syntax provides a convenient way of writing programs as graphs.

Because Emfrp is mainly targeted at small-scale embedded systems, we designed the language to have the following characteristics:

- Nodes (time-varying values) are not first-class values in the language. We must, therefore, always specify nodes with their names.
- The language does not provide ways to alter the dependency relation between nodes at runtime. In other words, the graph representation of a program is static.
- Recursion is not allowed in function and type definitions.

Although the design choices shown above greatly reduce the flexibility and dynamicity of the language, they enable us to enjoy the following advantages:

- A developer can concentrate on the semantics of each time-varying value with simple and intuitive abstractions.
- The entire structure of an Emfrp program is more clear and readable than a program written using higher-order functions and type constructors.
- A compiled Emfrp program can run with a small and fixed memory footprint.

We can say that Emfrp chooses simplicity, performance, and small memory footprint rather than flexibility and dynamicity provided by functional languages with higher-order features.

2.2 Programming in Emfrp

Code 1 is an Emfrp program for the controller of a simple thermostat system that has a temperature sensor and a heater. The controller periodically (at 1 minute interval) reads the sensor value and turns on the heater if the measured temperature is less than 50. It keeps the previous heater state otherwise.

Code 1 consists of a single module definition (lines 1–8) that contains a node definition (lines 10–14). Characters from '#' through the end of line constitute a single-line comment. The

```
1  #include "Thermostat.h"
2
3  void Input(int* temperature, int* pulse1min) {
4    /* Your code goes here... */
5  }
6  void Output(int* switching) {
7    /* Your code goes here... */
8  }
9  int main() {
10   ActivateThermostat();
11 }
```

Code 2. I/O Code Skeleton for Thermostat

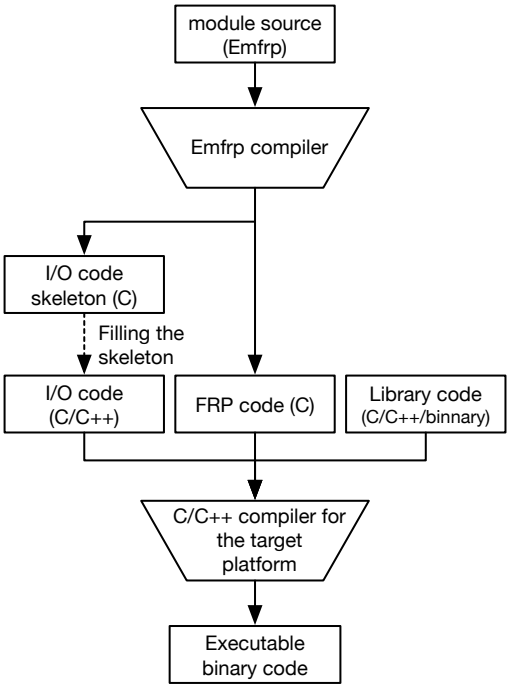

Figure 1. Emfrp Development Workflow

module definition declares two input nodes (`temperature` and `pulse1min`) and a single output node (`switching`) that are time-varying values. The node definition (lines 10–14) descries the behavior of `switching`. The behaviors of the two input nodes should be defined separately (in C/C++).

Because the program is intended to run on an embedded device, the nodes declared as input and output are to be connected to the I/O of the device. Our Emfrp compiler generates a C source file called *I/O code skeleton* as well as the compiled node definitions and functions. The generated I/O code skeleton contains empty function definitions that correspond to the input and output nodes of the original program. The developer should fill the body of the functions with the actual low-level I/O codes in C/C++. Code 2 is the I/O code skeleton generated from Code 1.

Figure 1 depicts the Emfrp development workflow. From an Emfrp source file, the compiler generates two platform-independent ANSI-C source files: FRP code and I/O code skeleton. The FRP code contains the C code implementing the nodes and functions in the Emfrp source. The developer should prepare the I/O code

```
module Iterate
in
  a : Int, b : Int
out
  x, y
use
  Std

node init[1] x : Int = a + y@last
node init[0] y : Int = b * x@last
```

Code 3. Iterate Process in Emfrp

```
#include "Iterate.h"
#include <stdio.h>

void Input(int* a, int* b) {
  scanf("%d %d", a, b);
}
void Output(int* x, int* y) {
  printf("%d %d\n", *x, *y);
}
int main() {
  ActivateIterate();
}
```

Code 4. I/O Code for Code 3

by manually filling the generated I/O code skeleton as explained above. Finally, she/he can obtain the executable binary by compiling and linking these C/C++ source files and platform-dependent libraries.

2.3 The Last Value of a Node

Code 1 contains an expression `switching@last` (line 14). This expression refers the *last value* of the node `switching`. From the viewpoint of the FRP semantics, the last value should refer the value at the snapshot taken a moment ago. In practical, this is the value of the node evaluated at the *last iteration*. An iteration corresponds to an evaluation cycle of the entire program (see Section 6.1). In the following example, we show that how a loop-like execution can be realized in Emfrp.

The module `Iterate` in Code 3 has two input nodes and two output nodes. As Code 4 shows, they are connected to the standard I/O via `scanf` and `printf`. When we feed some inputs like `1 2`, `3 4` and `5 6`, the output should be `1 2`, `5 4` and `9 30`. This behavior is realized because `x` and `y` refer to each other with last values. This example demonstrates that how we can write a simple input-process-output iteration pattern in Emfrp.

In a node definition, a clause `init[exp]` after the keyword `node` specifies the initial value of the node. The initial value specification is necessary for a node to be referred with `@last` in case it is accessed at the first iteration.

As we have explained in Section 2.1, a program in Emfrp forms a directed graph. For example, the graph in Figure 2 represents the program in Code 3. Edges that correspond to the references to the last values are drawn as broken lines[2]. For debugging purpose, our Emfrp compiler has a facility to visualize the graph representations of programs by generating dot[3] files.

[2] We call them *past edges*. See Section 6.1 for details.

[3] File format used in Graphviz (http://www.graphviz.org/)

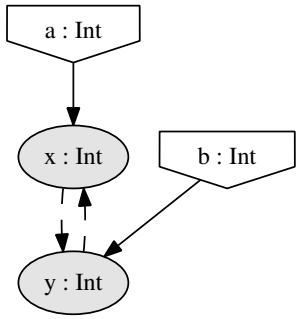

Figure 2. Graph Representation of Code 3

```
type Box[a] =
  RectangleBox(a) | CircleBox(a)

func exchangeBox(b) = b of:
  RectangleBox(x) -> CircleBox(x)
  CircleBox(x) -> RectangleBox(x)
```

Code 5. Type and Function Definitions

2.4 Types and Functions

Emfrp is a strongly-typed language whose type system supports parametric polymorphism and type inference. We can define algebraic data types and functions apart from node definitions. Code 5 is an example including a type declaration and a function definition.

Pattern-matching expressions are also available to take values from algebraic data safely. We don't need explicitly specify types except for input nodes declarations since Emfrp supports ML-like polymorphic type inference.

An important limitation in Emfrp is that we can define neither recursive data types nor recursive functions. Furthermore, the language does not provide any loop constructs. Thanks to these (somewhat severe) constraints, the runtime memory footprint of a program is bounded and time-/space-leaks do not occur. If some iterative behaviors are needed, we are required to use `@last` as explained in the previous subsection instead of using recursions or loops.

2.5 Modules

An Emfrp program consists of one or more modules. A program file (with extension `mfrp`) contains a single module definition followed by node definitions used in the module.

A module can be used as the main module of a program, or as a submodule used in other modules. The main module of a program plays the role of the launch point. A module can be instantiated as a submodule of another module using the language construct `newnode`. Code 6 and Code 7 show an example program consist of two modules. The former is the main module that instantiate the latter twice as the submodule.

3. Case Study: Digital Clock

3.1 Application Overview

The application is an orthodox digital clock displaying the current time as `HH:MM:SS`. We can set the time at any time the program is running using buttons A, B and C. The behaviors of the application is shown in the hierarchical state-transition diagram in Figure 3. Note that the clock continues ticking even while setting the time.

```
module MyMainModule
in x : Int out a, b, c, d use Std

newnode a, b = MySubModule(x, x + 1)
newnode c, d = MySubModule(x, x + 2)
```

Code 6. MyMainModule.mfrp

```
module MySubModule
in x : Int, y : Int
out sum, pro
use Std

node sum = x + y
node pro = x * y
```

Code 7. MySubModule.mfrp

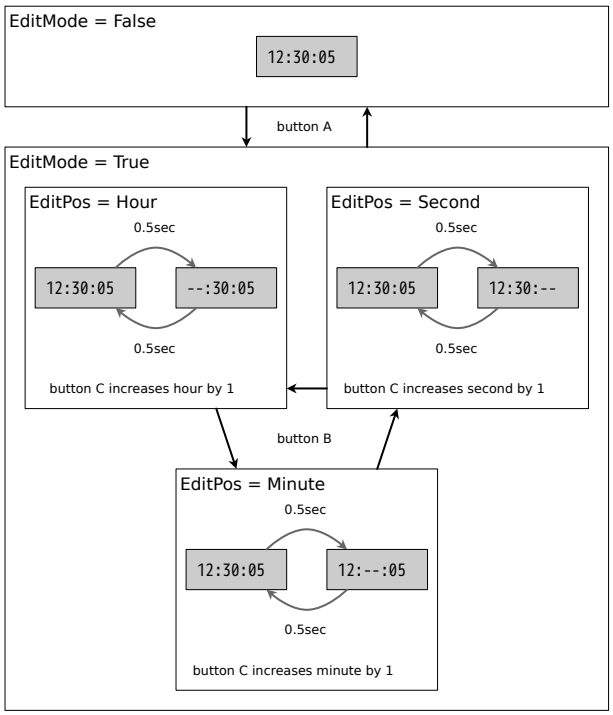

Figure 3. Behavior of the Digital Clock

We have tested the code on mbed LPC1768[4] with mbed application board[5]. The platform has 96MHz ARM Cortex-M3 core with 32KB RAM and 512KB flash memory. The full source code is available on our Github repository[6].

3.2 Declaring and Defining I/O Nodes

Code 14 (in Appendix A) shows the whole source code. In the first part (lines 2–11), the following Boolean-typed nodes are declared as inputs.

[4] https://developer.mbed.org/platforms/mbed-LPC1768/

[5] https://developer.mbed.org/cookbook/mbed-application-board

[6] https://github.com/sawaken/emfrp/tree/develop/examples/LCDClock

```
#include "DigitalClock.h"

void Input(int* btnMode, int* btnNext, int* btnInc
    , int* pulse100ms) {
  /* Your code goes here... */
}
void Output(int* hour, int* min, int* sec, int*
    maskHour, int* maskMin, int* maskSec) {
  /* Your code goes here... */
}
int main() {
  ActivateDigitalClock();
}
```

Code 8. I/O Code Skeleton for DigitalClock

- `btnMode`: becomes `True` only while button A (mode-button) is pressed
- `btnNext`: becomes `True` only while button B (next-button) is pressed
- `btnInc`: becomes `True` only while button C (inc-button) is pressed
- `pulse100ms`: becomes `True` at 100ms interval

In the next part (lines 12–17), the following nodes are declared as outputs.

- `hour`, `min`, and `sec`: represent integer values to be displayed as current time hour, minute, and second respectively
- `maskHour`, `maskMin`, and `maskSec`: represent that currently setting the hour, minute or second respectively. For example, if the value of `maskHour` is `True`, then blinking '`--`' is shown in the hour part of the display.

After these I/O node declarations, type definitions (lines 25–27), function definitions (lines 29–61) and node definitions (lines 66–140) follow. They are used to define the whole behaviors of output nodes. Due to the limitation of space, we do not explain semantics of nodes in Code 14 here. However, the code is fully annotated so that you may be able to read and understand it.

3.3 I/O Code

As mentioned in Section 2.2, in addition to write the main part of the code in Emfrp, it is necessary to write IO code in C/C++ by filling the I/O code skeleton generated by the Emfrp compiler. Code 8 is the I/O code skeleton generated from Code 14 (in Appendix A).

An example I/O code derived from Code 8 is provided as Code 15 (in Appendix B). The code is for the tested platform (described in Section 3.1) and uses the standard mbed library and a user-provided library[7] for the LCD display on the application board.

3.4 Using FRP Modules (Optional)

We can use the submodule feature (described in Section 2.5) for more comfortable reactive programming. For example, we can define a reusable module `PositiveEdge` (Code 9) that detects positive (rising) edges of a Boolean node.

Using this module as a submodule, we can define the node `curMode` (in Code 14) as Code 10. The resulting code is easier to read than the original code that use function `positiveEdge` for the same purpose. Note that the specification of the initial value specification for the input node `btnMode` (line 5 in Code 14) is no longer needed.

[7] https://developer.mbed.org/users/dreschpe/code/C12832_lcd

```
module PositiveEdge
in x : Bool out y : Bool
use Std

node init[False] buf = x
node y = !buf@last && buf
```

Code 9. PositiveEdge.mfrp

```
newnode btnModeEdge = PositiveEdge(btnMode)
node init[Normal] curMode : Mode =
  if
    btnModeEdge
  then
    curMode@last.nextMode
  else
    curMode@last
```

Code 10. Definition of `curMode` using Code 9

4. Testing FRP Code

A program written in Emfrp has good testability. The reason is that each element in Emfrp has no implicit state. Thus we can replay the behavior of a node from is arbitrary state or can feed arbitrary input to it. Those are essential benefits of FRP, however, thanks to its design characteristics, Emfrp has no testing obstacles.

4.1 Testing Functions

The Emfrp interpreter provides `:assert-equals` command to check the equality of two expressions. We can always use it for testing because all Emfrp functions are pure.

```
#@ :assert-equals 3, add(1, 2)
func add(a, b) = a + b
```

As the above code shows, a special comment syntax opening with `#@` allows the interpreter command used as an annotation embedded in the source code.

4.2 Testing Nodes

We can use `:assert-node` command to test a node by regarding it as a function. For example, the command in the code below asserts that the value of the node `xnode` must be 3 if the values of the nodes `foo` and `bar` are 1 and 2 respectively.

```
#@ :assert-node xnode 1, 2 => 3
node from[foo, bar] xnode = foo + bar
```

The current version of the interpreter requires an extra annotation `from[...]` to use `:assert-node`.

4.3 Testing Modules

A module can be tested by checking the values of the output nodes after feeding a combination of values to the input nodes. The Emfrp interpreter provides `:assert-module` command for this purpose. For example, following commands tests the module `Iterate` in Code 3 (in Section 2).

```
:assert-module 1, 2 => 1, 2
:assert-module 3, 4 => 5, 4
:assert-module 5, 6 => 9, 30
```

The values of the output nodes of a module is history-sensitive if the module uses `@last`. It is possible to set the current states of the module to arbitrary values for testing.

```
x : Signal Int
x = ...
lastOfX' : Signal (Int, Int)
lastOfX' = foldp (\a state -> (snd state, a)
    ) (0, 0) x
lastOfX = lift fst lastOfX'
sampler = lift (+1) lastOfX
```

Code 11. Getting Feedbacks in Elm

```
node init[0] x : Int = ...
node sampler = x@last + 1
```

Code 12. Getting Feedbacks in Emfrp

5. Alternative Style for FRP

Elm-style FRP provides one of the most simple and accessible abstractions for reactive programming. The design of Emfrp is inspired by Elm and partly adopts Elm-style FRP. However, we think that programming in Elm is sometimes non-intuitive for reactive programming due to its basic language design influenced by usual functional languages such as Haskell. Emfrp provides more intuitive syntax and semantics to handle time-varying values. The rest of this section discusses the FRP style provided by both languages.

Getting feedbacks, namely, referring the past value of a time-varying value is the key concept in FRP. In Elm-style FRP (including Emfrp), the only way to get feedbacks is by referring the last-value of a time-varying value. In Elm, a time-varying value can only refer to the last value of itself using `foldp`. In contrast, Emfrp provides `@last` notation that allows a node can refer to the last value of an arbitrary node.

This is a major difference between Elm and Emfrp for users to write FRP program. Although Elm's design of syntax focuses on handling time-varying values in the conventional functional syntax similar to Haskell, this is not an intuitive way to express the semantics of time-varying values. Therefore, users are forced to use their brain unnaturally to express application's logic by FRP.

Both Code 11 and Code 12 define the same example in Elm and Emfrp respectively. In the example, time-varying value `sampler` refers last-value of time-varying value `x`. The Emfrp version is highly intuitive because the last value of `x` can be accessed directly by `sampler` using `@last` while both time-varying values should form a pair in Elm.

6. Implementation

Emfrp code is compiled into platform-independent ANSI-C code. This section briefly explains the compilation strategy and runtime behaviors.

6.1 Iterations

As explained in Section 2.1, an Emfrp program can be represented by a directed graph whose nodes and edges correspond to time-varying values and their dependencies respectively. We categorize the edges into two kinds, namely, *past* and *present*. A past edge from node x to y means that y refers to the last value of x (*i.e.*, x`@last`). A present edge from node x to y, in contrast, means that y refers to x without `@last` annotation. For example, the graph representation of `Foo` in Code 13 has two past edges: one from `f` to `b` and the other from `d` to `d`. All other edges are present.

By removing the past edges from the graph representation of a program, we obtain a directed-acyclic graph (DAG). Figure 4 is the

```
module Foo
in a : Int, c : Int, e : Int
out d, f
use Std

node init[0] b : Int = if f@last then a else
     c
node init[0] d = max(b, d@last)
node init[False] f : Bool = e > 0
```

Code 13. A Module with 6 Nodes

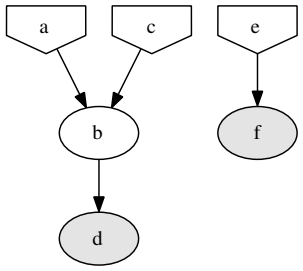

Figure 4. A DAG Representation of `Foo` in Code 13

resulting DAG for `Foo`. Using the topological sorting on the DAG, we have a sequence of the nodes, for example, (a, c, e, b, f, d). For each node in a program, the compiler generates a C function that computes the current value of the node. The sequence of the nodes described above determines the linear evaluation order of such functions that is compatible with the partial order specified determined by the edges in the DAG. This means that the evaluation order obeys the node dependencies.

The Emfrp runtime system updates the current values of the nodes by repeatedly evaluating the sequence of the functions. We call a single evaluation cycle an *iteration*. The last value of a node (node name with `@last`) is realized by the reference to the node value computed in the last iteration.

6.2 Memory Management

Thanks to the static characteristics described in Section 2.1, the size of the runtime memory can be determined at the compile time. Thus, runtime data other than primitive values (such as integers) can be allocated in the static area of the program memory. For such data, the Emfrp runtime system provides a kind of mark-and-sweep GC that runs each time an iteration finishes.

6.3 Performance

As explained in Section 6.1, the Emfrp runtime repeats iterations (evaluation of functions corresponds to nodes). Thus the runtime performance depends on the performance of each function. There is no big gap between an Emfrp node definition and the C function generated for the node. This means that Emfrp's performance bears comparison with C. Only one possible exception is the process of data type instantiations. Due to the process, the runtime system pays additional costs relative to the size of the instantiated data. Our experience so far states that the performance impact regarding such process is not significant. But we leave a through performance evaluation for future work.

6.4 Avoiding Space-/Time-Leaks

In the context of (functional) reactive programming, space-leak means that a program allocates more memory than expected and time-leak means that the program consumes more time than expected. These phenomena are usually caused by the lazy nature of the host language[8] or by history sensitive behaviors[16]. They disturbs the normal execution of the program.

The Emfrp runtime does not allocate data structures such as tuples on the stack or heap of the program memory. As explained in Section 6.2, they are allocated in the static area. This allocation strategy is possible because the language allows neither recursive data structures nor recursive functions, which may require unbounded memory spaces. Excluding recursions from type and function definitions guarantees that we can determine the upper bound of the memory sizes needed (a) to evaluate an arbitrary expression and (b) to allocate an arbitrary data structure. Thus, the upper bound of the runtime memory size can be determined at compile time. The fact implies that Emfrp can avoid space-leak.

Emfrp can also avoid time-leak by its nature. Because the language do not allow recursive functions and do not provide loop constructs, the time for evaluating a node value is always bounded. Moreover, a node can hold only the current and previous (last) values whose sizes are bounded as explained above. In other words, the memory usage of each node is always bounded. Thus, a node cannot keep data that unboundedly glows (*e.g.*, a list). The above facts imply that there are no ways to define a node whose evaluation time glows over time.

7. Related Works

7.1 Céu

Céu[14, 15] is an imperative programming language designed for small-scale embedded systems. The method of describing reactive behaviors in this language is completely different from that of FRP languages including Emfrp. However, Emfrp is inspired by Céu especially on designing the language suitable for resource constrained environments.

For reactive programming, Céu provides simple mechanisms for event handling and synchronous concurrent execution. The **await** statement plays roles of event handling and synchronization points as well as context-switching points. Using this statement and other constructs, we can describe the reactive behaviors without bothered by callbacks and threads that might make the programming of embedded systems a complex task.

Céu does not provide sufficient tools for testing. Because a Céu program may have complex control flows due to the above mentioned language mechanisms, it is generally difficult to offer a easily usable testing mechanism such as simple assertions provided by Emfrp.

7.2 FRP Libraries in Haskell

FRP was originated as a technique for defining time-dependent behaviors in lazy functional languages and hence mainly studied using Haskell[3, 1]. After the publication of the founder paper on Fran[5], majority of previous FRP systems, such as Yampa[10, 6, 2], Reactive Banana[11] and FRPNow[16] are based on Haskell. Problems thought to be peculiar to FRP, such as space- and time-leaks, are, in fact, caused by the attempt to realize FRP in lazy languages like Haskell.

Of course, the contributions of Haskell-based research on FRP are significant. For example, by defining sophisticated types for handling time-varying values, they showed that we can realize safe reactive programming without introducing new and dedicated language constructs. Now, however, some of such types or equivalents are known to be realized by somewhat simple and dedicated

language mechanisms, such as built-in `Signal` type constructor in Elm[4].

As explained in Section 1, the runtime systems of currently available Haskell compilers require relatively large resources. Thus, it is virtually impossible to use Haskell-based FRP systems on small-scale embedded systems.

7.3 Elm

Elm[4] is an FRP language for client-side scripting run on web-browser. Both Elm and Emfrp are pure functional languages. The crucial difference between them is in their basic language design. Elm is based on the lambda-calculus tradition (*e.g.*, ML, Haskell), but Emfrp is not.

For example, first-class, anonymous function (*i.e.*, lambda) is a powerful language mechanism available in most major functional languages. Unfortunately, the mechanism is not suitable for small-scale embedded systems without enough amount of memory and CPU power. The reason is that dynamic memory management is essential to fully enjoy the power of lambdas.

Emfrp throws lambdas (and other convenient language mechanisms) away to be able to utilize resource constrained environments. However, as described in Section 5, `@last` provides better flexibility and simplification than Elm and other Haskell-based FRP frameworks.

7.4 Reactive Extensions

Reactive Extensions (Rx)[12] is a cross-platform library to realize reactive programming along with methodology of *dataflow programming* based on the Observer-Pattern. The methodology can be called as a *non-pure style* FRP that manipulates time-varying values using side-effects. Rx is currently active project and is a low-cost solution for comfortable reactive programming because it is not a framework but just a library. Thus there is no need to migrate our existing program from to a new framework's culture. Although Rx supports various languages, they, at least, require environment in which C++11 works sufficiently. This means that Rx depends on the powerful language features in C++11 such as lambda expressions. Unfortunately, C++11 is currently bit heavy for most small-scale microcontrollers.

8. Concluding Remarks

We have designed and developed Emfrp, a functional-reactive language that is targeted at small-scale embedded systems. For the design of Emfrp, we do not follow the lambda-calculus tradition to make the language suitable for resource constrained environments such as microcontrollers. Instead, we introduced a simple mechanism that realizes the reference to the last values of arbitrary nodes. The mechanism provides better flexibility and simplification than previous FRP languages and libraries. Thanks to the simple structure of the language and its runtime, there are no chances of space-/time-leaks. We presented a digital clock application as a simple but non-trivial example that demonstrates the capabilities of the language.

Future work includes the following:

- Performance evaluation in terms of CPU usage and memory footprint
- Improvement of the compiler in terms of generated code size and speed
- Introduction of flexible data structures that will not disturb the language characteristics described in Section 2.1
- Formal semantics (especially the semantics of `@last`)

Acknowledgments

This work is supported in part by JSPS KAKENHI Grant No. 15K00089.

References

[1] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):52, 2013.

[2] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *ACM SIGPLAN Workshop on Haskell (Haskell 2003)*, pages 7–18. ACM, 2003.

[3] E. Czaplicki. Elm: Concurrent FRP for functional GUIs. Senior thesis, Harvard University, 2012.

[4] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, pages 411–422. ACM, 2013.

[5] C. Elliott and P. Hudak. Functional reactive animation. In *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*, pages 263–273. ACM, 1997.

[6] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[7] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[8] H. Liu and P. Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, 2007.

[9] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, and A. Bromfield. Flapjax: A programming language for Ajax applications. In *24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2009)*, pages 1–20. ACM, 2009.

[10] I. Pembeci, H. Nilsson, and G. Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *4th International Conrefernce on Principles and Practice of Declarative Programming (PPDP 2002)*, pages 168–179. ACM, 2002.

[11] Reactive banana: a library for functional reactive programming. https://wiki.haskell.org/Reactive-banana. (accessed in Jan. 2016).

[12] ReactiveX: an API for asynchronous programming with observable streams. http://reactivex.io. (accessed in Jan. 2016).

[13] G. Salvaneschi and M. Mezini. Towards reactive programming for object-oriented applications. In *Transactions on Aspect-Oriented Software Development XI*, volume 8400 of *Lecture Notes in Computer Science*, pages 227–261. Springer-Verlag, 2014.

[14] F. Sant'Anna, R. Ierusalimschy, and N. Rodriguez. Structured synchronous reactive programming with Céu. In *14th International Conference on Modularity (Modularity 2015)*, pages 29–40. ACM, 2015.

[15] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas. Safe system-level concurrency on resource-constrained nodes. In *11th ACM Conference on Embedded Networked Sensor Systems (SenSys 2013)*, pages 13:1–13:14. ACM, Nov. 2013.

[16] A. van der Ploeg and K. Claessen. Practical principled FRP: Forget the past, change the future, FRPNow! In *20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*, pages 302–314. ACM, 2015.

A. Emfrp Source Code of Digital Clock

```
module DigitalClock
in
  # Is mode-button pushed now?
  # (False is initial value)
  btnMode(False) : Bool,
  # Is next-button pushed now?
  btnNext(False) : Bool,
  # Is increment-button pushed now?
  btnInc(False) : Bool,
  # True/False which toggles by 100ms
  pulse100ms(False) : Bool
out
  # current time to display
  hour, min, sec,
  # Is it now needed to display --
  # instead of two-digits?
  maskHour, maskMin, maskSec
use
  # using Emfrp's standard library
  Std

# Functions
# --------------------

type Mode = Normal | Edit
type EditPos = HourPos | MinPos | SecPos
type Time = Time(Int, Int, Int)

func nextMode(m) = m of:
  Normal -> Edit
  Edit -> Normal

func editable(m) = m of:
  Normal -> False
  Edit -> True

func nextPos(p) = p of:
  HourPos -> MinPos
  MinPos -> SecPos
  SecPos -> HourPos

# note: you can define local variables
#   with expression {(<name>=<exp>)+ <exp>}
#   and you can use pattern match for
#   <name>
func proceedTime(t) = {
  Time(h, m, s) = t
  newS = s + 1
  newM = m + (newS / 60)
  newH = h + (newM / 60)
  Time(newH % 24, newM % 60, newS % 60)
}

func increaseTime(t, dh, dm, ds) = {
  Time(h, m, s) = t
  Time((h+dh)%24, (m+dm)%60, (s+ds)%60)
}

func positiveEdge(a, b) = !a && b

func bothEdge(a, b) = !a && b || a && !b

# Nodes
# --------------------

# mod-10 counter incremented by every 100ms
# note: you can put type annotation
#   following the node name
# note: expression x `f` y is syntax sugar
#   for expression f(x, y)
node init[0] counter : Int =
  if
    pulse100ms `bothEdge` pulse100ms@last
  then
    (counter@last+1)%10
  else
    counter@last % 10

# node to be True only once by every 1sec
node pulse1s : Bool =
  counter == 0 && counter@last != 0

# node switching True/False by every 500ms
node flash : Bool =
  counter < 5

# node representing current time-set mode
# note: you can use pattern match
#   for left side value of node-definition
node init[Normal] curMode : Mode =
  if
    btnMode@last `positiveEdge` btnMode
  then
    curMode@last.nextMode
  else
    curMode@last

# node representing current cursor position
node init[HourPos] curPos : EditPos =
  if
    btnNext@last `positiveEdge` btnNext
  then
    curPos@last.nextPos
  else
    curPos@last

# node representing diffs
#   to add to current time
node (dh, dm, ds) : (Int, Int, Int) =
  if
    curMode.editable &&
      (btnInc@last `positiveEdge` btnInc)
  then
    curPos of:
      HourPos -> (1, 0, 0)
      MinPos -> (0, 1, 0)
      SecPos -> (0, 0, 1)
  else
    (0, 0, 0)

# node representing current time
# note: expression x.f(y) is syntax sugar
#   for expression f(x, y)
node init[Time(0, 0, 0)]
Time(hour, min, sec) as curTime =
  if pulse1s then
    curTime@last.proceedTime
                .increaseTime(dh, dm, ds)
```

```
  else
    curTime@last.increaseTime(dh, dm, ds)

# node representing need of masking
node (maskHour, maskMin, maskSec) =
  if curMode.editable && flash then
    curPos of:
      HourPos -> (True, False, False)
      MinPos -> (False, True, False)
      SecPos -> (False, False, True)
  else
    (False, False, False)
```

Code 14. Emfrp Source Code of Digital Clock

B. I/O Code of Digital Clock

```
#include "DigitalClock.h"
#include "mbed.h"
#include "C12832_lcd.h"

C12832_LCD lcd;
DigitalIn center(p14), up(p15), right(p16);
int current_pulse100ms = 0;

void pulse_toggle() {
  current_pulse100ms ^= 1;
}

void put_digits(int number, int mask) {
  if (mask) {
    lcd.printf("--");
  } else {
    lcd.printf("%02d", number);
  }
}

void Input(int* btnMode, int* btnNext,
           int* btnInc, int* pulse100ms) {
  *btnMode = center.read();
  *btnNext = right.read();
  *btnInc = up.read();
  *pulse100ms = current_pulse100ms;
}

void Output(int* hour, int* min, int* sec,
            int* maskHour, int* maskMin,
            int* maskSec) {
  lcd.cls();
  lcd.locate(45, 10);
  put_digits(*hour, *maskHour);
  lcd.printf(":");
  put_digits(*min, *maskMin);
  lcd.printf(":");
  put_digits(*sec, *maskSec);
  wait(0.01);
}

int main() {
  Ticker ticker;
  ticker.attach(&pulse_toggle, 0.1);
  ActivateDigitalClock();
}
```

Code 15. I/O Code of Digital Clock